Vers un modèle à composants orienté services pour supporter la disponibilité dynamique

Humberto Cervantes

UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I

THÈSE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

Présentée et soutenue publiquement par :

Humberto CERVANTES

Le 29 Mars 2004

VERS UN MODÈLE A COMPOSANTS ORIENTÉ SERVICES POUR SUPPORTER LA DISPONIBILITÉ DYNAMIQUE

Directeurs de thèse :
Jacky Estublier et Jean-Marie Favre

Composition du jury :

F. Ouabdesselam	LSR-IMAG, Université de Grenoble	Président
J.M. Geib	LIFL, Université de Lille	Rapporteur
Michel Riveill	ESSI, Université de Nice	Rapporteur
Philippe Merle	INRIA Lille	Examinateur
Gérard Vandome	BULL	Examinateur
Jacky Estublier	LSR-IMAG, Université de Grenoble	Directeur de thèse
Jean-Marie Favre	LSR-IMAG, Université de Grenoble	Co-directeur

Thèse réalisée au sein du laboratoire LSR - Equipe ADELE avec un financement du gouvernement du Méxique (CONACYT-SEP)

à Gaby,

à Humberto, Ilse et Miguel,

au Méxique.

Grenoble, le 31 Mars 2004

Remerciements

Je remercie tout d'abord le CONACYT et la SEP, les organismes du gouvernement du Méxique qui m'ont donné le financement qui m'a permis de réaliser ces études.

Je tiens à remercier les différentes personnes qui m'ont permis de réaliser ce travail. Jacky Estublier, qui m'a accueilli dans son équipe. Jean-Marie Favre, qui m'a particulièrement soutenu pendant le DEA, la première année et la fin de la thèse. Richard Hall, mon collègue et ami avec qui j'ai travaillé pendant les deux dernières années de la thèse et qui est devenu en quelque sorte un autre directeur de thèse. L'ensemble de l'équipe Adèle, avec qui on a partagé de très bons moments de travail mais aussi de discussion, de cinéma, ainsi que de dégustation de cuisine internationale.

Je souhaite aussi remercier les gens qui ont utilisé le Service Binder et contribué d'une manière ou d'une autre à son amélioration. Les gens de Schneider Electric, et en particulier Marc Chachereau qui a réalisé une relécture de mon document de thèse. Les gens de Ascert, et en particulier Rob Walker qui a contribué avec des retours très importants. Merci aussi à Didier Donsez qui a beaucoup promu ce projet et qui a toujours beaucoup d'idées pour l'améliorer.

Je remercie les membres du jury qui m'ont fait l'honneur de participer à la soutenance. Je tiens à remercier Philippe Merle pour tous les commentaires qu'il à réalisé sur mon document.

Finalement, je remercie particulièrement mon épouse Gaby qui est toujours à mes côtés, ainsi que mes parents Humberto et Ilse et mon frère Miguel, qui m'ont toujours soutenu. Pour terminer je souhaite remercier Joseph et Araceli, David et Kattia, Sergio et Andrea ainsi que tous mes autres amis qui m'ont aidé et accompagné le jour de ma soutenance.

Merci, Gracias, Thanks !

Humberto

Résumé

L'approche à composants promeut la construction de logiciels à partir de l'assemblage de briques logicielles réutilisables appelées composants. Cette approche ne suppose cependant pas que les fonctionnalités offertes par les composants puissent être disponibles dynamiquement. La disponibilité dynamique fait référence à la situation où des fonctionnalités fournies par des composants qui forment une application deviennent indisponibles ou disponibles de façon continue et en raison de causes diverses, comme le déploiement des composants à l'exécution. Les changements dans les fonctionnalités ont lieu au cours de l'exécution de l'application et sont en dehors de son contrôle. Pour les supporter, une application doit être capable de s'adapter, par exemple en cherchant une fonctionnalité remplaçante ou en incorporant des nouvelle fonctionnalités. Bien que cette situation puisse être supportée dans les modèles à composants actuels, ce support n'est pas explicite et doit être réalisé à travers la programmation. Ceci résulte dans un mélange de logique applicative et de logique d'adaptation dédiée à la gestion de la disponibilité dynamique.

Ce travail propose un moyen de supporter la disponibilité dynamique dans un modèle à composants à partir d'une approche qui introduit d'un côté des concepts de l'approche à services dans le modèle à composants et d'un autre des concepts permettant à un environnement d'exécution, associé au modèle, d'adapter dynamiquement une application à partir d'informations fournies par les composants qui la constituent. Ce travail présente d'abord une étude des approches à composants et à services, et ensuite les concepts et l'implémentation d'un modèle à composants orienté services. Cette implémentation a été évaluée dans un contexte industriel et a été libérée comme un projet de source libre.

Mots Clés

Modèle à Composants, Plate-forme à Services, Disponibilité dynamique, Adaptation dynamique

Toward a service-oriented component model to support dynamic availability

Summary

Component orientation promotes the construction of applications from the assembly of reusable building blocks called components. This approach, however, does not assume that functionalities provided by components may be dynamically available. Dynamic availability represents the situation where functionalities provided by the components that constitute an application continually become unavailable or available for different reasons, such as component deployment activities that take place during execution. Changes in the functionalities occur during application execution and potentially outside of its control. To support these changes, an application must be capable of adapting itself, for example by searching a replacing functionality or by incorporating new functionalities. Although this situation can be supported in existing component models, support is not explicit and must be handled programmatically. This results in a mixture of application logic and adaptation logic, the latter being responsible for managing dynamic availability.

This work presents a way to support dynamic availability in a component model based on an approach that introduces, on one side, concepts from service orientation into a component model, and on the other concepts and that allow an execution environment associated to the model to perform dynamic adaptation of an application based on information provided by its constituent components. This work presents an overview of component and service approaches which is then followed by the concepts and the implementation of a service oriented component model. This implementation has been evaluated in an industrial context and has been published as an open source project.

Keywords Component orientation, service orientation, dynamic availability, dynamic adaptation

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte

L'évolution technologique qui a lieu dans l'actualité est accompagnée de façon parallèle par une évolution dans les techniques de construction de logiciels destinées à faire face à des contraintes qui autrefois ne jouaient pas un rôle décisif dans cette activité. L'augmentation de la taille des programmes nécessite la prise en compte de l'intervention de différents acteurs dans le cycle de vie de développement, le besoin de construction d'applications dans des délais plus courts ainsi que la spécialisation dans les tâches de construction motivent la réutilisation du code, la disponibilité de moyens tels que l'accès aux réseaux ou aux bases de données demandent la prise en compte des aspects tels que la distribution ou la persistance.

Ces problématiques, dont certaines ont déjà été identifiées et traitées par des méthodologies telles que l'approche modulaire et l'approche orientée objet, sont à l'origine d'une approche apparue récemment dans le domaine du génie logiciel : il s'agit de l'approche à *composants*, qui promeut la construction d'applications à base de briques logicielles. Bien qu'il n'existe pas une définition consensuelle de ce qu'est un composant, la définition donnée par Szyperski [Szy98] est reprise dans cette thèse car elle est amplement référencée dans la littérature. D'après cette définition, un composant est une unité binaire de composition avec des interfaces contractuellement spécifiées et des dépendances de contexte explicites ; il peut être déployé indépendamment et est sujet à composition par des tierces. L'approche à composants, qui peut être vue comme une évolution de l'approche orientée objet, jouit actuellement d'une grande popularité, qui se manifeste par l'existence d'une grande variété de modèles à composants soutenus par des acteurs industriels majeurs. Un aspect qui n'est cependant pas traité de façon courante dans cette approche (du moins dans les modèles industriels) est celui des changements dans la structure d'une application lors de son exécution. Parmi les changements qui peuvent avoir lieu lors de l'exécution d'une application, cette thèse s'intéresse en particulier à ceux dé-

coulant de la *disponibilité dynamique* des composants. La disponibilité dynamique fait référence au fait que la présence des composants qui constituent une application n'est pas statique, mais peut changer lors de l'exécution d'une application, potentiellement en dehors du contrôle de celle-ci. Un composant peut devenir disponible ou indisponible alors qu'une application qui peut l'utiliser ou qui l'utilise est en exécution.

La disponibilité dynamique fait, cependant, partie des hypothèses d'une autre approche récente et qui est couramment en vogue : l'approche à *services*. Cette approche partage avec l'approche à composants l'idée que les applications sont assemblées à partir de briques logicielles, qui dans cette approche sont appelées services. Un service est un comportement défini de façon contractuelle [BC01] et dont le contrat est rempli par un fournisseur de services. La différence principale entre l'approche à composants et l'approche à services est que la deuxième se focalise sur la description et l'organisation des services afin de supporter leur découverte dynamique en temps d'exécution [Bur00]. La nature des services oblige à attendre le moment de l'exécution pour les incorporer dans une application. Bien que dans l'approche à services, la disparition ou l'arrivée de services pendant l'exécution est envisagée, cette approche ne définit pas le comportement que doit avoir une application face à ce type de changements pendant son exécution. Finalement, il est important de mentionner que du fait que l'approche à services se concentre sur les interactions ayant lieu lors de l'exécution, elle ne traite pas des aspects tels que la livraison et le déploiement.

1.1.1 Problématique

Cette thèse étudie la problématique de l'introduction de la disponibilité dynamique dans un modèle à composants et des moyens visant à permettre à des applications construites à base de composants de s'adapter de façon autonome aux changements de disponibilité des composants. Une application doit entre autres être capable de s'adapter, pendant son exécution, au départ d'un composant, par exemple en cherchant un substitut, ou bien être capable d'incorporer des fonctionnalités fournies par un nouveau composant qui devient disponible.

Dans ce travail, la disponibilité dynamique représente plus concrètement deux situations. Premièrement, l'introduction ou retrait d'instances de composant d'un environnement où s'exécute une application par un acteur autre que l'application, et deuxièmement l'introduction ou retrait de classes de composants en raison des activités de déploiement continu.

La vision de ce travail est que le support de la disponibilité dynamique par rapport à ces deux situations fournit une base qui pourra, dans le futur, permettre de construire des applications pouvant s'adapter à diverses situations et permettra notamment d'explorer des approches nouvelles, telles que la sensibilité au contexte [DA00]. Dans ce cas, l'in-

formation contextuelle, telle que l'emplacement d'un utilisateur, dirige l'introduction ou le retrait de composants dans un système pendant son exécution. Si les applications sont capables de s'adapter par rapport à la disponibilité dynamique de ces composants, elles seront éventuellement en mesure de s'adapter aux changements contextuels.

Trois exemples décrits à la suite illustrent des cas dans lesquels la disponibilité dynamique se manifeste :

Exemple 1 :

Une application supportant l'interaction avec un utilisateur et formée à partir d'un ensemble d'outils pouvant être incorporés de façon indépendante, est installée sur un système dans lequel certains de ces outils sont installés au moment où une carte à puce est insérée dans le système. Les outils installés depuis la carte à puce correspondent par exemple à des outils administratifs qui ne doivent pas être présents de façon permanente. Lorsque la carte se trouve présente, l'application incorpore les outils contenus dans la carte à puce, lorsque celle-ci est retirée, ces outils sont retirés. L'insertion et le retrait de la carte à puce dans le système peuvent être réalisés à tout moment pendant l'exécution de l'application. Dans cet exemple, la disponibilité dynamique des outils contenus dans la carte à puce est introduite par la présence physique de la carte à puce.

Exemple 2 :

Un système de gestion d'un réseau d'appareils électriques se trouve en opération constante pour surveiller l'opération des différents appareils présents dans le réseau. A tout moment un appareil peut être branché ou débranché du réseau et un administrateur doit être en mesure d'installer, ou de retirer, du système de gestion des composants permettant de réaliser la surveillance de l'appareil en question. Au moment de l'ajout ou du retrait de ces modules, l'exécution du système ne doit pas être interrompue. Dans ce cas, la disponibilité dynamique des composants est causée par une interaction avec un utilisateur qui administre le système.

Exemple 3 :

Un utilisateur dispose d'un assistant personnel capable de se connecter à des réseaux non filaires ; au moment d'arriver dans un aéroport, l'assistant se connecte à un réseau disponible à l'intérieur de celui-ci. L'utilisateur décide alors de rédiger un courrier électronique pour transmettre l'information de son vol à un correspondant. Au moment d'écrire un courrier électronique, l'application de courrier lui fournit automatiquement des fonctionnalités permettant d'insérer des informations concernant des vols. Ces informations sont fournies par un composant obtenu à partir du réseau de l'aéroport, et qui n'est disponible qu'à l'intérieur de l'espace physique de celui-ci. Au moment où l'utilisateur quitte

les lieux, le composant est retiré automatiquement de l'application de courrier électronique. Dans ce cas, des changements dans le contexte d'utilisation du système sont à l'origine de la disponibilité dynamique.

1.1.2 Objectif de la thèse

L'objectif de cette thèse est de montrer que la disponibilité dynamique de composants dans un modèle à composants peut être traitée à travers l'introduction de concepts de l'approche à services dans un modèle à composants, ainsi que par l'extraction, du code des composants, de la logique permettant aux applications de s'adapter. Ces concepts donnent lieu à une approche appelée *modèle à composants orienté services*.

Le modèle à composants orienté services permet de construire des applications, à base de *composants à services*, qui supportent la substitution, l'ajout ou le retrait d'instances de composants pendant l'exécution. Les changements au niveau des instances sont gérés par un environnement d'exécution associé au modèle à partir d'informations définies au niveau des composants, ce qui permet de séparer la logique d'adaptation du code applicatif. Les caractéristiques des composants à services et la gestion de l'adaptation résultent dans des applications qui sont non seulement capables de s'adapter mais aussi de s'assembler dynamiquement au moment de l'exécution.

1.1.3 Contributions de la thèse

Pour proposer un modèle à composants qui combine des concepts de l'approche à composants et à services, il est indispensable de comprendre les concepts de ces deux approches. La première contribution de cette thèse est la réalisation d'études détaillées des approches à composants et à services. Ces études sont basées sur les concepts présents dans un certain nombre de modèles à composants et de plateformes à services qui sont décrits et comparés. L'intérêt de ces études repose sur le fait que ces approches étant récentes, leurs concepts ne sont souvent pas clairement définis. Cette situation se voit renforcée par le fait que dans la littérature, la description des aspects technologiques est souvent favorisée ou confondue avec celle des aspects conceptuels.

La deuxième contribution de cette thèse concerne la description des concepts du modèle à composants orienté services ainsi que sa réalisation. La réalisation du modèle et de son environnement d'exécution est présentée en deux temps. Dans un premier temps, seulement le déploiement continu est considéré comme étant la source de la disponibilité dynamique des composants. Dans un deuxième temps, des instances de composant à services peuvent être introduites ou retirées pendant l'exécution en dehors des activités de déploiement.

L'environnement d'exécution du modèle à composants orienté services, appelé Service Binder, est construit au dessus de la plateforme de services OSGi. Le Service Binder

a été libéré comme un projet de source ouverte qui a été utilisé dans deux projets industriels qui sont présentés comme des évaluations. D'autres évaluations réalisées au sein du laboratoire sont aussi présentées.

1.2 Plan de la thèse

La problématique traitée dans ce travail résulte de l'étude menée au long de la thèse autour des approches à composants et à services, et pour cette raison une place importante est consacrée à l'étude de ces approches. Ce document se divise en trois parties :

La première partie présente une étude des approches à composants et à services. Cette partie se divise en trois chapitres :

- Le chapitre 2 décrit brièvement un ensemble d'approches considérées comme étant des antécédents des approches à composants et à services. Ces approches incluent : l'approche modulaire, l'approche de description architecturale et l'approche orientée objet.
- Le chapitre 3 présente un étude détaillée de l'approche à composants. Cette étude présente les concepts principaux de cette approche, suivis d'une description et comparaison d'un ensemble de modèles à composants (COM, JavaBeans, EJB, CCM et Fractal). Pour chacun de ses modèles, une description détaille le domaine d'application ainsi que les concepts supportés par le modèle en question. A la fin de ce chapitre, un tableau de synthèse permet de comparer les divers modèles présentés.
- Le chapitre 4 présente une étude détaillée de l'approche à services, de façon parallèle à celle réalisée pour l'approche à composants. Cette étude présente les concepts principaux de cette approche, suivis d'une description et comparaison d'un ensemble de plateformes à services (Courtier CORBA, BeanContext, Jini, OSGi et services web). Pour chacune de ces plateformes, la description détaille le domaine d'application ainsi que les concepts supportés. Ce chapitre se termine par un tableau de synthèse permettant de comparer les diverses plateformes présentées.

La deuxième partie de ce document se concentre autour du modèle à composants orienté services :

- Le chapitre 5 présente les concepts principaux du modèle à composants orienté services. Ce chapitre se concentre particulièrement sur les aspects relatifs à l'adaptation et à la gestion de celle-ci lors de l'exécution.
- Le chapitre 6 présente le concept d'instance de déploiement ainsi que la réalisation de l'environnement d'exécution du modèle à composants orienté services, appelé Service Binder, au dessus de la plateforme de services OSGi. Le concept d'instance de déploiement permet de traiter la disponibilité dynamique par rapport aux activités de déploiement continu. Ce concept permet de construire des applications compatibles avec la plateforme de services sous-jacente. Des exemples d'applica-

tions obtenus à partir d'évaluations industrielles de l'implémentation du modèle à composants sont présentées ainsi qu'un positionnement par rapport à la plateforme de services.

– Le chapitre 7 présente les concepts relatifs à la création d'applications à partir d'instances dynamiques, c'est à dire des instances de composant pouvant être introduites et retirées du système pendant l'exécution. Ces concepts sont celui de fabrique et d'espace de résolution. Ce chapitre présente aussi la proposition d'un mécanisme permettant de décrire et de gérer une composition d'instances de composants à partir d'une description. Des prototypes qui ont été réalisés dans le laboratoire pour évaluer ces travaux sont aussi présentés.

Ce document se termine par une conclusion ainsi que les perspectives de ce travail.

Première partie

Approches à Composants et à Services

Chapitre 2

Antécédents

2.1 Introduction

Ce chapitre présente trois approches de construction logicielle qui ont précédé et eu une large influence sur les approches à services et à composants. Les approches qui sont présentées sont l'approche modulaire, l'approche de description architecturale et l'approche orientée objet. Ces approches sont présentées car elles introduisent des bases qui sont ensuite réutilisées dans les chapitres consacrés à l'étude des approches à services et à composants.

2.2 Approche Modulaire

L'approche modulaire est basée sur le concept de *module*. Parnas [Par72] décrit ce concept comme étant une tâche particulière ("work assignment") qui dans la pratique est matérialisée par un paquetage contenant des sous-routines et des éléments de données qui sont accessibles à travers une interface bien définie. L'importance de l'approche modulaire est qu'elle introduit des principes tels que l'encapsulation à travers la séparation entre interfaces et implémentations, la compilation séparée, la spécialisation dans les tâches de développement du logiciel et la réutilisation du fait qu'un même module peut être utilisé dans la construction de différents systèmes.

Le concept de module a donné lieu aux langages d'interconnexion de modules et a été introduit dans divers langages comme un concept de première importance.

2.2.1 Langages d'Interconnexion de Modules (MILs)

La programmation modulaire apparaît à la fin des années 1970, et se base sur l'idée soutenant que la programmation de systèmes se divise en deux activités différentes : la *programmation globale* (“in-the-large”) et la *programmation détaillée* (“in-the-small”)

```
module ABC                  Nom du module
  provides a,b,c            Ressources fournies
  requires x,y              Ressources requises
  consist-of function XA, module YBC   Constituants
    function XA
      must-provide a
      requires x
      has-access-to module Z
      real x, integer a
    end XA
    module YBC              Module imbriqué
      must-provide b,c
      requires a,y
      real y, integer a,b,c
    end YBC
end ABC
```

FIG. 2.1 – Exemple de MIL

[DK76]. La programmation globale concerne la structuration de grandes quantités de modules pour former un système tandis que la programmation détaillée concerne la construction des modules en eux mêmes. Le fait que ces deux activités diffèrent justifie l'existence de langages distincts pour les réaliser. Ainsi, pour la programmation détaillée, les langages 'classiques' sont considérés comme adéquats. Cependant, en ce qui concerne la programmation globale, de nouveaux langages spécifiques sont crées : les langages d'interconnexion de modules (ou MILs) [PDN86].

Les langages d'interconnexion de modules permettent de décrire de manière formelle la structure d'un système logiciel par rapport a un ensemble de modules et à leurs interconnexions. Ces langages expriment les coopérations entre modules en décrivant le flot de ressources entre ces modules. Une ressource est toute entité (variable, constante, procédure, etc.) pouvant être nommée, dans un langage de programmation, et pouvant devenir disponible pour être référencée par un autre module dans un système logiciel. Un programme MIL peut être analysé pour vérifier l'intégrité d'un système ainsi que la compatibilité entre les modules. Les MIL sont normalement employés après qu'un système a été analysé, évalué et conçu, et le code MIL doit être maintenu lors de l'implémentation et ensuite employé pour la maintenance du système.

Un exemple de description dans le langage MIL 75 est montré dans la figure 2.1. Les primitives syntaxiques de ce MIL qui décrivent le flot de ressources entre modules sont : `provide`, `require`, `has-access-to` et `consists-of`. Ces primitives peuvent de plus être précédées du qualificatif `must` pour différentier des ressources optionnelles ou obligatoires.

2.2.2 Langages avec support de modules

La plupart des langages procéduraux fournissent du support pour le concept de module : des procédures peuvent être définies comme appartenant à un module, elles peuvent être exportées pour les rendre accessibles à d'autres modules qui les importent. Des exemples de tels langages son C, Modula [Nel91] et Ada [Bar98].

2.2.2.1 Ada

Le langage Ada est fondé sur un système bien défini de modularisation et supporte la compilation séparée des modules appelés *packages*. Même si la compilation séparée est permise, les caractéristiques du langage obligent à suivre un ordre précis lors de sa réalisation.

Un package Ada fait référence à une collection d'entités liées entre elles et peut être composé de procédures, fonctions, variables, constantes, types, sous-types et même d'autres packages. Le package est constitué d'une interface et de son implémentation et peut éventuellement contenir une section d'initialisation qui est exécutée une seule fois, au moment du chargement du package. Pour utiliser les fonctionnalités d'un package dans un autre, il est nécessaire de référencer explicitement le premier package (à l'aide de la clause `with`).

2.3 Approche de Description Architecturale

L'*architecture* d'une application décrit la structure globale d'un système informatique comme une collection de composants qui interagissent entre eux [SG96]. La description de l'architecture revêt une importance particulière du fait qu'une telle description permet de raisonner sur un système à un niveau d'abstraction supérieur à celui qui peut être obtenu à partir de la lecture directe du code source. Cette section décrit les *langages de description d'architecture* (ADL), qui sont les successeurs des MIL, et dont le but est de décrire une architecture pour pouvoir ensuite l'analyser ou la simuler.

2.3.1 Langages de Description d'Architecture

Les langages d'architecture sont des notations formelles destinées à représenter l'architecture d'un système logiciel en vue de son analyse. L'utilisation des ADL se situe principalement au moment de la conception d'un système. Les ADL ont été conçus pour être accompagnés d'outils permettant d'analyser ou de simuler les architectures pouvant être décrites [MT00].

Dans les ADL, l'architecture d'un système est décrite principalement en terme de *composants* qui implémentent des *interfaces*, de *connecteurs* et de leurs *configurations*

(ou compostions) [GMW00]. Un composant est une unité de calcul ou un entrepôt de données (par exemple un client, serveur, base de données, etc...). Une interface spécifie les services (messages, opérations et variables) que le composant fournit. Les connecteurs modélisent les interactions entre les composants à travers leurs interfaces ainsi que les règles qui gouvernent ces interactions. Une configuration (ou composition) représente un graphe de composants connectés entre eux à l'aide de connecteurs, et ce graphe peut devenir lui même un autre composant, ce qui permet aux ADL de supporter des descriptions de *compositions hiérarchiques*.

Un nombre important de langages de description d'architecture ont été conçus pour des domaines spécifiques. Ces ADL, incluent C2 [Med95], qui supporte la description de systèmes d'interaction homme-machine, Rapide [Luc96], qui permet de réaliser la simulation de systèmes, Wright [AGD97], qui supporte la spécification formelle et l'analyse des interactions entre les composants et finalement ACME [GMW00], qui se veut comme étant un ADL générique.

Dans [Sen00], une différentiation est faite entre les langages de description d'architecture et les langages de configuration par rapport à la possibilité des composants d'être instantiés. Dans les langages de configuration, la configuration est décrite en terme d'instances de composant. Des exemples de ces langages sont Darwin [MK96] et OCL (Olan Configuration Language) [BAKR95]. Ces langages incluent en plus des constructions permettant de spécifier des aspects dynamiques.

2.3.1.1 Darwin

Le langage de l'ADL Darwin permet de décrire l'architecture de systèmes distribués comme des hiérarchies de configurations qui représentent des ensembles d'instances de composant connectées entre elles, bien que Darwin ne propose pas le concept de connecteur. Ce langage fournit des constructions permettant de spécifier l'instantiation des composants : l'instantiation peut être *paresseuse* ou bien *dynamique*. L'instantiation paresseuse implique qu'une instance de composant n'est créée qu'au moment où elle est appelée (cas du Client dans la figure 2.2), l'instantiation dynamique permet de créer une instance de composant au moment de réaliser une connexion ; cependant cette instance (qui n'est pas nommée) n'est pas accessible aux autres instances.

2.3.1.2 OLAN

Le langage de configuration d'OLAN, appelé OCL [BBRVD98], est un ADL conçu pour les systèmes répartis. OCL est inspiré du langage Darwin mais inclut en plus les concepts de *connecteur* et de *collection*. Une collection, qui est identifiée par un nom et un type de composants, représente un ensemble d'instances de composants qui peuvent être contrôlées de façon dynamique à travers le langage de configuration. De plus, le

```
component Server {
   provide receiveRequest <port,string>;
}
component Client {
   require sendRequest<port,string>;
}
component System {
   inst
     server1 : Server;
     client :  dyn  Client;
   bind
       client.sendRequest--compte.receiveRequest;
}
```

FIG. 2.2 – Exemple Darwin

nombre d'instances contenues dans une collection peut évoluer pendant l'exécution de l'application. L'évolution de l'ensemble d'instances de composant est contrôlée par un connecteur spécifique qui est lié à la collection. L'accès aux membres de la collection est réalisé à travers un mécanisme qui permet de communiquer avec un sous ensemble des membres de la collection du moment qu'ils satisfont certaines propriétés.

2.4 Approche Orientée Objet

L'approche orientée objet peut être vue comme une évolution de l'approche modulaire, même si les origines de la programmation orientée objet se situent autour des années 1960, avec le langage SIMULA [Hay03]. La programmation orientée objet est fondée sur l'idée qu'une solution logicielle à un problème peut être réalisée en modélisant le problème comme un ensemble d'objets travaillant en collaboration.

2.4.1 Concepts

Cette section décrit les concepts principaux de l'approche orientée objet.

2.4.1.1 Principes fondamentaux

Un *objet* est une abstraction d'un concept existant dans la réalité. Concrètement, il s'agit d'une entité qui encapsule des données (son état) et un comportement (des opérations sur l'état). Un objet est identifié de façon unique, et plusieurs *instances*, ou copies, d'un même objet peuvent exister. Les instances sont créées à partir d'un plan appelé *classe* ou bien à partir d'un autre objet qui est alors utilisé comme *prototype* des instances. L'accès à l'état d'un objet doit théoriquement se faire de manière exclusive à travers un ensemble de méthodes définies dans la classe de l'objet.

2.4.1.2 Héritage

L'*héritage* permet à une classe d'obtenir des données et comportement à partir d'une autre classe appelée classe *de base* ou *super-classe*. La classe qui hérite, ou *sous-classe* peut alors rajouter des éléments aux données de la super-classe ainsi que spécialiser son comportement en redéfinissant les méthodes de la classe de base.

Bien que l'héritage est souvent décrit comme un moyen d'organisation (car une abstraction qui hérite d'une autre est une spécialisation de l'abstraction de base) et de réutilisation de classes, il n'est pas libre de problèmes. La réutilisation à travers l'héritage se fait au moyen de la programmation par différence ; l'idée essentielle est qu'il suffit d'hériter d'une classe et de modifier certaines parties de son comportement pour créer une nouvelle classe différente de la classe de base, mais qui réutilise certaines parties de cette dernière. Ce mécanisme de réutilisation ne prend cependant pas en compte la possibilité de la classe de base d'évoluer, ce qui peut engendrer des problèmes au niveau des sous classes. Ce problème, appelé le problème de la *classe de base fragile* [MS98] est un problème important qui limite l'utilisation de l'héritage comme mécanisme de réutilisation.

2.4.1.3 Polymorphisme

Le polymorphisme est causé par la liaison tardive des méthodes d'un objet : une référence vers une classe de base peut contenir lors de l'exécution un objet créé à partir d'une sous-classe. L'appel à une méthode déclarée dans la classe de base, qui est redéfinie dans la sous-classe, engendre un appel à la méthode de la sous-classe. Un exemple de polymorphisme est montré dans la figure 2.3 ; au moment de l'appel de la méthode `receive` de la classe `Client`, l'objet qui exécute cette méthode peut être créé à partir d'une sous classe de `Client`, telle que `SpecializedClient`.

2.4.2 Méthodologie de développement

La programmation orientée objet est basée sur une méthodologie de développement dans laquelle un problème du monde réel est modélisé en terme d'abstractions. Ces abstractions donnent lieu ensuite à un ensemble de classes qui correspondent directement à des concepts du problème. Les classes obtenues à partir de l'analyse d'un problème sont ensuite complétées, au moment de la conception, par de nouvelles classes qui représentent des objets qui ne correspondent pas directement à des concepts du problème, mais qui sont nécessaires pour l'implémentation d'une application. La réalisation est effectuée typiquement dans un langage orienté objet tel que Java ou C++.

Il faut noter que le déploiement (qui est le processus qui couvre des activités telles que la configuration, installation, mise à jour, reconfiguration et des-installation) n'est souvent pas considéré de manière explicite dans le processus de développement ; il se situe après la livraison d'un système et est souvent laissé à la charge des utilisateurs [Hal97].

2.4.3 Langages orientés objet

Un langage est considéré comme étant orienté objet lorsqu'il supporte le concept de classe ainsi que celui d'héritage [Weg87]. Divers langages orientés objet existent, parmi lesquels figurent Smalltalk, C++ et Java. La figure 2.3 présente un exemple de programme Java. Cet exemple montre des concepts tels que l'héritage ainsi que le polymorphisme.

2.4.4 Mécanismes de réutilisation

La conception d'un système inclut l'activité de recherche de moyens de réutilisation qui permettront, lors de l'implémentation du système, d'éviter d'écrire un système à partir de zéro. Deux mécanismes de réutilisation sont présentés ici : les patrons de conception et les frameworks.

2.4.4.1 Patrons de conception

Les patrons de conception sont popularisés au milieu des années 90 par le livre de Gamma et Al. : "Design Patterns, Elements of Reusable Object-Oriented Software" [GA95] qui propose un catalogue de divers patrons de conception. Les patrons, tels qu'ils sont définis dans le livre, sont des description d'objets et classes qui communiquent et qui sont adaptés pour résoudre un problème général de conception dans un contexte particulier. Les patrons motivent la réutilisation de solutions conceptuelles pouvant être appliquées dans les langages orientés objet.

2.4.4.2 Frameworks

Un *framework* (cadre de conception) est un ensemble de classes qui définit un plan abstrait orienté à résoudre des problèmes liés à un domaine particulier [JF88]. Le framework définit des classes abstraites pour un ensemble d'abstractions principales ; ces classes abstraites doivent être concrétisées par des sous-classes pour pouvoir exécuter le framework. Par exemple, un framework pour construire un compilateur peut inclure des classes abstraites pour le parseur, la table des symboles, le vérificateur de types et le générateur de code ; des classes concrètes pourraient exister pour divers langages tels que Java, C++, etc...

Un framework permet de faire de la réutilisation à un niveau de granularité supérieur à celui qui peut être réalisé à partir d'une classe à travers l'héritage. A différence des patrons de conception, les frameworks sont spécifiques pour un domaine d'application, et ils sont plus concrets que les patrons, car un framework est du code qu'il faut compléter alors qu'un patron est une solution qu'il faut introduire dans le code que l'on écrit ; un framework peut d'ailleurs inclure lui même un certain nombre de patrons dans sa conception.

```
class Client  // Classe de base
{
   private Server server;
   Client(Server s)
   {
     server = s;
   }
   public void receive(String data)
   {
     ...
   }
}
class SpecializedClient extends Client  // sous classe
{
   SpecializedClient(Server s)
   {
     super(s);
   }
   ...
}
public class Server
{
   private List clients = new ArrayList();
   public Server()
   {
     initialize();
   }
   public void addClient(Client c)
   {
     clients.add(c);
   }
   public void startServing()
   {
     Iterator it=clients.iterator();
     while(it.hasNext()){
       Client currentClient = (Client)it.next();
       currentClient.receive(data);  // appel polymorphique
     }
   }
   ...
}
public class System
{
   public System()
   {
     Server s = new Server();  // création d'un objet serveur
     s.addClient(new SpecializedClient());
     s.addClient(new OtherClient());
     s.startServing();
   }
}
```

FIG. 2.3 – Exemple de programme Java

Les frameworks sont souvent divisés en deux catégories : les frameworks à *boite blanche* et les frameworks à *boite noire*. Dans le cas du framework à boite blanche, il est nécessaire de connaître la structure interne du framework pour pouvoir l'utiliser. Un framework est catégorisé comme étant de type boite noire quand l'héritage est remplacé par la composition d'instances obtenues à partir des classes du framework. Un framework de ce type fournit typiquement un ensemble de classes "prêtes à l'emploi" et qui ne nécessitent, à priori, pas de connaissances sur la structure interne du framework. Un exemple de framework de ce type est le framework Swing du langage Java, dédié à la construction d'interfaces graphiques. Dans ce framework, les classes abstraites sont définies par des interfaces et du fait qu'un certain nombre d'implémentations par défaut de ces interfaces sont fournies, le framework est prêt à l'emploi. Un problème des frameworks est qu'ils dépendent fortement de l'héritage ce qui rend très difficile l'évolution d'un framework sans causer de dommage pour les sous-classes qui le concrétisent [MB00].

2.5 Synthèse

Ce chapitre a présenté trois approches qui peuvent être considérées comme des antécédents des approches à composants et à services. L'approche modulaire introduit un nombre de principes importants qui sont l'encapsulation à travers la séparation entre interfaces et implémentations, la compilation séparée, la spécialisation dans les tâches de développement du logiciel et la réutilisation du fait qu'un même module peut être utilisé dans la construction de différents systèmes. Une limitation de l'approche modulaire est qu'elle ne définit pas de mécanismes permettant de personnaliser un module, c'est à dire de faire des variations sur un module existant [WD01]. Une limitation concernant les MIL est qu'ils sont employés pour réaliser l'analyse d'un système mais sont séparés de l'implémentation même du système.

Les ADL, qui sont une évolution des MIL, se focalisent sur description de systèmes à un haut niveau, ils différent cependant de ces derniers sur les points suivants :

- Les ADL traitent les connecteurs comme des entités de première importance. Les connecteurs explicites permettent de décrire les flots de données et de contrôle entre les composants. Les MIL se limitent à décrire les relations d'utilisation entre modules et supportent seulement un type de connexion.
- Les ADL sont employés lors de l'analyse d'un système alors que les MIL sont employés après qu'un système a été conçu.
- Certains ADL (langages de configuration) supportent le concept d'instantiation et font la différence entre une classe de composant et les instances de celle-ci.
- Certains ADL, tels que OCL, supportent la description de changements dynamiques pouvant avoir lieu pendant l'exécution.

Du fait que les ADL sont traditionnellement utilisés dans une phase de conception de

l'application, ils permettent difficilement de prendre en compte l'évolution des besoins et de l'environnement d'exécution [RS02]. Les ADL peuvent être employés pour générer un programme exécutable, cependant une fois que celui ci a été crée, il n'existe plus un lien continu entre la description architecturale et le programme en exécution.

Un des apports principaux de l'approche orientée objet par rapport à l'approche modulaire est le concept d'instantiation, c'est à dire la création d'objets à partir d'une classe. Une classe est similaire sur certains points avec un module, notamment au niveau de l'encapsulation et du fait qu'elle peut être compilable de façon indépendante ; les différences portent cependant sur les points suivants :

- Une classe peut être instantiée.
- Une classe peut hériter l'état et comportement d'une autre classe.
- Le polymorphisme fait qu'une association entre instances de différentes classes puisse varier selon le contexte d'exécution, alors que les liens entre modules sont statiques.

Une limitation dans cette approche est que souvent l'architecture d'un système n'est pas explicite car elle se trouve imbriquée dans le code des objets, de plus, le polymorphisme complique la possibilité de connaître l'architecture d'un système pendant son exécution [Cer00]. Il existe cependant des travaux comme ArchJava [ACN02] qui tentent de rapprocher les aspects concernant l'architecture et l'implémentation.

Chapitre 3

Approche à composants

Ce chapitre étudie l'approche à composants. Le chapitre présente d'abord les concepts principaux de l'approche puis une description de diverses implémentations de cette approche.

3.1 Concepts principaux

L'approche à composants, bien qu'en vogue aujourd'hui, se caractérise par le fait que ses concepts ne sont souvent pas exprimés de façon claire. Ceci est dû, entre autres, au fait que la description des aspects technologiques est souvent privilégiée par rapport à celle des concepts. Cette section tente de clarifier les concepts généraux de l'approche et des modèles à composants, en restant éloignés d'une technologie particulière.

3.1.1 Introduction

L'approche à composants est relativement récente dans l'histoire du génie logiciel, elle est apparue autour du milieu des années 90. Cette approche promeut la construction d'applications à partir de l'*assemblage* de *composants* [Com03]. De façon simpliste, un composant peut être décrit comme étant une 'brique' logicielle pré-fabriquée qui est conçue pour être composée, c'est à dire assemblée, avec d'autres composants [MN97]. Un composant est *réutilisable*, c'est à dire qu'un même composant peut être employé dans la construction de différentes applications, et sa réutilisation ne nécessite, à priori, pas de connaissances sur l'implémentation. L'approche à composants est fondée sur l'idée que le développement et l'assemblage de composants peuvent être réalisés de façon totalement séparée par des acteurs différents et dans des emplacements différents. Un *modèle à composant* définit les caractéristiques des composants, de leurs assemblages et est accompagné par un support d'exécution. Le modèle à composants permet de réaliser le développement et l'exécution d'applications à base de composants.

L'approche à composants est motivée d'un côté par des arguments économiques, tels que la réduction du temps et du coût de développement, ou bien la spécialisation des acteurs intervenant dans le cycle de vie du développement [BA00b], et d'un autre côté par la proposition de solutions pratiques à des limitations d'approches précédentes, telle que l'approche orientée objet [WD01]. Par rapport à l'approche orientée objet (présentée dans la section 2.4), l'approche à composants promeut notamment l'emploi de la composition au lieu de l'héritage comme mécanisme de réutilisation, le support pour la livraison et le déploiement indépendant des composants ainsi que la prise en compte de l'existence d'aspects non-fonctionnels ; ces concepts sont détaillés dans les sections suivantes.

3.1.2 Définition

Il n'existe pas une définition consensuelle de ce qu'est un composant. Cependant une des définitions les plus souvent citées dans la littérature est celle que donne Szyperski [Szy98] :

> "Un composant logiciel est une unité binaire de composition ayant des interfaces spécifiées de façon contractuelle et possédant uniquement des dépendances de contexte explicites. Un composant peut être déployé de manière indépendante et est sujet à composition par des tierces."

L'intérêt de cette définition est qu'elle contient plusieurs des concepts importants qui caractérisent un composant et qui seront discutés dans ce chapitre. Dans ce travail, cependant, cette définition est employée comme une base qui est complétée en précisant que, dans la pratique, les caractéristiques d'un composant sont réparties dans trois entités que l'on nommera dans cette thèse *classe de composant*, *instance de composant* et le *paquetage de composant*. Les classes de composant sont similaires aux classes de l'approche objet, car elles donnent lieu aux instances de composant. Les classes de composant sont les unités de réutilisation et sont conditionnées dans des paquetages pour permettre de réaliser non seulement le déploiement mais aussi la livraison de manière indépendante (ceci sera expliqué dans la section 3.1.3).

Deux situations se présentent souvent dans la littérature : où bien ces trois entités sont confondues entre elles et sont toutes appelées composant, où bien le mot composant est seulement utilisé pour faire référence à une d'entre elles. Par exemple, dans [BCS02], un composant est défini comme suit :

> "Les composants sont des structures de 'run-time', ce qui veut dire qu'elles se manifestent lors de l'exécution d'un système",

alors que dans [MN97], un composant est défini comme :

> "Une entité généralement statique qui est nécessaire au moment de la construction d'un système et qui n'existe pas forcément au moment de l'exécution :

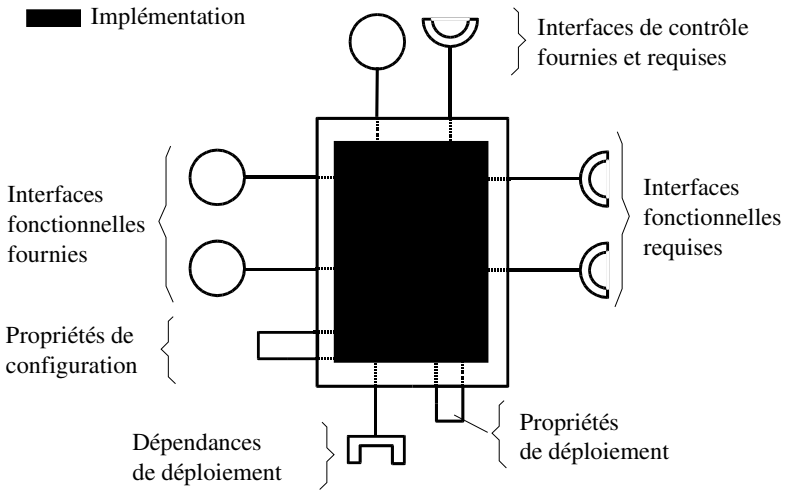


FIG. 3.1 – Représentation schématique d'une classe de composant

par exemple une classe [...]",

finalement dans [Tha99], la définition est la suivante :

"Un composant est un module binaire, tel qu'un fichier DLL ou EXE".

Il faut noter que dans le reste de ce travail, lorsque le mot composant est employé en tant que tel, il fera référence à une classe de composant conditionnée dans un paquetage.

3.1.3 Classe, instance et paquetage de composant

Les trois entités dans lesquelles sont réparties les caractéristiques d'un composant sont décrites à la suite. Il est important de noter que la description de ces caractéristiques se veut générique et qu'un modèle à composants particulier peut présenter des variations quant à la présence de ces caractéristiques.

3.1.3.1 Classe de composant

Une *classe de composant* se situe à un niveau équivalent à celui de la classe dans l'approche orientée objet. Une classe de composant peut être vue à partir d'une *vue externe* qui représente la vision qu'ont les clients des instances du composant, et d'une *vue interne*, qui représente la vue que possède l'environnement d'exécution des instances du composant. Les vues externes et internes sont réalisées par *une implémentation du composant*, qui contient la logique fonctionnelle. La figure 3.1 montre une représentation schéma-

tique des éléments d'une classe de composant et de l'implémentation[1], ces éléments sont les suivants :

Interfaces fonctionnelles et propriétés de configuration : Une interface, constituée par un ensemble de méthodes, est qualifiée comme étant fonctionnelle lorsqu'elle contient des méthodes relatives aux fonctions fournies, ou requises, par les instances du composant. Ces interfaces peuvent de plus supporter une communication de type synchrone ou asynchrone, c'est à dire par évènements. Les interfaces fonctionnelles requises représentent des dépendances au niveau des instances du composant et doivent être satisfaites lorsqu'une instance est créée pour que celle ci puisse être utilisée à travers les interfaces fournies. La déclaration explicite des interfaces fonctionnelles permet à un tiers de réaliser la composition des instances de composant.

Les propriétés de configuration permettent de configurer une instance de composant, par exemple, changer le nom d'un bouton. Les propriétés de configuration peuvent concerner seulement le moment de la création d'une instance et ne pas être accessibles postérieurement, ou bien permettre des changements à tout moment. Ceci est similaire aux valeurs passées dans un constructeur et aux valeurs pouvant être changées à travers des accesseurs (`get`/`set`).

interfaces de contrôle : ces interfaces, fournies ou requises, contiennent des méthodes permettant de gérer le cycle de vie des instances du composant pendant l'exécution (ceci est décrit dans la section 5.4.3). Ces méthodes sont normalement destinées à être appelées par l'environnement d'exécution du modèle à composants, et non pas par les clients des instances. Une interface de contrôle requise représente par exemple une interface qu'utilise une instance pour interagir avec l'environnement d'exécution (aussi appelée *contexte*).

dépendances et propriétés de déploiement : Ces dépendances, qui sont propres à une implémentation de composant, doivent être satisfaites au moment du déploiement d'une classe de composant pour permettre son utilisation. Elles peuvent représenter par exemple une dépendance envers une version particulière de l'environnement d'exécution, des librairies, des ressources binaires (images, sons), des fichiers de configuration ou d'autres classes de composants. Les propriétés de déploiement, qui sont définies au niveau de l'implémentation, sont similaires aux variables de classe dans l'approche objet. Les propriétés d'implémentation sont employées pour configurer des caractéristiques communes à toutes les instances, comme par exemple l'adresse d'un serveur de noms.

[1] Ce schéma ne présente pas la différence entre la vue interne et externe qui est présentée plus loin.

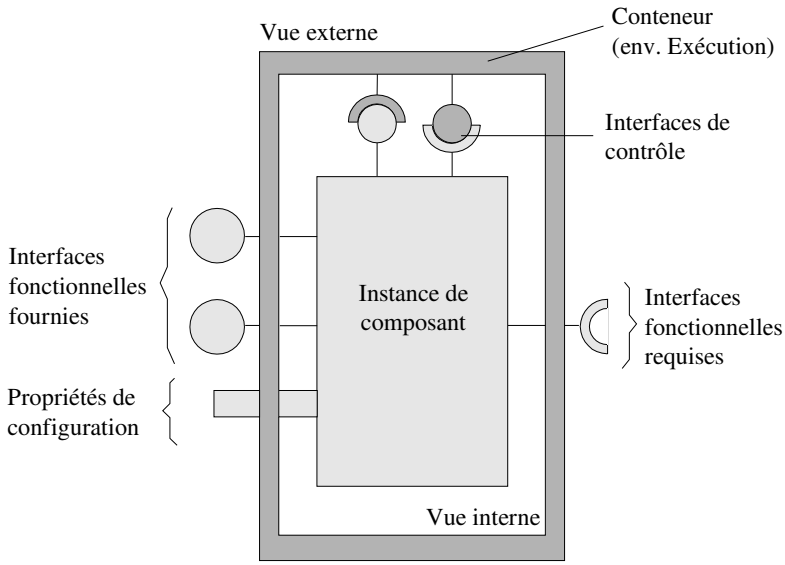


FIG. 3.2 – Instance de composant et conteneur

Selon le modèle à composant considéré, la présence de ces éléments varie. Il faut noter qu'alors que les interfaces fonctionnelles et les propriétés de configuration ont tendance à être présentes sur la vue externe, tandis que les interfaces de contrôle ont tendance à être présentes seulement dans la vue interne. Une vue interne est typiquement associée à une implémentation particulière du composant. La séparation entre vue externe et interne n'est cependant pas présente dans tous les modèles à composants. La différence entre la vue externe et interne est représentée dans la figure 3.2.

Finalement, il est important de noter que lorsqu'il existe une séparation explicite entre la vue externe et son implémentation, il est possible d'avoir différentes implémentations de cette vue externe, et que cette dernière définit alors *un type de composant*. Dans ce cas, la vue externe joue le rôle d'un contrat de structure et de comportement qui doit être rempli par les différentes implémentations ; ces implémentations peuvent éventuellement être substituées.

3.1.3.2 Instance de composant

Une instance de composant est obtenue à partir d'une classe de composant et fournit, en temps d'exécution, les fonctionnalités associées au composant. Une instance se situe à un niveau équivalent à celui des objets du fait qu'elle est identifiée de façon unique par rapport aux autres instances et qu'elle peut avoir un état, leur différence réside surtout dans la différentiation qui est faite entre les interfaces fonctionnelles et celles de contrôle et dans la gestion du cycle de vie.

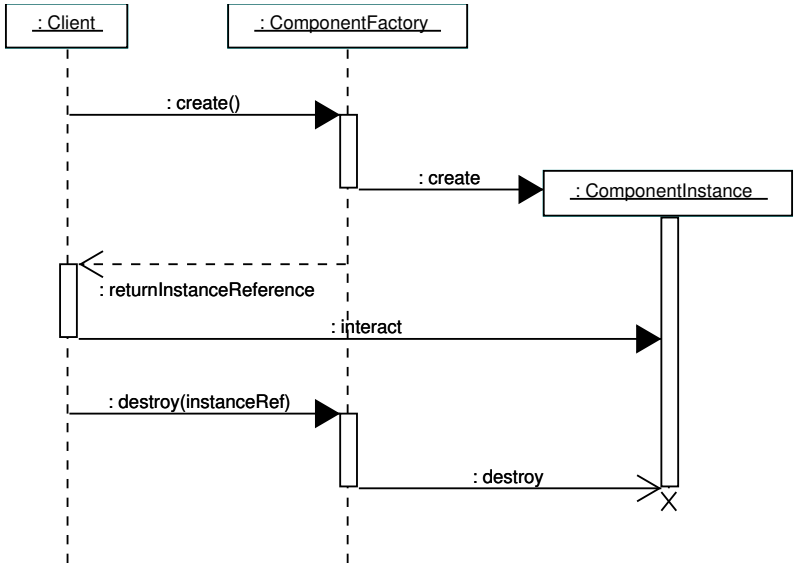


FIG. 3.3 – Création d'une instance de composant

Création et destruction des instances

Lors de l'exécution, les instances de composant sont créées typiquement à partir de *fabriques* [GA95] qui permettent de découpler un client (ici, le client est un acteur qui crée une instance) d'une implémentation particulière du composant. La figure 3.3 montre l'interaction de base qui a lieu au moment où un client demande la création d'une instance de composant. Dans cette interaction, le client contacte une fabrique de composants qui est associée à une classe de composant particulière ; la fabrique est alors chargée de créer une instance de composant qui est retournée au client. Postérieurement, le client peut demander la destruction explicite de l'instance à la fabrique. Le fait que ce soit la fabrique et non le client qui crée réellement l'instance de composant rend possible l'existence de différentes politiques de création des instances, telles que :

instances multiples : dans ce cas (qui est le cas standard), une instance est créée chaque fois qu'un client invoque la méthode de création de la fabrique. C'est l'équivalent du `new` dans un langage orienté objet tel que Java.

instance partagée : dans ce cas une instance unique est créée (patron singleton [GA95]) et elle est retournée à tous les clients chaque fois qu'un client demande la création d'une instance. Dans ce cas l'état de l'instance est partagée par tous les clients.

stock d'instances : dans ce cas une quantité limitée d'instances sont créées ; elles sont retournées aux clients dans la limite de disponibilité du stock. Quand un client demande la destruction d'une instance, celle-ci est retournée au stock pour être utilisée postérieurement avec un autre client. Cette politique est

utile lorsque les ressources, telles que la mémoire sont limitées ou bien lorsque le coût de création d'une instance est élevé.

Dans l'approche à composants un client est normalement au courant de la politique de création des instances associée à une fabrique.

Conteneur

Un avantage de l'utilisation de fabriques pour la création des instances est que la responsabilité de la gestion du cycle de vie des instances peut être retirée des clients, ce qui permet d'avoir des cycles de vie plus complexes que la simple création et destruction de l'approche objet. Lorsqu'une instance est créée, la gestion de son cycle de vie peut être réalisée par une entité nommée *conteneur* qui fait partie de l'environnement d'exécution d'un modèle à composants (voir 4.1.6). Le conteneur, qui est schématisé dans la figure 3.2 contrôle le cycle de vie d'une instance en invoquant les méthodes définies dans les interfaces de contrôle. Les interfaces de contrôle permettent par exemple de rendre une instance persistante, reconfigurable ou bien de suspendre son exécution de façon temporaire.

3.1.3.3 Paquetage de composant

Un *paquetage de composant* est une unité permettant de réaliser la livraison et le déploiement d'une classe de composant de manière *indépendante*. Le mot indépendant fait référence ici au fait que le paquetage contient tout ce qui est nécessaire pour réaliser la création d'instances de la classe de composant, à l'exception de ce qui est déclaré comme étant une dépendance explicite, par exemple vers un environnement d'exécution.

Le fait que l'assemblage des composants peut être réalisé par des acteurs pouvant résider dans une entreprise différente à celle où le composant est développé (voir 3.5) explique qu'un composant soit défini comme étant une unité *binaire*. Le mot binaire, qui est employé ici au sens large, signifie que le code source correspondant à l'implémentation du composant n'est pas forcément livré avec le composant. Le paquetage de composants contient du code binaire correspondant à l'implémentation du composant, ainsi que des ressources telles que des fichiers binaires (librairies, images), ou des fichiers de configuration. De plus, un paquetage de composant peut contenir de l'information facilitant l'assemblage et le déploiement, comme par exemple une description des différents éléments présents dans les vues externe et interne.

3.1.4 Composition (ou Assemblage)

La *composition*, ou assemblage, de composants est, en plus du développement de composants, l'autre activité fondamentale de l'approche à composants (voir 3.1.5). En

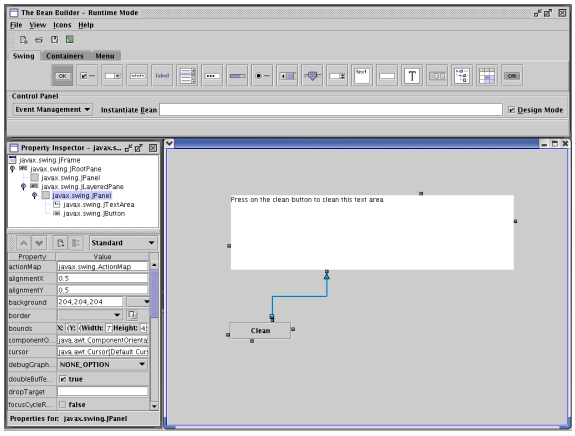


FIG. 3.4 – Environnement pour la composition visuelle

dépit de son importance, moins d'effort a été dédié jusqu'ici dans la recherche de techniques permettant de réaliser la composition que dans le développement de modèles à composants particuliers [LSSG02]. De plus, le fait que le concept de composant ne soit pas clairement défini a pour conséquence de rendre vague le concept de composition.

Dans ce travail, la composition de composants fait référence spécifiquement à la *composition des instances de composant* à partir des éléments présents sur leur vue externe. La composition est réalisée comme une description qui résulte dans la création, configuration et de connexion d'instances de composant ; la composition contient donc l'information concernant l'architecture d'une partie ou de l'ensemble d'une application. Le fait que l'information architecturale soit en dehors du code des composants facilite leur réutilisation dans la construction de divers systèmes. Au moment de l'exécution, l'information contenue dans la composition est utilisée pour créer, configurer et connecter un ensemble d'instances de composant. Finalement, il faut noter que la connexion d'instances peut nécessiter de l'écriture de code d'adaptation ("glue") permettant de réaliser la connexion entre des instances ayant par exemple des interfaces incompatibles [AN00].

3.1.4.1 Types de compositions

La composition peut être réalisée de plusieurs manières :

composition visuelle : ce type de composition est réalisé dans un environnement qui permet d'assembler des instances de composants de façon interactive. La composition visuelle est réalisée en connectant des représentations graphiques d'instances (par exemple des icônes) ou bien, des instances réelles. Les connections entre les instances sont réalisées en 'tirant des traits' entre elles, et des dialogues permettent de changer les valeurs des propriétés de configuration des instances. Une fois que l'assemblage visuel est terminé, l'information de composition doit être sauvegardée afin de recréer la composition ultérieure-

ment. Un exemple d'environnement orienté à la composition visuelle est le BeanBuilder de Sun [Dav02], qui est montré dans la figure 3.4. Cet environnement contient un catalogue de classes de composants (en haut), une aire d'assemblage (en bas à droite) ainsi qu'un dialogue de configuration des instances (en bas à gauche) et un arbre montrant la hiérarchie des instances (à gauche).

composition déclarative : la composition déclarative est basée sur l'utilisation de langages décrivant des relations entre des concepts propres à la composition, tels que 'composant' ou 'connecteur'. Bien que ces langages soient proches des langages de description d'architecture ADL [Cle98], ils se différentient de ces derniers par le fait que le résultat de la composition est toujours exécutable. Des exemples de langages déclaratifs de composition sont décrits dans [Bir01], et [WD01].

composition impérative par langage de programmation : dans ce type de composition, un langage de programmation 'classique', tel que Java, est employé pour réaliser la composition des composants. Les langages de programmation classique sont normalement typés statiquement et compilés. Le fait d'utiliser le même langage pour réaliser l'assemblage et le développement rend difficile la spécialisation des acteurs dédiés à ces activités, car la réalisation de ces tâches implique que les différents acteurs ont des connaissances de même niveau.

composition impérative par langage de script : dans ce type de composition, un langage pouvant contenir des concepts spécifiques à la composition, et donc de plus haut niveau d'abstraction, est utilisé. Les langages de script, tels que CorbaScript [MGG97], se différencient des langages de programmation classique par le fait qu'ils ont tendance à être non-typés et interprétés mais aussi moins performants [Ous98]. L'avantage de l'utilisation de ce type de langages pour réaliser la composition est que l'assembleur ne nécessite pas d'avoir les mêmes connaissances que les développeurs, surtout si le langage d'assemblage est plus simple que celui de programmation.

La composition visuelle et déclarative ont tendance à donner lieu à des architectures statiques alors que la composition impérative permet de réaliser des changements dans l'architecture pendant l'exécution, comme par exemple la création et destruction d'instances et de connexions entre celles ci.

3.1.4.2 Composition hiérarchique

La composition hiérarchique fait référence au fait d'utiliser une composition comme une implémentation de composant, et d'utiliser ensuite des instances de cette classe de

composant à l'intérieur d'autres compositions. L'intérêt de réaliser une composition hiérarchique est que la complexité d'une architecture peut être mieux gérée grâce au groupement d'instances réalisant des tâches communes, et le fait qu'une composition qui devient une classe de composant puisse, comme tout autre composant, être réutilisée dans divers contextes. La composition hiérarchique nécessite de réaliser la connexion d'éléments présents dans les vues externe et interne de la classe de composant avec les éléments de la composition [CFD02].

En dépit du fait que la composition hiérarchique soit un des concepts principaux des langages de description d'architecture (par exemple dans ACME [GMW00]), ce concept n'est pas courant dans les modèles à composants. L'explication de ceci peut résider dans le fait que la réalisation de la composition hiérarchique présente plusieurs problèmes pratiques. Un premier problème est que lorsqu'une composition devient une implémentation de composant, l'information architecturale qu'elle contient devient normalement inaccessible aux assembleurs du composant résultant. Dans ce contexte, connaître l'architecture globale d'un système correspond à violer l'encapsulation. Un autre problème lié à la composition hiérarchique est lié au comportement, car il est couramment impossible de garantir qu'une composition qui implémente une vue externe implémente non seulement la structure mais aussi le comportement défini par la vue externe. Finalement, d'autres problèmes apparaissent au niveau du traitement des propriétés non-fonctionnelles (voir 3.1.6.1) dans une hiérarchie, et au déploiement, car il est nécessaire de connaître les instances qui sont utilisées dans une composition pour déployer les classes de composant correspondantes. Dans l'actualité, aucun modèle à composants industriel (à l'exception de l'ADL Fractal, voir 3.2.5) ne fournit des moyens de réaliser des compositions hiérarchiques de façon déclarative. La composition hiérarchique peut cependant être achevée à travers la programmation, par exemple en utilisant la délégation, mais toute sa gestion reste à la charge du programmeur.

3.1.5 Cycle de vie de développement

La construction d'applications à base de composants, qui est schématisée dans la figure 3.5, se décompose en trois étapes dont deux sont fondamentales : le développement de composants et la composition (ou assemblage) ; la troisième étape concerne les activités qui ont lieu après le développement et l'assemblage, et incluent notamment le déploiement des applications et de leurs composants [MM01] ainsi que l'exécution et l'administration des applications. Bien que les étapes fondamentales de l'approche à composants concernent le développement de composants ou de compositions, cette approche ne propose pas de moyens d'analyse permettant de modéliser un problème d'une façon particulière comme dans l'approche orientée objet (voir 2.4.2) ; cette approche est plutôt focalisée dans les activités qui ont lieu après l'analyse. Chacune des trois étapes est

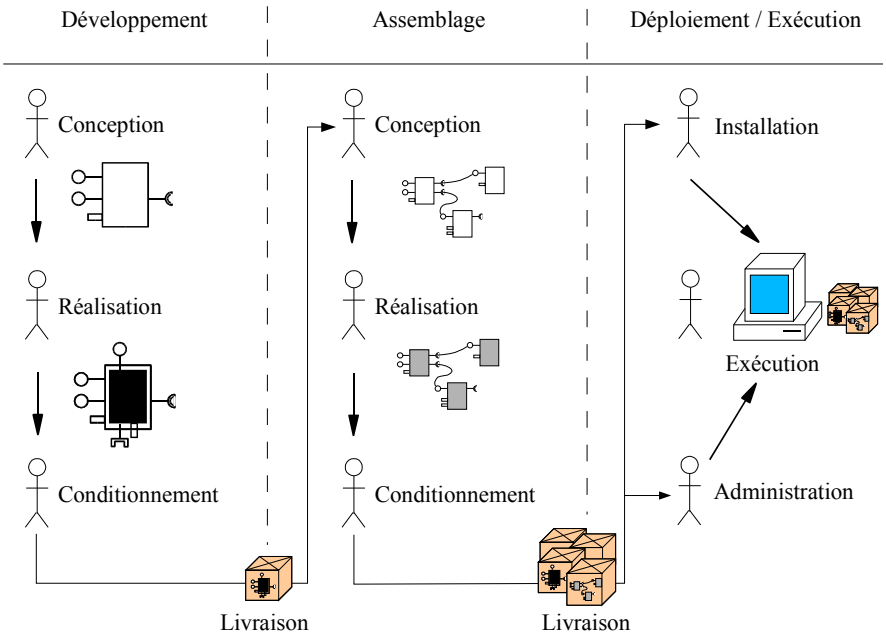


FIG. 3.5 – Cycle de vie de développement simplifié

constituée à son tour par un certain nombre d'activités pouvant être effectuées par des acteurs avec des compétences différentes, et ces trois étapes sont séparées entre elles par des activités de livraison.

3.1.5.1 Développement

Le développement de composants inclut les activités de conception, implémentation et conditionnement des composants destinés à être assemblés dans différentes applications.

conception : la conception de composants inclut la spécification de la *vue externe* d'une classe de composant ainsi que de son comportement.

réalisation : la réalisation de composants consiste à la création de l'implémentation de la classe de composant qui implémente les vues externe et interne. Plusieurs implémentations peuvent être réalisées pour une même vue externe.

conditionnement : lors du conditionnement, une classe de composant est introduite dans un paquetage de composant qui permet de réaliser la livraison et le déploiement indépendants.

3.1.5.2 Assemblage

L'assemblage (ou composition) de composants inclut les activités de conception, de réalisation et de conditionnement d'un assemblage qui peut représenter une partie ou bien

la totalité d'une application. L'étape d'assemblage suppose l'existence de composants développés auparavant qui sont utilisés par les acteurs de cette étape.

conception : La conception d'un assemblage est réalisée en étudiant la manière de composer un ensemble de composants pré-existants pour créer une composition représentant soit une partie ou bien la totalité d'une application, c'est à dire une architecture.

réalisation : La réalisation d'un assemblage consiste à l'écriture du code permettant de réaliser la composition des instances de composants.

conditionnement : Lors du conditionnement, un assemblage est introduit dans un *paquetage d'assemblage* qui peut inclure les composants employés dans l'assemblage ainsi que des ressources propres à l'assemblage telles que des fichiers de configuration ou des fichiers binaires (librairies, images, sons, etc...).

3.1.5.3 Déploiement, Exécution et Administration

Les étapes postérieures au développement et à l'assemblage sont les suivantes :

déploiement : lors de l'installation, des paquetages contenant des compositions et des composants sont installés et des activités de configuration peuvent être réalisées.

exécution : l'application est exécutée. A ce moment là, l'environnement d'exécution est actif.

administration : l'administration inclut l'application de mises à jour ainsi que la désinstallation d'une application et de ses composants.

3.1.6 Environnement d'exécution

Un modèle à composants, qui définit la structure des composants et des compositions, est associé à un environnement d'exécution qui fournit du support aux applications construites à partir du modèle lors de l'exécution. L'environnement d'exécution est parfois comparé à un mini-système d'exploitation [BA00a] car il est chargé de gérer des aspects divers tels que le cycle de vie des instances ou bien les propriétés non-fonctionnelles. L'environnement d'exécution est typiquement constitué par le conteneur (voir 5.4.3) ainsi qu'une infrastructure sous-jacente (par exemple un intergiciel, ou une machine virtuelle). L'environnement d'exécution peut aussi fournir des mécanismes d'introspection permettant de réaliser des analyses de la structure des composants ou même de l'application lors de l'exécution.

3.1.6.1 Gestion des propriétés non-fonctionnelles

Une implémentation de composant peut devoir réaliser, en plus des interfaces fonctionnelles, la gestion de propriétés dites *non-fonctionnelles* car elles ne sont pas directement liées aux fonctionnalités offertes par le composant. Un exemple de propriété non-fonctionnelle est la distribution, c'est à dire la possibilité d'une instance d'être invoquée de façon distante. Pour supporter la distribution, l'implémentation du composant doit contenir, en plus du code implémentant les interfaces fonctionnelles, du code destiné à gérer la communication à distance. Lorsque plusieurs propriétés non-fonctionnelles sont gérées, le code correspondant à la gestion des propriétés non fonctionnelles peut devenir beaucoup plus important que celui qui implémente les fonctionnalités mêmes du composant. De plus, les deux variétés de code peuvent se trouver mélangées dans l'implémentation du composant car elles peuvent être inter-dépendantes. Ce mélange est problématique car il implique que les développeurs des fonctionnalités d'un composant doivent écrire le code chargé de gérer les propriétés non-fonctionnelles, qui est souvent complexe et redondant, et non seulement se focaliser dans les aspects applicatifs.

Pour résoudre cette problématique, divers modèles à composants proposent des moyens permettant d'extraire le code de gestion des propriétés non-fonctionnelles de l'intérieur des composants. Le code de gestion des propriétés non-fonctionnelles est contenu dans l'environnement d'exécution et est configuré à partir d'informations définies dans les composants. Cette séparation est un des points importants de l'approche, car permet aux développeurs de se concentrer dans l'écriture de la logique applicative.

Dans la pratique, ceci est réalisé à travers l'utilisation d'intermédiaires entre les clients et les instances de composant. L'intermédiaire, qui peut être le conteneur (comme dans le modèle EJB), reçoit les appels de la part du client de l'instance pour ensuite les transmettre à l'instance. L'intermédiaire peut intervenir avant, durant ou après un appel pour gérer les propriétés non-fonctionnelles telles que la distribution, la sécurité ou les transactions [CPFD01]. Ceci explique que dans le schéma de la figure 3.2, le conteneur soit représenté comme une entité qui entoure une instance de composant. Du fait que les propriétés non-fonctionnelles sont gérées de façon identique pour toutes les instances d'une classe de composant, les propriétés de déploiement peuvent être employées pour configurer la gestion des propriétés non-fonctionnelles[2].

3.1.6.2 Introspection

L'introspection permet de réaliser l'analyse, pendant l'exécution, de la structure d'une instance de composant (typiquement sa vue externe) et éventuellement de l'architecture

[2]Les valeurs de ces attributs peuvent éventuellement être définies jusqu'au moment de la configuration dans l'étape de déploiement.

même du système. L'introspection nécessite que l'environnement d'exécution puisse disposer d'informations concernant la structure des composants et du système [Mar01]. L'introspection au niveau de la structure de composants est relativement peu coûteuse, cependant l'introspection au niveau de l'architecture d'un système complet peut être coûteuse du fait qu'elle peut nécessiter une très grande quantité d'informations correspondant à chaque connexion ainsi qu'à chaque instance présentes à un moment donné dans le système.

3.1.7 Synthèse

L'approche à composants introduit une méthodologie visant à faciliter la construction d'applications à partir de l'assemblage de briques logicielles pré-fabriquées appelées composants. Cette méthodologie insiste particulièrement sur une séparation entre l'étape de développement et celle d'assemblage des composants. L'assemblage est réalisé à partir de composants qui peuvent être obtenus ailleurs que dans le lieu où ils sont assemblés. Un modèle à composants définit la structure des composants et de leurs assemblages et permet de réaliser le développement suivant le cycle de développement propre à cette approche. Un modèle à composants est accompagné par un environnement d'exécution qui fournit du support aux applications pendant l'exécution.

3.2 Étude de divers modèles à composants

L'étude présentée dans la section précédente permet de définir une liste de caractéristiques utile pour comparer divers modèles à composants.

classe de composant : caractérise les classes de composant du modèle, notamment les éléments présents dans les vues externe et interne ainsi que leur implémentation.

instance de composant : caractérise les instances, notamment les politiques de création ainsi que les cycle de vie supportés.

paquetage : caractérise le paquetage employé pour le conditionnement.

composition : caractérise la façon suivant laquelle la composition est réalisée.

environnement d'exécution : caractérise l'environnement d'exécution, notamment au niveau du support des aspects non fonctionnels.

Cette section présente une étude de divers modèles à composants suivant un ordre historique de leur apparition. Chaque modèle est présenté en décrivant son domaine d'application et ses caractéristiques. Un exemple permettant de voir la manière suivant laquelle les concepts sont réalisés au niveau du code est aussi présenté.

3.2.1 COM

Le modèle COM (Component Object Model) de Microsoft [Box98] a été conçu pour résoudre le problème de l'interopérabilité au niveau binaire entre des composants appartenant à des applications écrites par différents vendeurs. Un exemple typique d'interopérabilité est l'incorporation de fonctionnalités fournies par une application dans une autre ; par exemple le composant moteur HTML d'un navigateur peut être incorporé dans un reproducteur de médias qui nécessite d'afficher des pages web, même si ces deux composants sont implémentés dans des langages différents.

Bien que COM impose une séparation entre les interfaces fonctionnelles et leur implémentation, il ne propose pas de moyen de décrire la vue externe d'un composant, du fait que le nombre d'interfaces fonctionnelles peut varier. Un client d'une instance de composant doit toujours vérifier la présence d'une interface qu'il compte utiliser. Une interface dans COM contient un ensemble de méthodes, est décrite dans un langage de description spécifique (IDL) et est identifiée de façon unique. Une interface est considérée comme étant un contrat de structure et de comportement statique (une fois publiée, l'interface ne peut plus être changée). Pour permettre l'évolution des interfaces, un composant COM peut implémenter simultanément plusieurs versions d'une même interface. Toutes les interfaces de COM héritent d'une interface racine appelée `IUnknown` qui contient des méthodes permettant aux clients d'introspecter une instance de composant pendant l'exécution. Les méthodes fournies par `IUnknown` sont employées par un client pour deux raisons : pour tester la présence d'une interface particulière ou bien pour voir si le client implémente une nouvelle version d'une interface.

COM est un modèle à composant qui souffre de problèmes de complexité au niveau de l'implémentation mais qui cependant a été novateur à son époque et reste probablement aujourd'hui le modèle à composants le plus répandu. COM a inspiré d'autres modèles qui ont repris ses idées fondatrices (par exemple l'Object Modeler de Dassault Systèmes [San01] ou Bonobo de Gnome [SLB00]). Même si .Net [Pla02] vise à remplacer COM, les idées de ce dernier sont incorporées dans la nouvelle plateforme de Microsoft.

Caractéristiques

classe de composant : dans COM, l'implémentation d'un composant est appelé une *classe COM* et est identifiée de façon unique. Dans COM, il n'existe pas réellement de concept de vue externe en tant que groupement de plusieurs interfaces ; la seule contrainte imposée à un composant est de fournir l'interface `IUnknown` à partir de laquelle toute autre interface implémentée par le composant peut être découverte. De plus, dans COM vue externe et interne sont confondues car les interfaces de contrôle sont accessibles à partir de `IUnknown`. Une classe COM implémente un certain nombre d'interfaces

a) Interfaces fonctionnelles

```
[ object, uuid(b26c1237-11ed-24d0-9a4b-01d975aab51e), ]
interface IClient : IUnknown {
  HRESULT receive([in] string Data);
};
[ object, uuid(a03d1420-b1ec-11d0-8c3a-00c04fc31d2f), ]
interface IServer : IUnknown {
  HRESULT addClient([in] Client *client);
  HRESULT GetClients([out] int clients);
};
```

b) Instantiation

```
IServer *pServer = NULL;
HRESULT hr = CoCreateInstance (
                   CLSID_ServerImpl, // identifie implementation
                   NULL,
                   CLSCTX_SERVER,
                   IID_IServer,
                   reinterpret_cast <void**> (&pServer));
if( !SUCCEEDED(hr))
{
// L'objet COM n'a pas pu être créé...
}
```

c) Introspection

```
IServerConfig *psrvCfg = NULL;
hr = pServer -> QueryInterface(IID_IServerConfig,
                     reinterpret_cast<void **>(&psrvCfg));
if(SUCCEEDED(hr))
{
  // Cette instance implémente IServerConfig
  psrvCfg->setMaxClients(1);
}
```

d) Composition

```
IClient *pClient = NULL;
hr = CoCreateInstance (...);
if(SUCCEEDED(hr))  // Le client à pu être créé
{
  pServer->addClient(pClient); // connexion des instances
}
```

FIG. 3.6 – Exemple COM

fonctionnelles fournies mais ne définit pas quelles interfaces sont requises.

instance de composant : dans COM une instance de composant est appelée un *objet COM* (un objet COM peut en réalité être réalisé par plusieurs objets). Les instances de composant sont créées à partir de fabriques (méthode `CoCreateInstance` dans la figure 3.6.b), et celles ci peuvent implémenter les diverses politiques de création des instances décrites auparavant. Chaque fois qu'un client obtient une interface d'un composant, il doit faire appel à une méthode de `IUnknown` permettant d'avertir l'instance qu'elle est employée (en incrémentant un compteur) ; quand le client termine son interaction il appelle une méthode qui décrémente le compteur. Une instance peut être détruite quand elle n'est plus utilisée par ses clients (compteur à 0), cependant cette technique est une des sources principales de problèmes du modèle.

paquetage de composant : les classes de composant sont déployées dans des librairies dynamiques (DLL) ou dans des applications (EXE). Ces paquetages ne peuvent cependant contenir que du code correspondant à l'implémentation des composants et pas d'autres types de ressources.

composition : du fait que les classes de composant COM ne décrivent pas quelles sont les interfaces requises par les instances de composant, il est difficile de réaliser une composition de façon explicite. Les dépendances entre instances de composant ne sont pas créées par un tierce mais sont décrites via des instructions de création enfouies à l'intérieur des instances mêmes. Par ailleurs, la composition peut être réalisée de façon impérative notamment à travers des langages de script comme Visual Basic Script.

environnement d'exécution : l'environnement d'exécution du modèle COM fait partie du système d'exploitation Windows ; et les classes de composant sont enregistrées dans le *registry* de Windows. L'environnement d'exécution de COM se charge de rendre transparente la communication entre les instances de composants qui peuvent s'exécuter sur des processus différents. Des extensions postérieures à COM ont rajouté en plus la possibilité de communication entre composants localisés sur des machines différentes (DCOM) puis la gestion d'aspects non fonctionnels comme la transaction avec une approche à container (COM+).

3.2.2 JavaBeans

En 1997, Sun crée la spécification du modèle à composants appelé JavaBeans [Sun97]. L'objectif principal de ce modèle est de simplifier la construction d'applications à travers la composition de façon visuelle. Le type d'applications visé par ce modèle concerne surtout des applications non-distribuées supportant l'interaction à travers une interface

utilisateur. Le modèle à composants JavaBeans introduit un aspect original permettant de faciliter l'assemblage visuel : un composant peut être livré avec un ensemble de classes destinées à être employées par l'environnement d'assemblage pour faciliter la configuration des instances du composant. Ces classes peuvent cependant être retirées du paquetage lors de la livraison définitive des composants. Dans le cas où un composant n'est pas accompagné de classes de configuration, l'environnement d'assemblage doit être capable de générer de façon autonome des dialogues permettant de configurer les instances à travers l'introspection.

Caractéristiques

classe de composant : dans JavaBeans, une classe de composant est une classe qui est écrite suivant un ensemble de patrons de nommage qui permettent de définir des éléments de la classe du composant. Un composant JavaBeans peut avoir plusieurs interfaces fonctionnelles fournies et requises, avoir un ensemble de propriétés de configuration, et envoyer ou recevoir des évènements. Dans JavaBeans il n'existe cependant pas de concept de vue externe explicite, et le modèle à composants JavaBeans n'oblige pas à réaliser une séparation explicite entre les éléments de la vue externe et l'implémentation du composant. Le concept de vue interne n'existe pas du fait qu'il n'y a pas d'interfaces de contrôle. Des dépendances et propriétés de déploiement peuvent cependant être définies au niveau de la classe. Un problème qui se pose avec ce modèle est que la séparation entre les propriétés de configuration et les interfaces requises n'est pas claire (toutes les deux sont représentées par des méthodes `set`/`get`, `add`/`remove`), l'information permettant de les différentier est contenue dans les classes de configuration et n'est pas explicite.

instance de composant : une instance de composant est créée directement à partir d'une classe de composant. Du fait qu'une séparation entre interfaces et implémentation n'est pas obligatoire, le concept de fabrique ne fait pas partie de la spécification des JavaBeans (bien que le framework fournisse une méthode permettant de créer des instances).

paquetage de composant : un JavaBean est conditionné dans un fichier JAR qui contient le code du composant ou bien une instance prototype. Le composant peut être livré avec des classes destinées à être employées par un environnement d'assemblage et qui permettent de connaître les éléments de la classe de composant et de personnaliser une instance en changeant ses propriétés. Le paquetage permet de décrire des dépendances de déploiement, mais seulement à l'intérieur du paquetage (à travers la balise `Depends-On` dans le manifeste), cependant ceci est optionnel.

a) Classe de composant

```
// Vue externe et implémentation
package org.examples;
interface ServerConfig
{
  public int getMaxClients();
  public void setMaxClients(int max);
  public void addClient(Client c);
}
public class Server implements ServerConfig
{
  private int maxClients = 2;
  private List clients = new ArrayList();
  Server()
  {
  }
  public int getMaxClients() // propriété de configuration
  {
    return maxClients();
  }
  public void setMaxClients(int max)
  {
    maxClients = max;
  }
  public void addClient(Client c) // interface requise
  {
    clients.add(c);
  }
};
```

b) Instantiation

```
Server server = (Server)
  Beans.instantiate(this.getClassLoader(),"org.examples.Server");
```

c) Introspection

// Exemple d'introspection en temps d'exécution

```
if(Beans.isInstanceOf(server,ServerConfig.class))
{
   ServerConfig cfg =
       Beans.getInstanceOf(server,ServerConfig.class);
   cfg.setMaxClients(2);
}
```

d) Composition

// Exemple de composition impérative

```
Client c1 = new Client();   // Création d'un client
c1.setName("Client1");      // Configuration
server.addClient(c1);       // Connexion des instances
```

FIG. 3.7 – Exemple JavaBeans

composition : dans JavaBeans, la composition est principalement réalisée de façon visuelle dans un environnement dédié (voir figure 3.4). Lors de l'assemblage, des instances sont créées à partir d'un catalogue de classes de composants. Les propriétés de chaque instance sont alors configurées et les instances sont connectées entre elles. Une fois que la composition est terminée, elle peut être sauvegardée dans un fichier qui est ensuite employé pour reconstruire l'assemblage [MW99]. Le modèle à composants JavaBeans ne spécifie pas la manière de conditionner un assemblage. La composition peut aussi être réalisée de façon impérative, comme le montre l'exemple dans la figure 3.7.d.

environnement d'exécution : l'environnement d'exécution du modèle à composants JavaBeans se trouve dans la machine virtuelle de Java et est représenté par un ensemble de classes du package `java.beans`, dont notamment la classe `Beans` qui fournit des méthodes permettant de réaliser la création d'instances de composants. L'introspection peut être réalisée avant l'instantiation à travers la classe `Introspector` et une instance peut être introspectée à travers les mécanismes standard de Java. Une extension postérieure à la spécification du modèle a introduit le concept de contexte d'exécution [Cab98], ce concept permet de regrouper des instances dans un contexte et des hiérarchies de contextes peuvent être créées. Un contexte fournit de plus des services aux instances qu'il contient (ceci sera présenté dans la section 4.2.2).

3.2.3 EJB

Le modèle à composants EJB de Sun [Sun01] se place dans un contexte d'applications construites selon une architecture répartie en trois tiers [RAJ02] : un tiers de présentation, qui réside principalement dans une machine du côté de l'utilisateur, un tiers applicatif, qui réside dans un serveur, et un tiers de données, correspondant par exemple à une base de données. L'idée de cette architecture est de permettre l'interaction des utilisateurs avec les données en passant par le tiers applicatif[3]. L'interaction des utilisateurs avec les instances du tiers applicatif est souvent réalisée à partir d'un navigateur, et donc cette interaction est de type sessionnel. Ces applications trois tiers nécessitent d'un support de plusieurs caractéristiques non fonctionnelles telles que la transaction, la sécurité, la distribution et la persistance. Le modèle EJB est spécifiquement orienté à la construction du tiers applicatif.

Il est intéressant de noter que bien que l'architecture trois tiers est de nature répartie, le modèle EJB ne suppose pas que les applications du tiers du milieu soient distribuées, bien que cela reste réalisable.

[3]Le tiers applicatif est souvent appelé *tiers milieu*.

Caractéristiques

classe de composant : La vue externe des classes de composant EJB est constituée d'une unique interface fonctionnelle fournie appelée l'interface *remote* (voir figure 3.8.a). EJB ne permet pas de décrire de façon explicite les interfaces requises par les instances du composant ou les propriétés de configuration (elles peuvent être réalisées comme dans JavaBeans à partir de conventions de nommage). L'implémentation du composant est réalisée par une classe Java qui doit implémenter en plus des méthodes de l'interface remote, un ensemble de méthodes de contrôle qui font partie de la vue interne du composant. Les méthodes de contrôle font partie de classes de base spécifiques qui varient selon la catégorie de l'instance (décrites à la suite).

instance de composant : Les instances de composants EJB sont divisés en trois catégories : *session*, *entité* et *message*. Les instances de composants session sont créées lors de l'interaction avec les utilisateurs et leur durée de vie correspond à la durée de l'interaction. Une instance de composant session peut avoir un état ("stateful") ou non ("stateless") et ceci est déterminé par le type d'interaction avec l'utilisateur ; si l'interaction est sessionnelle (un état est maintenu à travers différents appels de méthode), un état est nécessaire alors que si l'interaction est ponctuelle (requête/réponse), l'état n'est pas nécessaire. Les instances de composants entité représentent des données résidant dans la base de données et sont persistantes. Les instances de composant message sont proches des composants session mais supportent une communication de type asynchrone. Les instances de composant sont créées à partir d'une fabrique appelée un *home* qui permet soit de créer une nouvelle instance, soit de retrouver une instance persistante à partir d'une clé qui l'identifie. Il est important de noter que les clients du tiers présentation n'interagissent pas de façon directe avec les composants de type entité, mais seulement à travers les composants de type session.

paquetage de composant : L'unité de livraison des composants EJB (fichier de type JAR) contient en plus du code de la classe de composant, un fichier appelé *descripteur de déploiement* dans lequel sont déclarées un ensemble de propriétés de déploiement (qui incluent des propriétés non-fonctionnelles) ainsi que des dépendances de déploiement. Au moment du déploiement d'un composant EJB, les valeurs des propriétés et les dépendances de déploiement peuvent être modifiées pour configurer la classe de composant en fonction de l'environnement où elle est déployée.

composition : Développer une application à partir de composants EJB de base consiste à créer de nouveaux composants responsables d'instantier et connecter des

a) Classe de composant

```
// Interface fonctionnelle (vue externe)
package org.ejbexamples;
interface EJBServer extends javax.ejb.EJBObject
{
  public int getMaxClients() throws RemoteException;
  public void setMaxClients(int max) throws RemoteException;
  public void addClient(Client c) throws RemoteException;
  public void startServing() throws RemoteException;
}
// Implémentation de composant
public class EJBServerImpl implements javax.ejb.SessionBean
{
  private SessionContext ctxt;
  private int maxClients = 2;
  private List clients = new ArrayList();

  // Méthodes de contrôle du JavaBean (vue interne)
  public void ejbCreate() {...};
  public void ejbRemove() {...};
  public void ejbActivate() {...};
  public void ejbDeactivate() {...};
  public void setSessionContext(SessionContext ctxt)
  { this.ctxt = ctxt; }
  // Méthodes de l'interface fonctionnelle (vue externe)
  public int getMaxClients()
  {
    return maxClients;
  }
  public void setMaxClients(int max)
  {
    maxClients = max;
  }
  public void addClient(Client c)
  {
    clients.add(c);
  }
};
```

b) Instantiation

```
Context ctx = new InitialContext(System.getProperties());
// Obtenir référence vers le home
Object obj = ctx.lookup("ServerHome");
ServerHome srvHome = (ServerHome)
      PortableRemoteObject.narrow(obj,ServerHome.class); // cast
Server server = (Server) srvHome.create();
....
server.remove();  // destruction de l'instance
```

c) Composition

```
server.addClient(client);       // Connexion des instances
```

FIG. 3.8 – Exemple EJB

composants de base. La composition est normalement impérative à travers le langage Java. Les composants dont l'implémentation réalise la composition sont typiquement des composants de type session avec état.

environnement d'exécution : Le modèle EJB supporte un nombre limité et non extensible de caractéristiques non fonctionnelles mentionnées auparavant, ces caractéristiques sont supportées à travers une approche à conteneur. Les conteneurs sont installés dans des *serveurs* qui font partie de l'environnement d'exécution du modèle EJB.

3.2.4 CCM

Le modèle à composants de CORBA (CCM) [Obj99] ajoute une couche au dessus de l'intergiciel CORBA [Vin97] permettant de définir l'architecture d'une application distribuée sous forme de composition d'instances de composant. La description d'une composition, appelée *Assembly*, est réalisée de façon déclarative et permet de guider le déploiement, sur plusieurs emplacements, des classes de composant ainsi que la création, configuration, connexion et activation des instances de ces derniers.

Une autre caractéristique de ce modèle est qu'il supporte l'implémentation de composants dans différents langages ; ceci est facilité par l'utilisation d'un langage abstrait pour la description d'interfaces (IDL), qui est employé pour décrire la vue externe des classes de composants, d'un langage appelé CIDL qui permet de décrire les implémentations et de l'infrastructure de communication commune proposée par l'intergiciel. Bien que le support de multiples langages d'implémentation soit un avantage, cela oblige à tout réaliser de façon abstraite puis de passer par des étapes de compilation qui traduisent les descriptions abstraites vers des langages concrets.

EJB et CCM sont destinés à un domaine d'application commun ; d'ailleurs EJB est décrit dans [MM01] comme un sous ensemble de CCM. A différence des EJB, les application écrites avec CCM peuvent être distribuées sur plusieurs sites.

Caractéristiques

classe de composant : La vue externe d'une classe de composant est constituée d'un ensemble de *ports* dont il existe quatre variétés différentes : les *facettes* et les *réceptacles* qui sont des interfaces fonctionnelles synchrones fournies et requises, et les *sources* et *puits d'évènements* qui sont des interfaces fonctionnelles asynchrones fournies et requises ; la vue externe peut aussi contenir des propriétés de configuration. La vue externe d'une classe de composant peut contenir plusieurs instances d'un même port, les ports étant nommés. De plus, les interfaces ont une cardinalité associée qui peut être `simple`

a) Classe de composant

```
// Vue externe
interface Server {...};
interface ServerConfig {...};
interface Client {...};
component MyServer
{
  attribute int MaxClients;
  provides Server server;
  provides ServerConfig serverCfg;
  uses multiple Client clients;
}
//implémentation
composition process MyServerImpl
{
  home executor MyServerHomeImpl
  {
    implements MyServerHome;
    manages MyServerImpl;
  };
};
```

b) Balises de composition

```
<componentassembly>
  <description>
  <componentfiles>
  <partitionning>
  <connection>
    <connectinterface>
      <usesport>
      <providesport>
    </connectinterface>
    <connectevent>
      <consumesport>
      <publishessport>
    </connectevent>
  </connection>
</componentassembly>
```

FIG. 3.9 – Exemple CCM

ou `multiple` et qui contraint le nombre de connexions qui peuvent être réalisées vers une instance du composant. La vue externe est décrite dans le langage IDL (voir figure 3.9.a), et elle est implémentée dans un langage appelé CIDL. L'implémentation définit aussi la vue interne de la classe de composant qui implémente des méthodes de contrôle.

instance de composant : Comme dans EJB, il existe différentes variétés d'instances de composant, catégorisées en fonction de leur cycle de vie : la variété *service* représente une instance sans état pouvant être partagée par de multiples clients ; *session* représente une instance avec état dont la durée de vie s'étend sur plusieurs appels de méthodes ; *process* représente une instance avec état persistant mais non identifiée de façon unique et *entity* pour une instance avec état persistant et identifiée avec une clé unique. Les variétés service et session sont en relation avec la logique applicative tandis que la variété process est spécifique à la modélisation de processus, finalement, la variété entity représente une entrée dans une base de données. Les instances de composants sont créées à travers des fabriques appelées des *homes* et, une fois créées, leur cycle de vie est géré par un conteneur ; dans CCM, les homes permettent aussi de retrouver des instances persistantes.

paquetage de composant : Une unité de déploiement (fichier ZIP) contient la description externe d'une classe de composant, une ou plusieurs implémentations de la classe de composant, et un ensemble de descripteurs contenant des informations sur le composant ainsi que des caractéristiques non-fonctionnelles devant être appliquées sur les instances de composant.

composition : Lors de la composition, les facettes et réceptacles appartenant à diverses instances sont connectées ensemble pour créer des canaux de communication synchrones alors que les sources et puits d'évènements sont connectés ensemble pour créer des canaux de communication asynchrones ; des valeurs sont aussi données aux propriétés de configuration. Le résultat de la composition est décrit dans un Assembly qui décrit une composition de façon déclarative. Les balises permettant de réaliser un assemblage sont présentées dans la figure 3.9.b.

environnement d'exécution : Les caractéristiques non-fonctionnelles sont appliquées à travers des conteneurs et il s'agit essentiellement de caractéristiques similaires à celles décrites pour le modèle EJB. L'introspection est supportée à travers les interfaces `Navigation`, `Receptacles` et `Events` de l'interface de base `CCMObject`.

3.2.5 Fractal

Fractal est un framework de composition [BCS02] qui définit un modèle à composants 'générique' car il n'est pas orienté vers un domaine d'application particulier. Le framework contient un ensemble d'interfaces de programmation (API) orientées à la réalisation d'aspects tels que la définition de classes de composants, des fabriques, la reconfiguration dynamique des instances, la composition hiérarchique et l'introspection.

L'implémentation d'un composant est divisée en deux parties : le *contrôleur* et le *contenu*. Le contrôleur peut implémenter un nombre variable d'interfaces de contrôle dont un certain nombre sont définies dans la spécification du modèle à composants. Le contrôleur agit sur le contenu qui peut contenir du code fonctionnel ou bien être un ensemble d'autres contrôleurs ; de cette manière, le modèle supporte la composition hiérarchique. Lorsque le contrôleur agit sur le contenu, il joue un rôle similaire à celui du conteneur.

Une implémentation du framework nommée Julia[4] permet de décrire le contrôleur à partir d'un langage spécialisé et réalise ensuite un mélange du code de contrôle et du code fonctionnel à travers une approche à *mixins*. Un des aspects intéressants du modèle Fractal est qu'il n'impose pas un nombre fixe d'interfaces de contrôle, celles ci sont découvertes en temps d'exécution.

Caractéristiques

classe de composant : dans Fractal, une classe de composant possède une vue externe (appelée *ComponentType*) constituée par un ensemble d'interfaces qui sont fournies ou requises ; les interfaces sont nommées car la vue externe peut contenir plusieurs instances d'une même interface. Les interfaces peuvent de plus être catégorisées comme étant *obligatoires* ou *optionnelles* et ont une cardinalité simple (*singleton*) ou multiple (*collection*).

instance de composant : les instances de composant sont créées à partir de fabriques. La fabrique fournit une méthode de création (`newFcInstance`) qui retourne une nouvelle instance de composant à chaque invocation. Une instance de composant peut être activée ou désactivée si le contrôleur implémente l'interface `LifeCycleController`.

paquetage de composant : Fractal ne définit pas de mécanismes particuliers orientés à la réalisation du conditionnement des composants.

composition : la composition des instances d'un composant Fractal est réalisée à travers différentes interfaces de contrôle dont `AttributeController`, qui permet de réaliser la configuration d'une instance, `BindingController`, qui permet de réaliser la connexion des instances et `ContentController`,

[4] http ://fractal.objectweb.org/tutorials/julia/index.html

qui permet de créer de composites.
La composition peut être réalisée de façon déclarative à travers le langage *ADL Fractal*[5] qui permet de définir des composants et de composer des instances de ces derniers. La composition visuelle est supportée dans un outil appelé FractalGUI[6].

environnement d'exécution : le framework fournit la classe `Fractal` à partir de laquelle une instance de composant de démarrage est créée (cette instance est normalement au sommet de la hiérarchie).

3.2.6 Synthèse

Cette section a présenté une étude comparative de cinq modèles à composants soutenus par des acteurs industriels. La comparaison des modèles est réalisée selon une liste de caractéristiques obtenue de la description des concepts réalisée dans la première partie de ce chapitre. Le résultat de cette comparaison est résumé dans le tableau 3.1.

3.3 Conclusion

Ce chapitre a présenté une étude de l'approche à composants en décrivant dans un premier temps les concepts principaux de cette approche puis en réalisant une comparaison de cinq modèles à composant existant dans l'actualité.

L'étude de cette approche souligne plusieurs aspects : premièrement l'approche à composants est bâtie sur un cycle de vie de développement qui fait une séparation claire entre les activités de développement de composants et celles de leur assemblage. Ces deux activités sont réalisés par des acteurs spécialisés qui peuvent résider dans des emplacements différents. L'existence de ces deux activités explique diverses caractéristiques des composants, notamment l'existence d'une vue externe qui donne la vision d'un composant en tant que boite noire, ainsi que le fait que les composants soient conditionnés de façon à permettre leur livraison et déploiement indépendants.

La comparaison des différents modèles à composants permet de voir comment les concepts présentés dans la première partie sont implémentés. Cette comparaison montre aussi que les concepts décrits peuvent n'être que partiellement présents (par exemple la séparation entre la vue externe et la vue interne, ou l'existence d'interfaces de contrôle). Un autre aspect à noter est que certains modèles à composants sont très spécialisé par rapport à leur domaine d'application (JavaBeans, EJB et CCM) tandis que d'autres modèles sont plus 'génériques' (COM et Fractal).

[5] http ://fractal.objectweb.org/tutorials/adl/index.html
[6] http ://fractal.objectweb.org/current/doc/javadoc/fractal-gui/overview-summary.html

Caractéristique \ Plate-forme	COM	JavaBeans	EJB	CCM	Fractal
Classe de composant	Vue externe composée de plusieurs interfaces fournies) (au moins l'interface `IUnknown`)	Classe pouvant implémenter plusieurs interfaces et qui suit des patrons de nommage définissant des propriétés.	Une seule interface fournie, pas d'interfaces requises, pas de propriétés de configuration	Facettes, réceptacles, sources et puits d'événements + propriétés de configuration	Vue externe composée de plusieurs interfaces fournies ou requises, obligatoires ou optionnelles, simples ou multiples
Instance de composant	Promeut l'utilisation systématique de fabriques.	Pas de fabriques	Quatre catégories différentes : session avec état, session ans état, entité, message. Utilisation systématique de fabriques (homes)	Quatre catégories différentes : service, session, process et entity. Utilisation systématique de fabriques (homes)	Promeut l'utilisation systématique de fabriques. Une seule politique de création (instances multiples)
Paquetage de composant	DLL ou EXE, ne contient pas de ressources	Fichier JAR : code + classes de configuration et ressources mais pas d'information sur les composants	Fichier Jar pour composants, EAR pour applications. Descripteur de déploiement.	Fichier Zip	Ne définit pas la manière de conditionner un composant
Composition	Impérative à travers langage de programmation ou langage de script	Surtout Visuelle	Impérative à travers langage de programmation Java	Composition déclarative donnant lieu à un Assembly	Déclarative avec FractalADL, Visuelle à travers FractalGUI
Environnement d'exécution	Runtime COM	Machine virtuelle Java	Runtime EJB + conteneurs	Intergiciel CORBA + conteneurs	Contrôleurs
Autres particularités	Mécanismes d'introspection employés pour supporter l'évolution des interfaces	Composants peuvent être livrés avec des classes destinées à l'environnement d'assemblage	Domaine Application précis (tiers milieu)	Domaine Application précis, Définit processus de déploiement à partir d'assembly	Générique, découverte dynamique des interfaces de contrôle.

TAB. 3.1 – Comparaison des modèles à composants

Chapitre 4

Approche à services

Ce chapitre présente une étude de l'approche à services à travers une description des concepts principaux suivie d'une comparaison de diverses plateformes à services. Ce chapitre suit une structure similaire au chapitre qui présente l'approche à composants.

4.1 Concepts principaux

Comme l'approche à composants, l'approche à services souffre de manque de clarté au niveau de la description de ces concepts. L'objectif de cette section est de décrire ces concepts en dehors d'une technologie particulière.

4.1.1 Introduction

L'approche à services est aujourd'hui en vogue du fait de la récente apparition des services web ; cette approche n'est cependant pas nouvelle et ses concepts sont déjà présents dans les mécanismes de courtage des systèmes distribués (par exemple dans l'ODP trader, 1992 [IBR93]). Un service peut être décrit comme une fonctionnalité dont le comportement est défini de façon contractuelle et dans l'approche à services, les entités fournissant un telle fonctionnalité, les *fournisseurs de service*, sont découvertes en temps d'exécution à l'aide d'un intermédiaire. Une application construite à partir de services utilise un ensemble de services ; les fournisseurs de ces services ne sont cependant pas cablés dans l'application, et ils peuvent changer à travers les différentes exécutions de celle-ci. L'approche à services promeut la réutilisation, car un même service peut être utilisé dans différentes applications.

Historique

L'interaction propre à l'approche à services, dans laquelle un intermédiaire permet de découvrir les fournisseurs de services pendant l'exécution, peut être vue comme un résul-

tat de l'évolution dans la manière de créer une liaison entre un client et un serveur. Dans les premiers systèmes distribués, un identifiant numérique propre à chaque serveur était fixé au niveau du client, ce qui rendait difficile la réalisation de changements dans la liaison. Ensuite, un niveau d'indirection a été ajouté en donnant au client un nom identifiant le serveur ; avant de se connecter avec le serveur, le client passe par un intermédiaire qui traduit le nom vers l'identifiant numérique du serveur, ce qui rend les changements dans la liaison possibles au niveau de l'intermédiaire sans besoin de changer le client. L'approche à services pousse ce raisonnement plus loin : un client ne connaît ni l'adresse ni le nom d'un serveur mais uniquement le service, c'est à dire les fonctionnalités, que doit offrir le serveur. Un intermédiaire entre le client et le serveur permet au client de *découvrir*, c'est à dire de localiser, un serveur au moyen d'une demande contenant de l'information sur le service que doit fournir le serveur. Lorsqu'un serveur a été localisé, le client réalise une *liaison* avec lui. Bien que l'approche à services soit apparue dans le contexte des systèmes distribués, les concepts de cette approche sont indépendants de ce type de systèmes.

4.1.2 Définition

Un *service* est une fonctionnalité réutilisable qui est décrite de façon contractuelle dans un *descripteur de services*. Un tel descripteur inclut des informations syntaxiques, c'est à dire une *interface de service*, mais peut aussi contenir des informations qui décrivent le comportement ou qui caractérisent le service. Un *fournisseur de services* fournit des *objets de service* qui implémentent l'interface du service et permettent à un client d'utiliser sa fonctionnalité. Du fait que le services sont décrits de façon contractuelle, de multiples fournisseurs peuvent exister pour un même service, de plus, les différents fournisseurs d'un même service sont substituables.

Une caractéristique fondamentale de l'approche à services est que l'assemblage d'une application à base de services est réalisé à partir de descripteurs de services. La *découverte* et la *liaison* avec des fournisseurs de services n'a lieu que de façon tardive, c'est à dire avant ou pendant l'exécution de l'application. En conséquence, l'approche à services se focalise sur la description et organisation des services, de façon à supporter la découverte dynamique des fournisseurs de services en temps d'exécution [Bur00].

4.1.3 Acteurs et activités

La découverte de services se fait à partir d'une interaction entre trois acteurs (voir figure 4.1) qui sont :

Fournisseur de services : Le fournisseur de services a pour rôle de produire des objets de service qui implémentent une ou plusieurs interfaces de services. Pour supporter la découverte des fournisseurs de service, les *descripteurs de services* correspondant aux services d'un fournisseur doivent être publiés auprès

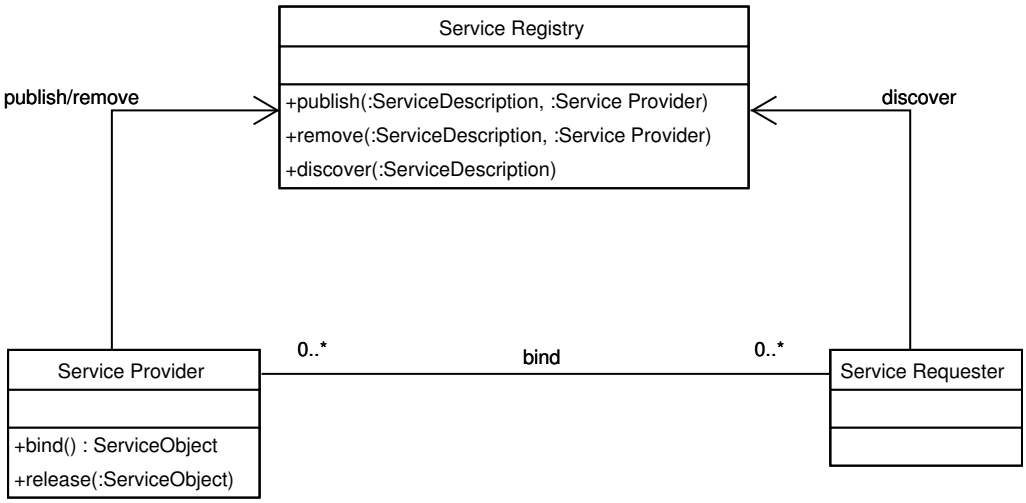


FIG. 4.1 – Acteurs de l'approche à services

d'un registre de services (cette publication peut être réalisée soit par le fournisseur, soit par un tiers). Les services d'un fournisseur ont une *disponibilité dynamique*, ils peuvent à tout moment être ajoutés ou retirés du registre de services.

Demandeur de services : Cet acteur est le client d'un service particulier. Pour pouvoir interagir avec le service, le demandeur de services doit être lié à un fournisseur du service qui doit être découvert auparavant dans le registre de services. La découverte d'un fournisseur dans le registre de services est réalisée à partir de critères relatifs à la description du service qu'il fournit. Lors de la liaison, le demandeur de services obtient un objet de service de la part du fournisseur. La découverte et la liaison peuvent être réalisées par le demandeur lui même ou bien par un tiers.

Registre de services : Le registre de services est l'intermédiaire entre les fournisseurs et les demandeurs de services. Le registre contient un ensemble de descripteurs de services ainsi que des références vers les fournisseurs de ces services. Le registre fournit des mécanismes permettant de l'interroger pour obtenir des références vers les fournisseurs de services (voir 4.1.6). L'ensemble de descripteurs de services qui se trouvent dans le registre change constamment.

Le patron d'interaction qui caractérise l'approche à services est présentée dans le diagramme de séquence de la figure 4.2. Ce diagramme montre un fournisseur de services qui publie la description de ses services dans un registre de services. Un demandeur de services interroge ensuite le registre pour découvrir des fournisseurs d'un service particulier à partir d'un ensemble de critères relatifs au descripteur du service. Si des fournisseurs répondant aux critères ont été publiés au préalable dans le registre, ce dernier retourne

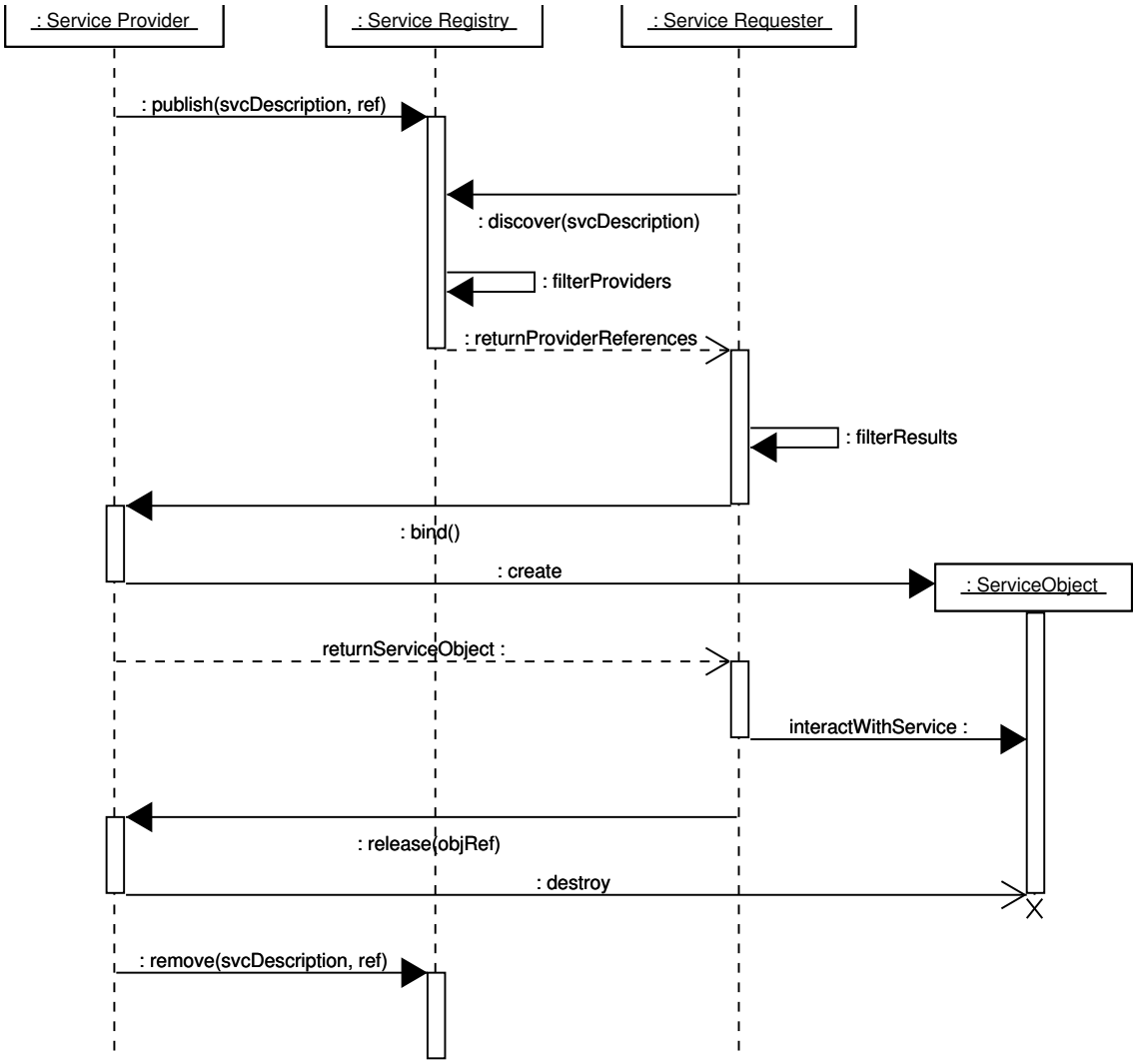


FIG. 4.2 – Patron d'interaction de l'approche à services

des références vers les fournisseurs au demandeur. Le demandeur doit ensuite choisir des fournisseurs appropriés et se lier avec eux. Au moment de la liaison, un fournisseur de service retourne au demandeur un objet de service. Une fois l'utilisation du service terminée, le demandeur de services termine d'utiliser le service en libérant l'objet implémentant le service.

4.1.4 Descripteur et objet de service

Cette section décrit les deux entités qui constituent un service : le descripteur et l'objet de service.

4.1.4.1 Descripteur de service

Un service est décrit à travers un descripteur de services qui contient une interface de service mais qui peut aussi contenir des propriétés de service permettant d'identifier le fournisseur du service ou des caractéristiques du service non associées à son comportement (par exemple le niveau de qualité du service). Bien qu'un service soit défini comme ayant un comportement défini de façon contractuelle, la technologie actuelle ne permet pas de décrire un comportement de façon adéquate pour permettre aux clients de services de découvrir les fournisseurs à travers une description du comportement (bien que des travaux académiques comme [PF03] cherchent à traiter ce problème). La solution actuelle à ce problème est de présenter l'interface de service comme un contrat syntaxique et lui associer un contrat sémantique qui peut être décrit, par exemple, dans la documentation accompagnant l'interface ; les interfaces de service peuvent, d'un autre côté, être définies par des organismes de standardisation. Cette limitation rend de plus la substituabilité des fournisseurs de services quelque chose de théorique, car il est impossible de garantir que divers fournisseurs d'un même service obéissent réellement au contrat défini par le service.

4.1.4.2 Objet de service

Les services sont parfois considérés comme étant des entités sans état qui ne supportent que des interactions de type requête/réponse[1], cependant cette vision est très limitée et ne correspond pas à la réalité. Dans la pratique, un demandeur de services interagit avec un objet de service, qui peut posséder un état, et qui implémente l'interface de service. Cet objet est retourné par un fournisseur de services au moment où un demandeur de services se lie avec lui. Les politiques de création des objets de service, l'état et la libération des objets de service sont décrits à la suite.

[1] Voir par exemple le type de composant "service" dans le modèle CCM, dans la section 3.2.4.

Politiques de création

Une caractéristique particulière de l'approche à services est que le demandeur de services n'a typiquement pas de connaissances sur les politiques suivies par le fournisseur de services pour la création des objets de service. Les politiques que peut suivre un fournisseur de services lors de la création des objets de service sont :

objet partagé : dans ce cas, le fournisseur de services crée un unique objet (singleton) qui est retourné à tous les demandeurs de services lorsqu'ils se lient avec le fournisseur ; l'objet est donc partagé par tous les demandeurs. Une variante de ce cas a lieu quand le fournisseur de services est lui même l'objet de service et donc au moment de la liaison il retourne une référence vers lui même.

stock d'objets : dans ce cas le fournisseur de services crée un ensemble fini, ou stock, d'objets de service. Un objet différent est retourné à chaque demandeur lors de la liaison selon la disponibilité du stock. Une fois qu'un objet est libéré par un demandeur, il est remis dans le stock pour être réutilisé dans une prochaine liaison. Dans ce cas les objets de service sont partagés par les clients mais pas de façon simultanée car deux demandeurs n'obtiennent jamais une référence à un même objet. Cette situation est utile par exemple lorsque des ressources, comme la mémoire, sont limitées, ou bien lorsque chaque objet de service représente une ressource physique, comme par exemple un port de communication, dont la quantité est limitée.

un objet par demandeur : dans ce cas, un objet de service est crée pour chaque demandeur. Si un même demandeur est lié plusieurs fois au fournisseur de services, il obtient toujours le même objet de service. Cette politique requiert de pouvoir identifier le demandeur de services du côté du fournisseur. Cette politique est utile, par exemple, lorsque le demandeur et l'objet de services ne maintiennent pas une connexion continue (par exemple si la communication se fait à travers un protocole sans connexion comme HTTP).

un objet par liaison : dans ce cas, un objet différent est crée chaque fois qu'une liaison est réalisée vers un fournisseur de services.

État d'un service

Un service est défini comme ayant un état ("stateful") si il est capable de maintenir un état au long de plusieurs appels vers les méthodes de son interface par un même client. Cependant, le choix de la politique de création des objets de service peut rendre un état partagé par tous les clients, ce qui peut être problématique lorsqu'un client dépend de l'état du service où bien lors de modifications concurrentes de cet état. En général, un

fournisseur dont les services ont un état doit plutôt suivre une politique différente à celle de la création d'un seul objet partagé. La politique de création d'un objet par liaison peut être employée, mais elle implique que le demandeur est au courant de cette situation et ne réalise qu'une seule liaison avec le fournisseur puis garde l'objet retourné pendant toute son interaction avec le service. La politique de création d'un objet partagé est adéquate pour des services n'ayant pas un état ("stateless").

Politiques de libération

A la fin d'une interaction entre un demandeur et un service, le demandeur doit libérer l'objet de service. Cette action est nécessaire du fait que le fournisseur de services peut avoir besoin de savoir à quel moment un objet de service n'est plus utilisé pour détruire l'objet ou bien pour l'utiliser lors d'une autre liaison. La libération des objets de service peut être réalisée suivant deux politiques :

explicite : Lorsque la libération est explicite, le demandeur de services informe le fournisseur que son interaction avec le service est terminée.

expiration de bail : Dans cette politique, le demandeur ne doit pas demander la libération de l'objet de service car ce dernier n'est valable que pour une durée de temps limitée, ou *bail*. Lors de l'expiration du bail, le demandeur de services doit faire une nouvelle liaison vers le fournisseur pour obtenir un autre objet de service. Cette politique peut être employée par exemple en collaboration avec une politique de création de stock d'objets, pour garantir qu'au bout d'un certain temps les objets de service soient libérés.

4.1.5 Composition de services

La composition de services représente l'utilisation d'un ensemble de services pour réaliser une tâche particulière. Du fait que l'approche à services se focalise sur l'interaction permettant de réaliser la publication et la découverte de services, la composition des services est souvent considérée comme étant une responsabilité des demandeurs de services.

La composition de services implique l'intégration de divers services dans un programme qui utilise ces services, et qui joue donc le rôle de demandeur de services. Un tel programme contient un flot de contrôle et de données qui coordonne l'invocation des services ainsi que le transfert de données entre les différents services. Le programme coordinateur peut être écrit dans un langage de programmation standard, cependant l'utilisation de flots exécutables ("workflow") [LR97] est employée dans les services web (voir Orchestration dans 4.2.5).

Une composition de services est écrite en termes d'interfaces de services et est abstraite jusqu'au moment de l'exécution, où des fournisseurs des services utilisés sont dé-

couverts et liés. La composition de services doit faire face à plusieurs problématiques propres à l'approche à services, ces problématiques incluent la disponibilité des services, le filtrage des réponses retournées par le registre et l'ignorance des politiques de création des objets de services. Ces problématiques ont été traitées particulièrement dans le cadre des services web. La découverte des fournisseurs de services dont les services seront utilisés dans la composition peut être réalisée avant que l'exécution du programme coordinateur démarre, ou au fur et à mesure de son exécution. Dans le premier cas, il est cependant possible qu'au milieu de l'exécution, un fournisseur de services devienne indisponible. Ce problème est adressé dans l'approche à services web à l'aide de mécanismes de transaction, qui permettent de revenir en arrière dans l'exécution si l'invocation d'un service échoue. La sélection des réponses dans le cadre des services web n'est pas actuellement une problématique traitée, car celle-ci est réalisée de façon manuelle.

4.1.6 Environnement d'exécution

Dans l'approche à services, un environnement d'exécution fournit divers mécanismes nécessaires aux fournisseurs et aux demandeurs de services à la réalisation de l'interaction propre à cette approche. Ces mécanismes sont présentés ici.

4.1.6.1 Opérations sur le registre

Trois opérations essentielles peuvent être réalisées sur le registre de services :

publication : Cette opération est destinée aux fournisseurs de services, elle permet d'ajouter au registre un descripteur de service associé à une référence vers un fournisseur du service.

retrait : Cette opération est destinée aux fournisseurs de services, elle permet de retirer un descripteur de service préalablement publié par le fournisseur dans le registre.

découverte : Cette opération est destinée aux demandeurs de services, elle permet d'interroger le registre pour obtenir des références vers des fournisseurs de services ayant enregistré leurs descripteurs au préalable. Pour réaliser la découverte, le demandeur doit envoyer des informations au registre permettant de filtrer le nombre de réponses retournées au demandeur (*filtrage côté registre*). Ces informations sont typiquement construites à partir des propriétés de service. Le registre de services retourne un ensemble de références vers des fournisseurs ayant les critères requis par le demandeur, cependant la décision finale dans le choix du fournisseur est laissée au demandeur qui doit faire une deuxième sélection sur les réponses retournées par le registre (*filtrage côté client*). La sélection d'un fournisseur particulier lorsque de multiples fournisseurs sont disponibles (*disponibilité multiple*) est un problème complexe

car il requiert la définition de critères permettant de favoriser un fournisseur par rapport à un autre. La prise en charge de la disponibilité multiple est couramment laissée aux programmeurs.

4.1.6.2 Notifications

Les services ont une disponibilité dynamique : ils peuvent être publiés ou retirés du registre à tout moment en raison de changements au niveau de leurs fournisseurs. Il est important pour les demandeurs de services d'être au courant des changements dans les services, pour pouvoir par exemple incorporer un nouveau service qui devient disponible, ou bien pour arrêter l'utilisation d'un service qui devient indisponible. Pour faciliter cette tâche, certains environnements d'exécution permettent aux demandeurs de services de s'abonner afin d'être notifiés de changements dans les services. La notification concerne principalement les évènements suivants :

service publié : évènement ayant lieu quand un service est publié dans le registre.

service retiré : évènement ayant lieu quand un service est retiré du registre.

service modifié : évènement ayant lieu quand un service est modifié sans être retiré du registre (une modification peut par exemple concerner des changements dans les propriétés qui caractérisent le service).

Lorsque l'environnement d'exécution ne fournit pas de mécanismes de notification, le demandeur de services peut interroger de façon périodique le registre de services pour savoir à quel moment un nouveau service est enregistré ou retiré.

4.1.7 Synthèse

Cette section a présenté les concepts principaux de l'approche à services. Ces concepts concernent l'interaction ayant lieu entre des demandeurs, fournisseurs et registres de services pendant l'exécution pour réaliser la liaison entre les demandeurs et les fournisseurs. A partir de ces concepts il est possible de dresser la liste des caractéristiques utiles pour catégoriser une *plate-forme à services*, c'est à dire une réalisation particulière de l'approche à services. Ces caractéristiques sont les suivantes :

Description : caractérise la façon suivant laquelle les services sont décrits.

Publication : caractérise les opérations fournies par le registre destinées aux fournisseurs de services pour réaliser la publication et le retrait de services.

Découverte : caractérise les opérations fournies aux demandeurs de services pour réaliser la découverte et la liaison vers un fournisseur, ainsi que les mécanismes supportés pour le filtrage côté registre.

Politiques de création : caractérise les politiques supportées par un fournisseur de services lors de la création d'objets de service.

Notifications : caractérise les mécanismes et types de notifications supportées par la plate-forme.

Libération : caractérise les politiques suivies lors de la libération de service.

Particularités : autres caractéristiques particulières de la plate-forme (système centralisé ou réparti, nombre de registres, etc...)

4.2 Étude de plateformes à services

Cette section présente une étude de cinq plateformes à services existant aujourd'hui. Chacune est présentée en décrivant d'abord son domaine d'application puis par rapport à la liste des caractéristiques établie dans la section précédente.

4.2.1 Courtier CORBA

Un courtier CORBA [Str98] [LMMG02] est un registre de services qui permet de réaliser l'interaction propre à l'approche à services dans un système distribué. Le courtier fait partie d'un ensemble de mécanismes fournis par l'intergiciel CORBA [Vin97], et en conséquence, son domaine d'application est le même que celui de ce dernier, c'est à dire celui de la construction de systèmes répartis. Le courtier de CORBA se distingue d'autres plateformes à services par le fait que CORBA supporte la création de réseaux de courtiers (appelés *fédérations*) pouvant se relayer pour augmenter le nombre de réponses données à une demande ; ces réseaux peuvent de plus évoluer de façon continue.

Caractéristiques

Description : Dans CORBA, la description des services (appelés *type de service*) se fait dans le langage de description d'interfaces (IDL) au moyen du mot clé `service`. La description d'un service contient une référence vers l'interface de service (aussi en IDL) ainsi qu'un ensemble de propriétés caractérisant le service (voir figure 4.3.a). Le nombre de propriétés du descripteur est fixe mais les propriétés peuvent être marquées comme étant obligatoires ou optionnelles ainsi que modifiables ou non. Une description de service doit être publiée dans un registre propre aux descriptions avant que les fournisseurs ne puissent publier leurs services (appelées *offres* de services).

Publication : Les opérations de publication et de retrait de services dans le registre sont appelées respectivement : `export` et `withdraw` (voir figure 4.3.b et 4.3.c).

a) Description

```
interface PrinterService {       // interface de service
  typedef unsigned long JobID;
  JobID print (in string data);
};
service PrinterServiceDescription { // description
  interface PrinterService;
  mandatory property string building;
  property short floor;
  mandatory property string type;
  mandatory property string language;
  property string name;
};
```

b) Publication

```
Property[] props = new Property[5];
props[0]="Laboratory";
props[1]=(short)3;
props[2]="Color";
props[3]="Postscript";
props[4]="LabPrinter";
String id = reg.export(printer, "PrinterService", props);
```

c) Retrait

```
reg.withdraw(id);
```

d) Découverte

```
lookup.query( "PrinterService",
              "((color == 'black') and
                      (language == 'postscript'))",
              // Contrainte
              "min (floor)", // Ordre des reponses
              policies, // Politiques
              desiredProps, // Propriétés
              20, // Nombre de réponses
              servicerefs, // Résultats
              refsiterator, limits);
```

e) Liaison

```
Offer :
  building = '36'
  color = 'black'
  floor = 2
  language = 'postscript'
  Reference : IOR :00000000002449444c3a6f6d672e6f7 ...
Pas d'opération de liaison car la demande retourne
des référence vers l'objet de service.
```

FIG. 4.3 – Exemple courtier CORBA

Lors de la publication d'un service les propriétés définies au niveau du descripteur sont remplies.

Découverte : L'opération de découverte est appelée `query` (voir figure 4.3.d). Cette méthode permet de faire des requêtes sophistiquées car elle reçoit une contrainte basée sur les propriétés déclarées dans la description du service, des préférences permettant de changer l'ordre des réponses et des politiques destinées à limiter la propagation d'une requête dans une fédération de courtiers. Il n'y a pas d'opération de liaison car le registre retourne un ensemble d'offres contenant des références vers les objets fournissant le service (voir figure 4.3.e).

Politiques de création : CORBA fait une différence explicite entre l'enregistrement d'un objet partagé par tous les demandeurs de services et l'enregistrement d'une fabrique (appelée *proxy*) permettant d'avoir des politiques d'instantiation différentes. Un courtier doit implémenter une interface spécifique pour permettre l'enregistrement de fabriques.

Notifications : Pas de notifications.

Libération : Le demandeur de services doit libérer explicitement l'objet de service (à travers la méthode `remove` de l'objet).

Particularités : Dans ce système, de nature distribuée, un réseau dynamique de courtiers, appelé fédération, peuvent collaborer entre eux pour répondre à une requête réalisée par un client.

4.2.2 Contexte JavaBeans

Le concept de contexte des composants JavaBeans a été introduit dans une spécification postérieure à celle du modèle à composants [Cab98] (voir description du modèle dans la section 3.2.2). Ce concept a été introduit pour fournir des moyens de grouper des instances de JavaBeans dans des *contextes* d'exécution, pouvant à leur tour être organisés dans des hiérarchies, et pour permettre aux instances d'obtenir des services de la part du contexte qui les contient lors de l'exécution d'une application.

Le domaine d'application de cette plate-forme est celui des applications centralisées pouvant être assemblées de façon visuelle et qui normalement supportent l'interaction avec un utilisateur. Dans le BeanContext, un seul fournisseur d'un services particulier peut être enregistré dans le registre à un moment donné.

Caractéristiques

a) Description

```
interface PrinterService {
 public long print(String data);
};
```

b) Publication

```
BeanContextServiceProvider provider
             = new PostscriptPrinterProvider();
beancontext.addService(PrinterService.class,printerprovider);
```

c) Retrait

```
boolean revokeNow = true; // libération immédiate?
beancontext.revokeService(PrinterService.class
                   printerprovider, revokeNow);
```

d) Découverte

```
//tester présence du service
if(beancontext.hasService(PrinterService.class)==true) {
```

e) Liaison

```
  Object service = beancontext.getService(
               child, //instance bean associée à demande
               child, //demandeur du service
               PrinterService.class,
               paramsConfig, //paramètres configuration
               child //listener des évènements de retrait
  );
 ((PrinterService)service).print(data);
}
```

f) Libération

```
beancontext.releaseService(
            child,        //instance de bean associée à demande
            requestor,    //demandeur
            service);     //service
```

FIG. 4.4 – Exemple contexte JavaBeans

Description : Dans le BeanContext, un service est décrit à travers une interface ou classe Java. Du fait qu'un seul fournisseur peut être enregistré, il n'y a pas de propriétés permettant de différencier les fournisseurs.

Publication : Les opérations de publication et de retrait de services dans le registre sont appelées respectivement : `addService` et `revokeService` (voir figure 4.4.b et 4.4.c). La publication d'un service ne permet pas de donner des propriétés caractérisant le service du fait qu'un seul fournisseur par service peut être enregistré.

Découverte : Le BeanContext offre une opération, appelée `hasService` (voir figure 4.4.d), permettant de tester si un service est présent ou non, et une opération appelée `getService` (voir figure 4.4.e) permettant de réaliser la liaison vers le fournisseur du service. Lors de la liaison, un client peut passer des paramètres d'initialisation au service[2] et est enregistré automatiquement aux notifications de départ du service obtenu.

Politiques de création : Un fournisseur de services doit implémenter la méthode `getService` qui reçoit entre autres une référence vers le demandeur de service. Ceci permet d'implémenter l'ensemble des politiques de création d'objets implémentation.

Notifications : Le BeanContext permet aux demandeurs de services de s'abonner pour recevoir des évènements signalant l'arrivée (`serviceAvailable`) et le départ d'un service (`serviceRevoked`). Lorsqu'un demandeur se lie à un fournisseur, il est abonné automatiquement aux évènements signalant le départ du service.

Libération : Les demandeurs de service doivent libérer les objets de service de façon explicite (voir figure 4.4.f).

Particularités : Le BeanContext est un système centralisé. La spécification permet la création de hiérarchies de contextes et décrit la possibilité pour un contexte de déléguer une demande de service vers un contexte parent.

4.2.3 Jini

Jini [AOWW99] [New01] est une plate-forme de services distribuée promue par Sun et qui partage un grand nombre de concepts avec la plate-forme de courtiers de CORBA. Jini repose sur le langage Java et bénéficie particulièrement de la capacité qu'offre Java de télécharger du code à travers le réseau. Grâce à cela, Jini supporte le fait que l'objet de

[2] Ceci semble contradictoire à l'approche à services, car cela suppose que le demandeur connaît l'implémentation du service.

service soit envoyé sur le même emplacement où se trouve le demandeur de services (bien que la distribution reste possible si cet objet joue le rôle de souche, de là qu'il soit appelé *proxy* dans la spécification). Cette caractéristique différencie Jini des courtiers CORBA dans lesquels la communication entre demandeurs de services et objet d'implémentation se fait toujours à travers un appel à distance. Jini supporte de façon explicite les politiques de libération des objets de service par expiration de bail. Une autre caractéristique de Jini est que clients et fournisseurs de services doivent trouver un registre avant de commencer à interagir. Un registre peut être connu par une adresse fixe ou bien il peut être découvert à partir de la diffusion d'une demande. Les services dans Jini peuvent être organisés par groupes (par exemple groupe services d'impression) et un registre peut héberger un groupe de services particulier. Bien que Jini supporte l'existence de registres multiples, il n'offre pas des mécanismes permettant aux registres de déléguer une demande de service ; au lieu de cela, les fournisseurs de services enregistrent leur services dans de multiples registres.

Caractéristiques

Description : Dans Jini, un service est décrit par une interface Java et un nombre variable de propriétés (objets de sous-classes de la classe `Entry`).

Publication : La publication de services est réalisée à travers la méthode `register` (voir figure 4.5.b) . Lors de la publication, un fournisseur de services définit un temps de bail (*lease*) pendant lequel le service sera disponible. Avant que ce délai termine, le fournisseur doit renouveler le bail pour éviter le retrait de ses services du registre. Pour retirer un service, le fournisseur peut attendre la fin du bail ou bien forcer son expiration (voir figure 4.5.c).

Découverte : La découverte des services se fait à travers la méthode `lookup` qui permet de spécifier un nombre maximum de réponses, voir figure (4.5.d). Jini ne propose cependant pas de mécanismes flexibles pour réaliser le filtrage des fournisseurs appropriés du côté du registre. Le critère pour déterminer si un service correspond à une demande est que les interfaces du service coïncident avec celles demandées et que les propriétés envoyées par le demandeur soient présents dans la description de service.

Politiques de création : Lors de la publication d'un service, le fournisseur inclut une référence vers l'objet de service et ce dernier est copié dans le registre. Quand un demandeur de service obtient une réponse du registre, il obtient à son tour une copie de l'objet de service. Par défaut, la politique de création est donc un objet par liaison, cependant si l'objet obtenu joue le rôle de souche (stub) envers un objet distant, la politique de création devient un objet partagé, car tous les demandeurs interagissent alors avec le même objet distant.

a) Description

```
interface PrinterService extends Remote{
 public long print(String data) throws RemoteException;
};
```

b) Publication

```
ServiceRegistrar reg = findRegistry();   // trouver registre
Entry entries[]={printerType,resolution,...}; // propriétés
ServiceItem printsvc = new ServiceItem(
                        null,         // id de service
                        serviceObject,// objet serialisable
                        entries);     // propriétés
ServiceRegistration svcreg = reg.register(registration,
               1000000); // durée de validité (ms)
```

c) Retrait

```
// Expiration de bail ou bien :
Lease lease = svcreg.getLease();
lease.cancel();
```

d) Découverte

```
Class svcInterfaces []={PrinterService.class};
Entry entries[]={printerType};
ServiceTemplate template = new ServiceTemplate(
    null, svcInterfaces, entries);
ServiceMatches matches = reg.lookup(template,3); // 3 max
```

e) Liaison

```
if(matches.totalMatches>0)
{
  ((PrinterService)matches.items[0]).print(data);
}
```

FIG. 4.5 – Exemple Jini

Notifications : Jini fournit des mécanismes de notification asynchrones permettant aux clients d'être informés des changements au niveau des services : `TRANSITION_NOMATCH_MATCH` signale l'arrivée d'un service, `TRANSITION_MATCH_NOMATCH` signale le départ d'un service et `TRANSITION_MATCH_MATCH` signale un changement au niveau d'un service (par exemple la valeur de ses propriétés). Un client doit s'enregistrer auprès du registre à travers la méthode `notify`.

Libération : Un objet de service devient invalide lors de l'expiration du bail.

Particularités : Jini est un système distribué qui supporte l'existence d'un nombre indéfini de registres de services. Jini prend en compte des aspects tels que la sécurité et les transactions.

4.2.4 OSGi

L'Open Services Gateway Initiative (OSGi) est une corporation qui se charge de définir et promouvoir des spécifications ouvertes pour réaliser la livraison de services administrés dans des réseaux résidentiels ou autres types d'environnements restreints (voitures, etc...). La spécification OSGi [Ope03] définit une plate-forme Java non-distribuée de services ainsi que des moyens permettant de réaliser le déploiement des fournisseurs et des demandeurs de services à l'intérieur de la plate-forme (appelée *framework*). Dans OSGi, les services sont livrés et déployés dans des unités logiques et physiques appelées *bundles*. Du côté physique, un bundle correspond à un fichier JAR contenant du code et des ressources ; du côté logique, un bundle correspond à un demandeur ou fournisseur de services (un bundle peut aussi jouer les deux rôles en même temps). Le framework fournit des mécanismes d'administration permettant de réaliser l'installation, activation, dé-activation, mise à jour et dé-installation des bundles physiques de façon continue. L'activation ou déactivation d'un bundle physique résulte dans l'activation ou déactivation du bundle logique correspondant. Lorsque le bundle logique est actif, il peut publier ou découvrir des services et se lier avec d'autres bundles à travers un registre de services fourni par la plate-forme.

Caractéristiques

Description : Dans OSGi, un service est décrit à travers une interface ou classe Java et un nombre variable de propriétés de type clé-valeur. Bien que ces propriétés ne soient pas spécifiées dans l'interface elles sont spécifies dans la documentation de l'interface du service.

Publication : L'opération de publication dans le registre est appelée `registerService` (voir figure 4.6.b). Cette méthode reçoit le nom de l'interface du service ainsi

a) Description

```
interface PrinterService{
 public long print(String data);
};
```

b) Publication

```
// implémentation du service
class PSPrinter implements PrinterService, Configurable{
...}
PrinterService printersvc = new PSPrinter();
Dictionary props = new Dictionary();
props.put("printertype","Postscript");
props.put("color","true");
ServiceRegistration reg = bundlecontext.registerService(
        PrinterService.class.getName(),  // Nom de l'interface
        printersvc,                      // Objet de service
        props);                          // propriétés
```

c) Retrait

```
reg.unregister();
```

d) Découverte et liaison

```
ServiceReferences refs[]=getServiceReferences(
         PrinterService.class.getName(),
         "(&(printertype=Postscript)(color=*))" //filter
);
if(refs!=null){
  PrinterService printer
        =(PrinterService)bundlecontext.getService(refs[0]);
  if(printer instanceof Configurable)
  {...configure the service...}
  printer.print(data);
}
```

e) Libération

```
bundlecontext.ungetService(refs[0]);
```

FIG. 4.6 – Exemple OSGi

que l'objet de service et un dictionnaire contenant une liste de propriétés. L'enregistrement retourne une référence au service qui est ensuite employée pour retirer le service à travers l'opération `unregister` (voir figure 4.6.c).

Découverte : La découverte de services se fait à travers la méthode `getServiceReferences` qui retourne un ensemble d'objets représentant des références vers les fournisseurs de services. La liaison vers le fournisseur de services est explicite et se fait à travers la méthode `getService`. La méthode de découverte reçoit une chaine contenant un filtre dans une syntaxe LDAP[3] qui permet au registre de faire une sélection des fournisseurs de services basés sur la liste des propriétés employées au moment de la publication.

Politiques de création : OSGi supporte deux politiques différentes de création d'objets de service : création d'un objet partagé ou bien création d'un objet par demandeur. Pour implémenter la deuxième politique, lors de la publication du service, un objet de type `ServiceFactory` doit être enregistré. Cet objet est une fabrique chargée de créer des instances de l'objet de services propres à chaque demandeur. OSGi considère toutes les demandes de services provenant du même bundle comme faisant partie du même demandeur, et la fabrique de services reçoit une référence à un identifiant du bundle d'où la requête est originaire.

Notifications : OSGi permet aux demandeurs de services de s'enregistrer pour recevoir des évènements indiquant des changements dans les services. Lors de l'enregistrement, il est possible de donner un filtre permettant de limiter le nombre d'évènements reçus. Les évènements concernant les services supportés par la plate-forme sont service enregistré (`REGISTERED`), service retiré (`UNREGISTERING`), et service modifié (`MODIFIED`).

Libération : La libération des objets de service se fait de façon explicite à travers la méthode `ungetService`.

Particularités : OSGi est un système centralisé avec un seul registre. La spécification OSGi prend en compte les aspects sécurité et définit aussi un certain nombre de services standard (LogService, HttpService, etc...)

Liaison de services

Un mécanisme de composition de services a été introduit dans la 3ème livraison de la spécification de OSGi, ce mécanisme est appelé *Wire Admin Service*. A travers ce mécanisme, des producteurs sont liées à des consommateurs à l'aide de connecteurs appelés *wires*. Un producteur est un service qui génère de l'information qui est reçue par un

[3]Lightweight Directory Access Protocol

consommateur. Un connecteur définit une association entre un producteur et un consommateur et plusieurs connecteurs peuvent exister entre une paire de producteurs et de consommateurs. Chaque producteur et consommateur possèdent un identificateur fixe et au moment où les deux entités deviennent disponibles, un administrateur de connecteurs (*wire admin*) est chargé de les lier à travers des connecteurs. La liaison entre producteurs et consommateurs est donc définie de façon statique, cependant les connexions peuvent apparaître ou disparaître pendant l'exécution selon la présence des producteurs et des consommateurs. Les interfaces de service des producteurs et de consommateurs doivent hériter d'interfaces de base qui déclarent des méthodes de liaison.

4.2.5 Services web

D'après [AF01] et [CNW01], l'architecture à services web apparaît pour permettre à des applications hétérogènes s'exécutant au sein de différentes entreprises de pouvoir inter-opérer à travers des services offerts par les applications. L'hétérogénéité des applications n'est pas seulement considérée au niveau des langages d'implémentation des applications, mais aussi au niveau des modèles d'interaction, protocoles de communication et niveaux de qualité des services.

La description des services web est réalisée dans un langage appelé WSDL (web service description language) [Wor01]. WSDL, qui est basé sur une grammaire XML, permet de décrire l'interface du service, les types de données employés dans les messages, le protocole de transport employé dans la communication et des informations permettant de localiser le service spécifié. Le registre de services, appelé UDDI (universal description, discovery and integration) [udd02] supporte l'enregistrement de descriptions de services (appelés *types de services*) ainsi que de fournisseurs de services (des *entreprises*). UDDI fournit des moyens de publier une entreprise à travers des pages blanches, pages jaunes et pages vertes. Les pages blanches contiennent le nom de l'entreprise et sa description, les pages jaunes catégorisent une entreprise et les pages vertes décrivent les moyens d'interaction avec une entreprise (description de services fournis, processus, etc...). UDDI est un registre distribué dans lequel l'information est répliquée sur plusieurs sites, en conséquence, un fournisseur de services ne doit publier sa description de service que vers un seul noeud du réseau de registres. Le protocole de communication par défaut est SOAP (simple object access protocol).

Bien que la plate-forme des services web soit fondée sur les concepts de l'approche à services, l'interaction propre à cette approche n'est actuellement pas suivie de façon systématique. Il faut souligner que l'interaction de services dans les services web est réalisée sur une période de temps beaucoup plus longue que dans les autres plateformes présentées dans cette étude. Ceci, plus le fait que UDDI soit très récent (les premiers tests de UDDI ont été réalises fin 2000), sont quelques unes des raisons qui font que la majorité

des organisations qui fournissent des services web aujourd'hui le fassent sans passer par le registre [Ste03].

Caractéristiques

Description : Une description de service contient les informations suivantes (voir figure 4.7.a) :

`definitions` : nom du service et espace de nommage

`types` : description des types de données complexes

`message` : description d'un message (requête ou réponse). La description contient le nom du message et zéro ou plus `parts` qui décrivent les paramètres ou les valeurs de retour

`portType` : description d'une méthode formée par un groupe de messages, par exemple une méthode combinant requête et réponse.

`binding` : description sur le protocole de transmission des messages

`service` : emplacement du service (normalement une URL).

Publication : Les méthodes permettant de publier des informations dans le registre sont les suivantes :

`save_service` : publie un type de service.

`save_business` : publie un fournisseur de services.

Les méthodes permettant de retirer des informations du registre sont les suivantes :

`delete_service` : retire un type de service du registre.

`delete_business` : retire un fournisseur de services du registre.

Découverte : Le registre UDDI fournit diverses opérations de découverte :

`find_business` : retourne de l'information sur une ou plusieurs fournisseurs de services.

`find_service` : retourne de l'information sur des services fournis par des entreprises enregistrées.

Les méthodes de découverte permettent de faire des recherches sur des informations approximatives à travers des expression régulières. Les méthodes de découverte retournent des clés qui sont ensuite employées pour obtenir plus d'informations. Les méthodes de découverte peuvent retourner les résultats de façon ordonnée suivant plusieurs critères (ordre alphabétique, date d'enregistrement, présence de certificat, etc...).

Politiques de création : Un fournisseur de services peut supporter toutes les politiques de création de services, cependant la politique de création d'un seul objet partagé n'est pas réellement envisageable en raison du nombre de clients que peut avoir un service qui est publié à l'échelle mondiale.

a) Description

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace=
   "http ://www.foo.com/wsdl/PrinterService.wsdl" xmlns=...>
   <message name="PrintRequest">
      <part name="data" type="xsd :string"/>
   </message>
   <message name="PrintResponse">
      <part name="jobID" type="xsd :string"/>
   </message>
   <portType name="Print_PortType">
      <operation name="print">
         <input message="tns :PrintRequest"/>
         <output message="tns :PrintResponse"/>
      </operation>
   </portType>
   <binding name="Print_Binding"
            type="tns :Print_PortType">
     ...
   </binding>
   <service name="Print_Service">
      <documentation>WSDL pour service impression
             </documentation>
      <port binding="tns :Print_Binding" name="Print_Port">
         <soap :address
           location="http ://www.printhost :8080/printsvca"/>
      </port>
   </service>
</definitions>
```

b) Publication

```
save_service (exemple omis car trop encombrant)
```

c) Découverte et liaison

```
find_service (exemple omis car trop encombrant)
```

FIG. 4.7 – Exemple services web

Notifications : UDDI supporte l'enregistrement auprès du registre pour recevoir des notifications concernant des changements dans le registre (ajout, retrait et modification au niveau des services ou des entreprises).

Libération : Du fait que les services web supportent différents protocoles de communication, les deux politiques de libération, explicite ou expiration de bail, peuvent être implémentées.

Particularités : Système distribué, multi-registres, période d'exécution de l'interaction très longue.

Orchestration et Chorégraphie

La composition de services web (voir 4.2.5) est fondée sur deux activités différentes : l'orchestration et la chorégraphie. L'orchestration de services fait référence à l'activité de création de processus, exécutables ou non, qui utilisent des services web. La chorégraphie se concentre sur les séquences de messages échangés entre différents acteurs (typiquement l'échange public de messages entre des services web) plutôt qu'un processus spécifique exécuté par un acteur en particulier. Actuellement il existe différents standards permettant de réaliser l'orchestration et la chorégraphie, par exemple BPEL4WS, WSCI et BPML [Pel03].

L'orchestration de services est basée sur une technique de flot (workflow). Par exemple, BPEL4WS (Business Process Execution Language for Web Services) [CA02] inclut les concepts d'*activité de base* et d'*activité structurée*. Une activité de base représente une interaction avec un service web tandis qu'une activité structurée gère tout le flot du processus et spécifie à quel moment ont lieu les activités de base. L'activité structurée peut inclure des boucles ainsi que des ramifications, et permet de définir des variables qui peuvent contenir des valeurs retournées par un service web et qui peuvent ensuite être employées comme des paramètres dans l'appel d'un autre service web. BPEL4WS considère de plus la possibilité de créer des compositions de services hiérarchiques, c'est à dire d'utiliser le processus comme l'implémentation d'une interface de service. BPEL fournit en plus des mécanismes permettant de gérer les transactions et les exceptions pour supporter le fait qu'un service ne soit pas disponible au moment de son invocation.

La chorégraphie décrit les relations entre différents services. WSCI est un standard permettant de définir la chorégraphie, ou échange de messages, entre services web. WSCI décrit seulement le comportement qui peut être observé entre des services web, il n'adresse pas la définition de processus exécutables.

4.2.6 Synthèse

Cette section a présenté cinq plateformes à services en décrivant leurs domaine d'utilisation ainsi que la réalisation des caractéristiques présentée dans l'introduction de cette

section. Le tableau 4.1 résume les caractéristiques principales des plateformes décrites dans cette section.

4.3 Conclusion

Ce chapitre a présenté une étude sur l'approche à services qui est une approche orientée vers la création de connexions entre un client et un serveur lors de l'exécution du client et qui permet de découpler complètement le client d'un serveur en particulier. Un client connaît seulement le service, c'est à dire le comportement et la structure des fonctionnalités fournies par un serveur, mais ne connaît initialement ni le nom ni l'emplacement de celui-ci.

La deuxième partie de ce chapitre a été consacrée à la description et à la comparaison de cinq plateformes de services orientées à des domaines d'application très différents allant des applications centralisées avec support pour l'utilisateur, dans le cas du BeanContext, jusqu'aux systèmes distribués permettant à des applications hétérogènes de communiquer entre elles, dans le cas des services web. Cette étude permet d'apprécier la façon suivant laquelle une liste de caractéristiques déterminée dans la première partie du chapitre est réalisé dans les cinq plateformes. Il est intéressant de noter les variations sur la réalisation, par exemple au niveau du registre (parfois simple, parfois multiple) ou bien au niveau des mécanismes permettant de faire la sélection des fournisseurs de services au niveau du registre.

Pour conclure le chapitre, il est important de souligner le fait que l'approche à services se focalise sur l'interaction entre ses trois acteurs et ne traite pas de façon approfondie la problématique de la composition de services ou le déploiement des fournisseurs et demandeurs de services. Finalement, il est intéressant de réfléchir sur la limitation couramment imposée sur l'approche à services vis à vis de l'utilisation de l'interface de service comme un contrat de structure et de comportement ; il serait intéressant de trouver des moyens pour un demandeur de services de pouvoir décrire uniquement le comportement qu'il désire de la part d'un fournisseur. Cela reste cependant très difficile car il faudrait d'un côte pouvoir décrire le comportement du service et d'un autre être capable d'adapter le demandeur à n'importe quelle interface de service implémentées par les objets créés par le fournisseur de services.

Caractéristique \ Plate-forme	Courtier CORBA	BeanContext	Jini	OSGi	Services Web
Description de service	interface IDL + propriétés obligatoires et optionnelles	Interface ou classe Java	Interface Java + propriétés	Interface ou classe Java + propriétés	Description WSDL

Publication	`-export`
	`-withdraw`
		`-addService`
		`-revokeService`
			`-register`
			`-expiration`
			ou
			`lease.cancel`
				`-registerService`
				`-unregister`
					`-save_XX`
					`-delete_XX`

Découverte et type de filtrage	`-query`
	Langage de contraintes, organisation des résultats, politiques
		`-getService`
		Pas de filtrage
			`-lookup`
			Les propriétés de la demande doivent être présents dans la description
				`-getService References`
				`-getService`
				Filtre LDAP
					`-find_XX`

Politiques de création	Toutes (?)	Toutes	Un objet par liaison ou objet partagé (si service objet est une souche)	Objet partagé ou un objet par demandeur	Toutes
Notifications	Non (?)	Arrivée et départ de service	Arrivée, départ et modification	Arrivée, départ et modification	Arrivée, départ, modification
Libération	Explicite	Explicite	Expiration de bail	Explicite	Explicite ou expiration de bail
Type de système	Distribué	Centralisé	Distribué	Centralisé	Distribué
Nombre de Registres	Réseau de courtiers formant une fédération.	Un par contexte, mais hiérarchie de contextes possible	Multiples	Un seul	Multiples
Autres particularités	-	Un seul fournisseur d'un service peut être présent par contexte	Téléchargement des objets de service à distance	Support pour le déploiement des fournisseurs et demandeurs de services	Durée de l'interaction très longue.

TAB. 4.1 – Comparaison des plate-formes à services

Deuxième partie

Un Modèle à Composants Orienté Services

Chapitre 5

Principes d'un modèle à composants orienté services

Ce chapitre présente les principes d'une approche qui combine, d'un côté des concepts de l'approche à services qui sont introduits dans un modèle de composants, et d'un autre des mécanismes permettant aux applications de s'adapter, pour supporter la disponibilité dynamique des composants.

5.1 Introduction

La deuxième partie de ce document se focalise sur une proposition visant à introduire et à supporter la disponibilité dynamique dans un modèle à composants. La disponibilité dynamique de composants, qui est présentée dans la section 5.2.2, requiert qu'une application soit adaptée par rapport aux changements qui ont lieu au niveau des composants. La proposition présentée dans ce chapitre, appelée modèle à composants orienté services, est basée d'un côté sur une approche qui combine des aspects des approches à composants et à services, et d'un autre côté sur une logique chargée de réaliser l'adaptation, qui est contenue dans l'environnement d'exécution du modèle, et qui est configurée à partir d'informations contenues dans les composants. Les applications construites à partir de ce modèle sont capables de s'assembler et de s'adapter de façon dynamique.

Ce chapitre présente premièrement un ensemble de concepts relatifs à l'adaptation et à la disponibilité dynamique. Ces concepts permettent premièrement d'établir un vocabulaire utile au positionnement de l'approche et deuxièmement définissent précisément la disponibilité dynamique des composants. Ces concepts sont suivis par la présentation du modèle à composants orienté services.

5.2 Adaptation et disponibilité dynamiques

Cette section présente les concepts relatifs à l'adaptation dynamique ainsi qu'à la disponibilité dynamique.

5.2.1 Adaptation dynamique

L'*adaptation dynamique* fait référence à la possibilité d'adapter une application pendant l'exécution. Lorsque l'adaptation est réalisée en changeant l'architecture de l'application, cette architecture est qualifiée de dynamique [Ore96], et l'activité de modification de l'architecture est appelée la *reconfiguration dynamique* [Hof93]. Le mot architecture signifie ici une description de la façon suivant laquelle un système est composé à partir de ses composants[1] [SG96].

Une application est qualifiée comme étant *adaptative* lorsqu'elle fournit des moyens permettant à un acteur externe de réaliser des adaptations, tandis qu'elle est qualifiée comme étant *auto-adaptative* quand l'adaptation est réalisée de façon interne et qu'elle est déclenchée par des changements de l'état de l'application ou de l'environnement où elle s'exécute [DC01]. Ces différents concepts sont présentés à la suite.

5.2.1.1 Reconfiguration dynamique

Les concepts essentiels de la reconfiguration dynamique incluent des opérations primitives, l'existence d'une représentation architecturale, la préservation de la consistance et la participation des composants.

Opérations

Les changements pouvant être réalisées dans une architecture sont réalisés à partir d'un ensemble d'opérations primitives destinées à être appelées par l'acteur responsable de réaliser la reconfiguration. Ces opérations sont :

create Création d'un nouveau composant.

destroy Destruction d'un composant.

bind Création d'une liaison entre composants.

unbind Destruction d'une liaison entre composants.

set Configuration d'un composant.

move Re-localisation physique d'un composant.

Il faut noter que l'opération *move* est spécifique aux systèmes distribués.

[1] Le mot composant employé ici est employé dans un sens large.

Représentation architecturale

La réalisation de la reconfiguration dynamique est rendue possible par l'existence d'une représentation de l'architecture d'un système pendant son exécution [OT98]. L'existence d'une représentation de l'architecture permet, à l'acteur responsable de réaliser la reconfiguration, de connaître les entités sur lesquelles les opérations de reconfiguration sont réalisées. Les changements réalisés au niveau architectural sont répercutés au niveau du système en exécution, ce qui permet de maintenir les deux niveaux en correspondance [Mar01].

Préservation de la consistance

Un aspect essentiel dans la reconfiguration dynamique est de pouvoir assurer qu'une application reste consistante après une reconfiguration [DLB01]. Après une opération de reconfiguration, un système doit être dans un état 'correct', c'est à dire utilisable. Pour permettre ceci, trois aspects de la préservation de la consistance doivent être vérifiés [AA01] :

1. Le système satisfait les besoins d'*intégrité structurelle* : par exemple lors du remplacement d'un composant, le composant remplaçant doit implémenter la même interface que le composant qu'il remplace et être accessible aux clients du composant remplacé.
2. Les entités du système se trouvent dans des *états mutuellement consistants* : si une entité *A* interagit avec une entité *B* et que *B* est remplacé par *B'* après le début de l'interaction, soit l'interaction est avortée et *A* est prévenu, soit *B* termine l'interaction avant d'être remplacé, soit *B'* termine l'interaction qui avait été commencée par *B*.
3. Les *invariantes portant sur l'état* de l'application sont valables. L'état ne doit pas changer en raison d'une reconfiguration. Ceci est normalement vérifié à travers le transfert de l'état de composants qui sont remplacés.

Additionnellement, il peut être nécessaire de synchroniser les activités de reconfiguration de façon à éviter les blocages.

Participation des composants

Pour assurer les aspects relatifs à la préservation de la consistance, la reconfiguration dynamique est réalisée à partir de techniques qui permettent d'interrompre le flot de messages entrants et sortants d'un composant. Par exemple [KBC02] utilise des intermédiaires entre des instances de JavaBeans pour réaliser des substitutions d'instances. Bien que l'introduction d'intermédiaires permettant de bloquer les appels peut être réalisée de façon transparente, la réalisation des opérations *set, bind* et *unbind* peut nécessiter de la participation des composants d'une application.

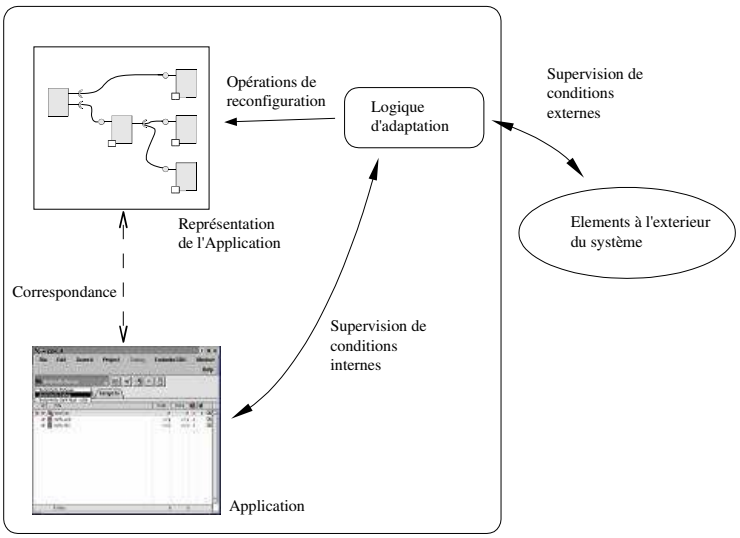


FIG. 5.1 – Logique d'adaptation

La *participation* d'un composant d'une application représente le fait que le composant est capable de répondre à certaines demandes nécessaires à la réalisation de la reconfiguration. Ces demandes incluent la divulgation de son état, l'initialisation, la réalisation de changements dans ses liaisons, et la suspension temporelle de son exécution [Hof93]. Dans la pratique, la participation de composants est réalisée à partir de méthodes de *contrôle*, qui sont destinées a être appelées par l'entité qui dirige les activités de configuration et de reconfiguration.

Un exemple de système supportant la reconfiguration dans lequel les composants participent est celui de [KC99] ; dans ce travail, un composant est défini comme un configurateur associé à un objet qui contient la logique applicative. Le configurateur fournit des opérations permettant au composant de participer dans les opérations de reconfiguration, notamment, la connexion et la dé-connexion. Dans ce travail, la reconfiguration est dirigée par un agent qui visite les différents composants et qui a une connaissance globale de la structure de l'application.

La participation d'un composant dans les activités de reconfiguration peut être rapprochée à la participation d'un composant dans le support des aspects non-fonctionnels. Par exemple, dans le modèle EJB (présenté dans la section 3.2.3), un composant qui gère la persistance lui même participe dans cette activité. Pour cela, il fournit une méthode permettant au conteneur de le prévenir du fait qu'il doit réaliser la persistance de son état à un moment donné.

5.2.1.2 Auto-adaptation

Une application auto-adaptative est capable de modifier son comportement en réponse à des changements dans son environnement d'opération. L'environnement d'opération fait référence à tout ce qui peut être observé par le système logiciel, depuis les entrées de la part d'un utilisateur jusqu'à l'information obtenue à partir de capteurs externes [OA99]. Lorsque l'auto-adaptation est réalisée à travers la reconfiguration dynamique, l'auto-adaptation résulte dans la modification de l'architecture d'une application par l'application même. L'auto-adaptation peut donner lieu à une *auto-réparation* [DVdHT02], un exemple de ceci est la création d'une liaison vers un composant qui substitue un autre qui devient défaillant.

La *logique d'adaptation* représente du code qui est chargé de réaliser la supervision ("monitoring") de conditions qui nécessitent de la réalisation d'une adaptation, et des activités de reconfiguration qui concrétisent cette adaptation. Alors que dans un système adaptatif, l'intelligence par rapport au moment et à la manière de réaliser la reconfiguration est fournie de façon externe et en temps d'exécution, dans un système auto-adaptatif, cette connaissance est fournie de façon interne et est définie au moment de la conception [DC01]. La figure 5.1 schématise un système auto-adaptatif au moment de l'exécution d'une application. La logique d'adaptation supervise des conditions à l'intérieur ou à l'extérieur du système et agit par rapport à celles-ci en réalisant des opérations de reconfiguration dynamique qui répercutent sur l'application en exécution.

Assemblage dynamique (ou auto-organisation)

La logique d'adaptation peut être soit centralisée soit disséminée dans tous les composants, dans ce dernier cas, l'architecture du système est capable de s'*assembler dynamiquement* (ou s'auto-organiser[2]). Dans une application pouvant s'assembler dynamiquement, les composants configurent automatiquement leur interaction d'une façon compatible avec une spécification globale [GMK02]. Dans ce type de systèmes, un composant doit créer des liaisons vers d'autres composants comme un résultat de ses propres actions.

Dans [Geo02], un mécanisme permettant à des systèmes distribués d'avoir des capacités d'auto-assemblage et d'auto-réparation est présenté. Dans ce travail, chaque composant est administré en temps d'exécution par un gestionnaire du composant appelé *component manager* qui contient la logique d'adaptation. Le component manager contient, en plus du composant qu'il administre, une vue de l'architecture courante du système et un ensemble de contraintes architecturales permettant de guider les changements qui peuvent avoir lieu au niveau des connexions entre le composant qu'il gère et d'autres composants. Lorsqu'un changement a lieu dans un composant, le gestionnaire du composant est noti-

[2]Dans cette thèse, ces deux termes sont considérés comme étant synonymes.

fié et agit en conséquence en reconnectant le composant qu'il administre par rapport aux contraintes et en mettant à jour sa vue de l'architecture. Ce travail se différentie de la proposition de cette thèse par le fait que la logique d'adaptation n'est pas générique et elle doit être générée manuellement à partir d'une spécification.

5.2.2 Disponibilité dynamique

Dans le cadre de cette thèse, la disponibilité dynamique concerne uniquement les composants[3]. La disponibilité dynamique est définie comme le fait qu'une instance ou une classe de composant puisse devenir indisponible ou disponible à tout moment pendant qu'une application qui l'utilise ou qui pourrait l'utiliser est en train d'être exécutée. Dans ce travail le terme *application* fait référence à un ensemble d'instances de composant qui, lors de l'exécution, sont connectées entre elles pour réaliser une tâche particulière.

La disponibilité dynamique est une situation qui peut avoir pour origine diverses causes dont :

- des changements dans la validité d'une instance. La validité représente le fait qu'une instance présente dans le système puisse s'exécuter et fournir ses fonctionnalités ou non, par rapport à l'état de ses dépendances. La validité des instances est discutée dans la section 5.4.3.1.
- la création ou destruction d'instances pendant l'exécution. Ces activités correspondent aux opérations create et destroy de la reconfiguration dynamique.
- l'introduction ou retrait de classes de composants en raison du *déploiement continu.*
- la dépendance d'un composant envers une entité physique qui est modifiée au moment où l'application s'exécute (par exemple un appareil connecté à la machine dans laquelle s'exécute l'application qui est déconnecté, ou bien une entité distante comme par exemple un service web qui ne répond pas).

Dans ce travail, seulement les trois premiers cas sont traités. Le dernier cas n'est pas traité, mais pourrait être modélisé à partir de dépendances d'implémentation décrites dans la section 5.4.1.

5.2.2.1 Déploiement continu

Le déploiement continu signifie que pendant l'exécution d'une application, les activités typiques du déploiement, telles que l'installation, retrait et mise à jour de classes de composants sont réalisées [Hal99]. Le déploiement continu est en relation avec les travaux traitant le *remplacement à chaud* ("hot swapping"), qui concerne le remplacement physique, ou mise à jour, de modules pendant l'exécution d'une application.

[3]La disponibilité dynamique pourrait concerner par exemple des connecteurs dans un modèle disposant de telles entités.

Opération	Acteur responsable	Résultat
validate	environnement d'exécution	valide une instance (voir 5.4.3.1)
invalidate	environnement d'exécution	invalide une instance (voir 5.4.3.1)
create	application / acteur externe	introduit une nouvelle instance de composant dans le système
destroy	application / acteur externe	retire une instance de composant du système
insert	acteur externe	introduit une classe de composant dans le système
remove	acteur externe	retire une classe de composant dans le système

TAB. 5.1 – Opérations et acteurs

Le remplacement à chaud est traité dans [HG98] à partir du concept de *classes dynamiques*. Une classe dynamique est une classe C++ qui peut être remplacée pendant l'exécution d'une application, le remplacement est rendu possible par l'utilisation d'intermédiaires qui représentent la classe et qui délèguent les appels vers une classe particulière. Au moment du remplacement, la classe référencée par l'intermédiaire est remplacée. Les objets existants ayant été créés au préalable ne sont pas touchés, tandis que la création de nouveaux objets est réalisée à partir la nouvelle classe. Les travaux qui traitent le remplacement à chaud se focalisent particulièrement sur la substitution et ne traitent pas la possibilité que de nouveaux composants puissent être incorporés ou que des composants qui sont utilisés deviennent indisponibles.

A différence de l'approche suivie dans les classes dynamiques, dans cette thèse, l'hypothèse est faite que lorsqu'une classe de composant devient indisponible, toutes ses instances doivent, elles aussi, devenir indisponibles. Cette hypothèse se justifie par le fait que la disponibilité d'une classe de composant peut dépendre d'une entité physique et donc toutes ses instances sont concernées par une modification dans cette dernière.

5.2.2.2 Opérations et acteurs

La table 5.1 présente les différentes causes de la disponibilité dynamique sous forme de noms d'opérations qui seront utilisés par la suite. Cette table présente aussi les acteurs responsables de ces opérations.

Une différence importante entre ce travail et les approches présentées auparavant est que la séparation entre le niveau classe et instance est prise en compte. Ceci justifie l'introduction de nouvelles opérations de reconfiguration dynamique par rapport à la liste présentée auparavant. Les opérations *insert* et *remove* représentent l'introduction et re-

trait de classes de composant tandis que *create* et *destroy* concernent spécifiquement la création et destruction d'instances. Les opérations *validate* et *invalidate*, qui seront présentées plus loin, sont propres à l'approche d'auto-adaptation présentée dans ce travail.

Dans le chapitre 6, le déploiement continu sera présenté comme étant à l'origine des opérations *insert*, *remove* ainsi que *create* et *destroy*, tandis que dans le chapitre 7, les opérations *create* et *destroy* seront aussi réalisées par les applications mêmes.

5.3 Principes du modèle

L'introduction de la disponibilité dynamique dans un modèle à composants est motivée par le fait qu'actuellement cette situation ne fait pas partie des hypothèses de l'approche à composants. Bien que l'introduction de la disponibilité dynamique ainsi que l'adaptation à celle-ci peuvent être réalisées dans les modèles existants à travers la programmation, ceci est une tâche complexe car elle requiert l'écriture de la logique d'adaptation en plus de la logique applicative.

Le modèle à composants orienté services est fondé sur deux principes qui permettent d'un côté d'introduire le concept de la disponibilité dynamique dans un modèle à composants et d'un autre permettre aux applications de s'adapter par rapport aux changements dans les composants. Ces deux principes sont décrits à la suite.

5.3.1 Introduction de concepts de l'approche à services dans un modèle à composants

L'étude réalisée dans le chapitre 3 révèle certains aspects de l'approche à composants qui représentent des limitations vis à vis de la possibilité de supporter la disponibilité dynamique :

- les applications sont assemblées en phase de conception, en réalisant une composition à partir de classes de composants disponibles au préalable. Cette situation est cependant indésirable face à la présence de la disponibilité dynamique, car celle-ci suppose l'introduction, pendant l'exécution, de classes de composants non disponibles en phase de conception (opération *insert*).
- la composition de composants réalisée de façon déclarative ou visuelle rend difficile l'expression de changements pouvant avoir lieu pendant l'exécution. Des exemples de ceci sont donnés par l'assemblage visuel de JavaBeans et l'Assembly dans CCM, dans ces deux cas, la composition donne lieu à une architecture statique.
- la composition impérative permet de réaliser des opérations de reconfiguration dynamique, telles que la création de nouvelles instances de composant ou de nouvelles connexions. Ces opérations sont cependant considérées dans le cadre de classes de

composants qui se trouvent déjà disponibles au moment où l'application est démarrée.
- Le retrait d'une classe de composant utilisée pendant l'exécution (opération *remove*) n'est pas une hypothèse de l'approche à composants.

D'un autre côté, dans l'approche à services, présentée dans le chapitre 4, offre certains aspects avantageux par rapport au support de la disponibilité dynamique :

- la composition de services n'est pas réalisée par rapport à des fournisseurs de service spécifiques, la composition de service est plutôt abstraite et ce n'est qu'au moment de l'exécution que des fournisseurs de services sont localisés et liés.
- l'interaction de l'approche à services supporte la substituabilité des fournisseurs.
- la disponibilité dynamique des fournisseurs de services est une hypothèse qui fait partie de l'approche à services : à tout moment un fournisseur de services peut arrêter de fournir ses services ou bien de nouveaux fournisseurs de services peuvent publier leurs services.

Le premier principe du modèle à composants orienté services est l'introduction de concepts de l'approche à services dans un modèle à composants. Dans le modèle à composants orienté services, les instances de composants deviennent des demandeurs et des fournisseurs de services, et les connexions entre les instances sont réalisées à partir d'une interaction propre à l'approche à services. Ceci permet de réaliser la découverte d'instances de composants à travers leurs services fournis en temps d'exécution. Le résultat de ceci est le concept de *composant à services*, qui sera présenté dans la section 5.4.1.

5.3.2 Extraction de la logique d'adaptation

La disponibilité dynamique des composants requiert qu'une application soit capable de s'adapter par rapport aux changements qui ont lieu au niveau des instances de composants pendant l'exécution. Ces tâches nécessitent l'écriture d'une logique d'adaptation chargée de réaliser l'adaptation. La programmation d'une telle logique d'adaptation est cependant une tâche complexe car elle nécessite de gérer les aspects concernant la supervision ainsi que la reconfiguration.

Le deuxième principe du modèle à composants orienté services est l'extraction de la logique d'adaptation de l'intérieur du code des composants. La logique est transférée vers une entité appelée *gestionnaire d'instance* qui est proche d'un conteneur (cette entité est décrite dans la section 5.4.3). Cette logique d'adaptation est configurée à partir d'informations attachées aux composants. L'avantage principal de cette approche est que la logique d'adaptation ne doit pas être programmée, ce qui permet de faire une séparation claire entre le code applicatif et le code d'adaptation. Cette approche est similaire à celle qui est utilisée dans les modèles à composants pour traiter les aspects non-fonctionnels. Du fait que les informations d'adaptation sont associées au niveau des composants équivaut à

avoir une logique d'adaptation disséminée dans tous les composants. Ceci donne aux applications construites à partir du modèle la capacité de s'assembler de façon dynamique, ceci est présenté dans la section 5.5.

5.3.3 Environnement d'exécution du modèle

Comme tout modèle à composants, le modèle à composants orienté service est associé à un environnement d'exécution qui est indépendant de toute application. L'environnement d'exécution est constitué par le gestionnaire d'instance et une infrastructure sous-jacente qui contient les éléments d'une plate-forme de services, notamment un registre de services et un mécanisme de notification de changements dans le registre. De plus, l'infrastructure sous-jacente doit être capable de supporter les opérations *insert* et *remove*, qui se traduisent par l'introduction ou le retrait de classes de composant dans le système.

Dans le modèle à composants orienté services, la logique d'adaptation, qui est contenue dans le gestionnaire d'instances, supervise des changements dûs aux opérations décrites dans la table 5.1. De façon concrète, ces opérations se traduisent par des changements au niveau du registre de services de l'infrastructure. Comme résultat de la supervision, la logique d'adaptation reconfigure une application à partir des opérations *set*, *bind* et *unbind*, ceci est présenté dans la section 5.4.3.

5.4 Éléments du modèle

Cette section présente les éléments du modèle à composants orienté services. Ces éléments sont présentés à partir des trois niveaux propres à un modèle à composants : le composant à services, qui correspond à la classe de composant, le paquetage de composant et l'instance de composant.

5.4.1 Composant à services

Le concept principal du modèle à composants orienté services est celui de classe de *composant à services*, dont une représentation schématique est présentée dans la figure 5.2. Un composant à services[4] reprend les concepts essentiels contenus dans une classe de composants, et qui ont été exposés dans le chapitre 3, avec les différences suivantes :

- Les interfaces fonctionnelles fournies correspondent à des interfaces de services fournies.
- Des propriétés de service sont associées à l'implémentation du composant.

[4]Lorsque "composant à services" est utilisé tout seul, il fait référence à la classe de composants à services.

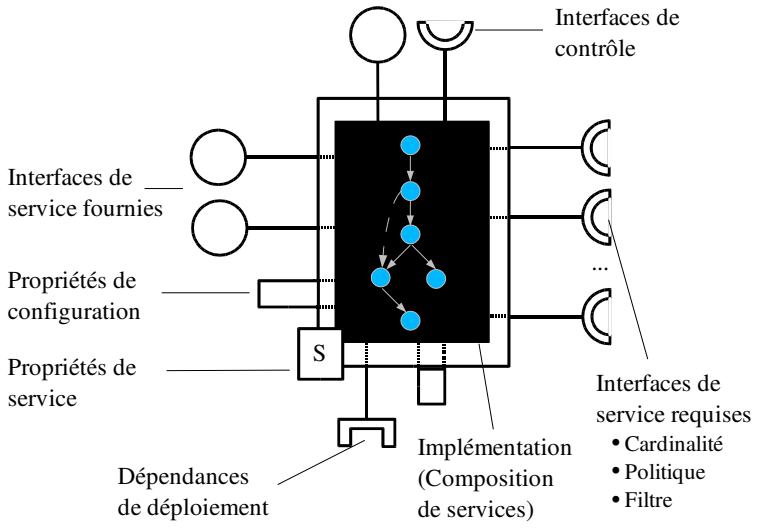


FIG. 5.2 – Classe de composant à services

– L'implémentation du composant à services peut réaliser une composition de services. Dans ce cas, le composant à services requiert des interfaces de service ; ces interfaces de service sont caractérisées par un certain nombre d'informations destinées à configurer la logique d'adaptation.

5.4.1.1 Interfaces de services fournies et propriétés de service

Un composant à services déclare un nombre variable d'interfaces de service qui sont fournies et implémentées par les instances du composant à services. Une instance de composant à services est qualifiée par rapport aux concepts présentés dans la section 4.1.4.2 comme étant un fournisseur de service qui suit une politique de création d'objets de service d'objet partagé. Dans ce cas, l'objet de service est l'instance de composant même. Plus simplement, ceci signifie que chaque instance de composant à services est enregistrée dans le registre de services, et qu'au moment où une liaison est réalisée, un demandeur de service reçoit une référence vers l'instance de composant. L'enregistrement des services est réalisé à partir de l'interface de service ainsi que des propriétés de service associées au composant à services.

Un composant à services peut ne pas déclarer d'interfaces de service fournies, dans ce cas, les instances d'un tel composant ne fournissent pas de moyens d'interaction aux autres instances du système. Lorsque plusieurs interfaces de service sont déclarées, une instance du composant à service est enregistrée autant de fois que le nombre d'interfaces de services fournies. Une instance de composant qui fournit plusieurs interfaces de service

fournit des services qui peuvent être utilisés de façon indépendante les uns des autres.

5.4.1.2 Interfaces de services requises

Un composant à services déclare des interfaces de service requises lorsqu'à l'intérieur de son implémentation il réalise une composition de services ; ceci signifie qu'une instance du composant utilise des services fournis par d'autres instances de composants. A l'intérieur de l'implémentation, la composition est une composition de services car elle n'est réalisée qu'en terme d'interfaces de services et non des instances fournissant des services. La composition de services est réalisée selon le principe décrit dans la section 4.1.5.

Une interface de service requise donne lieu, au moment de l'exécution, à un nombre variable de connexions entre instances de composants et ces connexions peuvent éventuellement changer au cours de l'exécution. La quantité de connexions qui sont réalisées ainsi que la spécification de leur comportement pendant l'exécution sont contraints par un certain nombre d'informations qui sont associés à chaque interface de service requise. Ces informations sont :

cardinalité la cardinalité permet d'exprimer le caractère *optionnel* ou *obligatoire* de la création d'une connexion avec un fournisseur de l'interface de service requise ainsi que la multiplicité, c'est à dire si une ou plusieurs connexions peuvent être créées (création *simple* ou *multiple*).

politique la politique définit si une fois que des connexions ont été créées, elles peuvent être changées (*dynamique*) ou non (*statique*).

filtre le filtre permet de contraindre la réalisation de connexions vers un sous-ensemble de fournisseurs de services ou bien vers un fournisseur de services particulier, pour réduire le problème de l'*imprévisibilité*. Le filtre est décrit en terme des propriétés de service associées aux composants. L'utilité du filtre sera décrite dans la section 5.5.7.

Les informations associées aux interfaces de service requises configurent la logique d'adaptation et permettent de réaliser l'assemblage et l'adaptation dynamiques des applications à travers la création et le changement de connexions. Finalement, il faut noter qu'alors que l'implémentation d'un composant à services contient une composition de services, l'assemblage d'une application est réalisé comme une composition d'instances de composants à services, ceci sera décrit dans la section 5.5.1.

5.4.1.3 Autres éléments

Les concepts de propriétés de configuration, d'interface de contrôle et les propriétés et dépendances de déploiement sont équivalents à ceux d'un composant 'classique'.

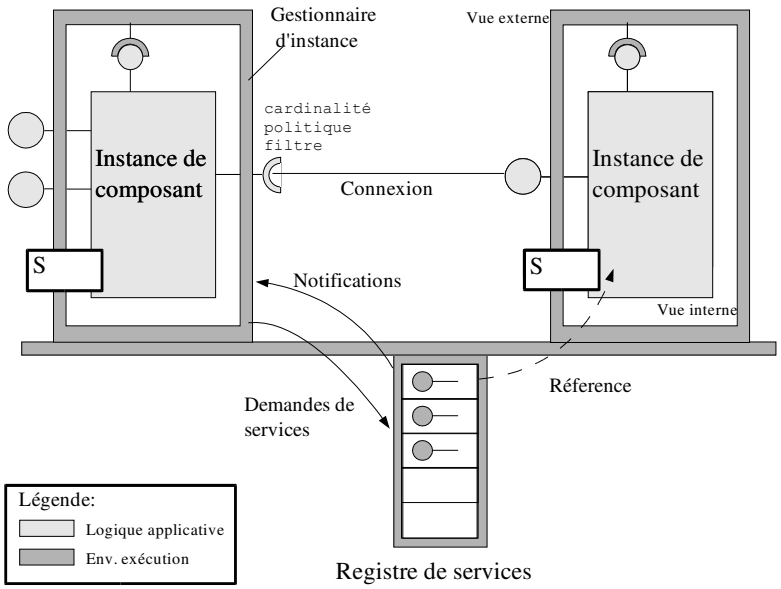


FIG. 5.3 – Instances de composants à services et gestionnaires d'instances

Les interfaces de contrôle permettent notamment aux instances de composants à services de participer dans les activités de reconfiguration dynamique. Les dépendances de déploiement représentent des dépendances envers des ressources telles que des librairies qui sont gérées par l'environnement d'exécution. Les propriétés de configuration et de déploiement sont présentées dans le schéma mais ne sont pas traitées dans ce chapitre, les propriétés de configuration seront discutées dans la section 6.4.3.2.

5.4.2 Paquetage de composant

Un composant à services est livré dans un paquetage de composants. Ce paquetage de composants contient d'un côté l'implémentation du composant à services ainsi que des ressources de ce composant. Ces ressources incluent des informations concernant les éléments présents sur la vue externe du composant à services (les éléments de la vue externe sont décrits plus loin), et qui permettent à un acteur de réaliser les opérations *create* et *insert*, qui nécessitent la connaissance des services fournis et requis par les instances de composants.

5.4.3 Instance de composant à services

Comme dans l'approche à composants classique, une application est construite à partir d'un ensemble d'instances de composant à services qui sont connectées entre elles. Dans le modèle à composants orienté services, le cycle de vie de chaque instance de composant

est géré de façon indépendante par un *gestionnaire d'instance*, qui est représenté dans la figure 5.3.

Chaque gestionnaire d'instance travaille de façon indépendante et est chargé de gérer une seule instance de composant. Le gestionnaire d'instance est comparable au conteneur de l'approche à composants dans le sens qu'il contrôle une instance de composant à services à partir d'appels vers des méthodes des interfaces de contrôle. Dans un composant à services, la vue externe est constituée par les interfaces de service fournies et requises ainsi que les propriétés de service, et la vue interne est constituée par les interfaces de contrôle ainsi que les dépendances d'implémentation. Le gestionnaire d'instances a accès à la vue interne de l'instance du composant qu'il gère tandis que les autres instances ne sont vues qu'à partir des services qu'elles fournissent.

Les responsabilités principales du gestionnaire d'instance sont :

1. gérer les taches concernant l'enregistrement des services fournis par l'instance de composant gérée, lorsque celle-ci devient valide.
2. superviser les changements dans les registres de services et agir par rapport à ceux ci à travers la création et le changement de connexions (opérations *bind* et *unbind*) qui existent entre l'instance de composant gérée, si elle requiert des services, et d'autres instances qui fournissent les services requis.
3. maintenir la *validité* de l'instance.

La création des connexions est réalisée par le gestionnaire de l'instance suivant l'interaction de l'approche à services (découverte, sélection, liaison). Le gestionnaire d'instance accède au registre de services et découvre des instances de composant qui fournissent un service requis par l'instance gérée par le gestionnaire. Lorsque des instances fournissant le service recherché sont localisées, le gestionnaire d'instance lie l'instance qu'il administre avec les instances qui fournissent des services requis par l'instance administrée. Pour gérer les changements dans les connexions, le gestionnaire d'instances est à l'écoute des notifications produites par le registre de services, et qui concernent l'arrivée ou le départ de services (ceci est représenté par les flèches qui vont du registre vers le gestionnaire dans la figure 5.3).

5.4.3.1 Étapes du cycle de vie

La création et destruction d'une instance de composant est accompagnée par la création et destruction d'un gestionnaire de cette instance. Une instance de composant à services peut se trouver dans deux états différents : *valide* ou *invalide.* Lorsque l'instance est valide, ses services sont enregistrés et sa logique applicative est en exécution ; au contraire, lorsque l'instance est invalide, les services de l'instance ne sont pas enregistrés et la logique applicative n'est pas en exécution. La validité est définie par rapport aux interfaces de service requises par l'instance ; lorsque des connexions sont associées aux

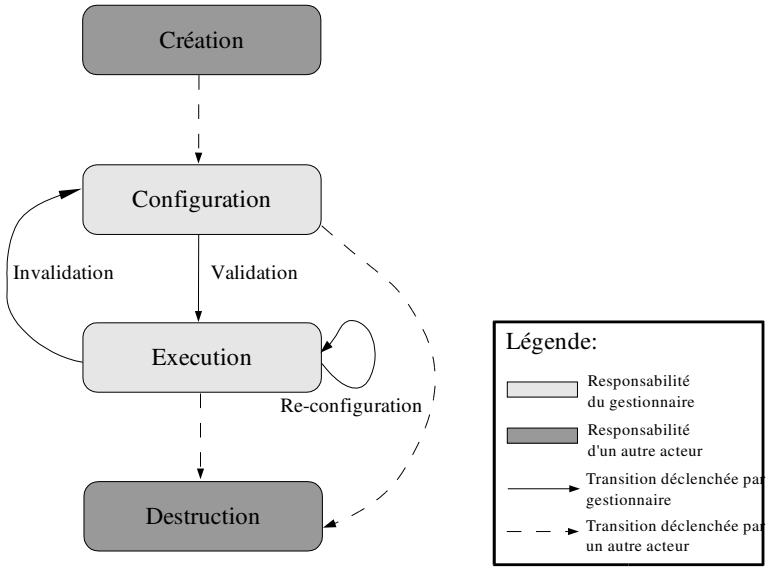


FIG. 5.4 – Étapes du cycle de vie

interfaces requises qui sont caractérisées comme étant obligatoires, l'instance peut être validée.

Au moment de la création, une instance est invalide. Le gestionnaire commence alors à réaliser la gestion du cycle de vie de l'instance, qui est présenté dans la figure 5.4. Les étapes de ce cycle qui sont la responsabilité du gestionnaire d'instance sont :

configuration dans cette étape, le gestionnaire cherche à créer des connexions entre l'instance qu'il administre et des instances qui fournissent des services requis. Si cette activité échoue, le gestionnaire d'instance reste à l'écoute de notifications d'arrivée de services et tente de réaliser à nouveau la configuration de l'instance dès que possible. Une fois que la configuration de l'instance administrée est terminée, le gestionnaire valide cette instance en la notifiant du fait que l'exécution de sa logique applicative peut démarrer. Après ceci, le gestionnaire d'instance enregistre les services que l'instance fournit dans le registre, l'instance est alors valide.

exécution pendant l'exécution, le gestionnaire peut recevoir des notifications concernant des changements au niveau des services, le gestionnaire peut alors procéder à réaliser une reconfiguration. Lors de celle-ci, le gestionnaire réalise des changements au niveau des connexions de l'instance. Dans le cas où la reconfiguration est réalisée avec succès, l'exécution continue normalement, et d'autres reconfigurations peuvent avoir lieu. L'invalidation est causée par l'échec d'une reconfiguration. A ce moment là, le gestionnaire d'instance re-

tire les services fournis par l'instance administrée du registre de services. Ensuite, le gestionnaire prévient son instance du fait qu'elle doit arrêter l'exécution de sa logique applicative. L'instance est alors invalide et le gestionnaire entre dans un nouveau cycle dans lequel il retourne à l'étape de configuration.

Le fait que l'invalidation mène à nouveau à la configuration représente le fait que le gestionnaire d'instance tente de maintenir une instance à l'état valide jusqu'à sa destruction. L'intérêt de ce cycle est montré dans la section 5.5.6.

Le succès de la configuration et de la reconfiguration dépendent des informations associées aux interfaces requises et aux services des instances qui sont enregistrées à un moment donné dans le registre de services. Ainsi, un composant sans interfaces requises ou dont les interfaces requises sont optionnelles sera toujours configuré correctement.

La destruction d'une instance peut avoir lieu au moment où une instance est invalide et que le gestionnaire tente de la configurer ou bien au moment où elle est valide et le gestionnaire est en attente de notifications pour réaliser des reconfigurations. Il faut cependant noter que la destruction d'une instance de composant valide implique son invalidation.

5.4.3.2 Algorithme de configuration

Un pseudocode représentant l'algorithme utilisé par le gestionnaire d'instance lors la configuration est présenté dans la figure 5.5. Cet algorithme montre que la configuration d'une instance peut échouer si le composant possède des interfaces requises obligatoires et qu'au moment de la configuration, le gestionnaire de l'instance ne peut trouver des instances fournissant les services requis. Cet algorithme montre aussi que du fait de l'utilisation de l'approche à services, le gestionnaire de l'instance affronte la problématique du filtrage côté client (décrite dans la section 4.1.6.1), ceci a lieu au moment ou le gestionnaire doit lier une interface requise simple et que le registre retourne plusieurs réponses. La gestion du filtrage côté client est représentée par l'appel de la méthode `choisirFournisseur`.

5.4.3.3 Algorithme de réception de notifications

Un pseudocode représentant l'algorithme utilisé par le gestionnaire d'instance au moment de la réception de notifications de la part du registre de services est présenté dans la figure 5.6. Cet algorithme peut invoquer celui de configuration si au moment de l'arrivée d'un nouveau service, l'instance qui est gérée par le gestionnaire n'a pas encore été configurée (si au moment de l'introduction de l'instance dans le gestionnaire la configuration a échoué).

L'algorithme présenté montre qu'avant et après la reconfiguration, l'instance qui est contrôlée par le gestionnaire est notifiée du fait qu'un changement va avoir lieu ou que les changements sont terminés. Cet algorithme montre aussi que lorsqu'une connexion

```
//*************************************
//* Fonction qui réalise la configuration
//*
//* entrée : instance (invalide)
//* sortie : instance (valide ou invalide)
//*
//*************************************
fonction configuration(instance)
  pourchaque(instance.interfaceRequises)
    interfaceRequise=instance.interfaceRequises.suivant();
    // découverte des fournisseurs
    fournisseurs=registre.decouvrirService
                  (interfaceRequise.nomService,
                   interfaceRequise.filtre);
    si(fournisseurs.taille==0 &&
            interfaceRequise.typeCadinalite==’obligatoire’)
      retour();  // echec
    si(fournisseurs.taille>0)
      si(interfaceRequise.multCardinalite==’simple’)
        fournisseur=choisirFournisseur(fournisseurs);
        instance.lierAvec(fournisseur);
        interfaceRequise.connexions.ajouter(fournisseur);
      sinon // cardinalité multiple
        pourchaque(fournisseurs)
          fournisseur=fournisseurs.suivant();
          instance.lierAvec(fournisseur);
          interfaceRequise.connexions.ajouter(fournisseur);
        finpour
  finpour
  validerInstance();
finfonction
```

FIG. 5.5 – Algorithme de configuration

```
//*************************************************
//* Fonction qui réagit par rapport à une
//* notification
//*
//* entrées : notification (du registre)
//*          instance     (valide ou invalide)
//*          interfaceRequise (interface
//*               requise associée à la notification)
//* sorties : instance (valide ou invalide)
//*
//*
//*************************************************
fonction recoitNotification(notification, instance,
                                          interfaceRequise)
si(notification==’service_retiré’)
  si(instance.valide==vrai &&
     interfaceRequise.connexions.contient(notification.fournisseur))
    si(interfaceRequise.politique==’statique’)
      instance.invalider(); // pas de changements permis
      instance.terminer(); // car changement dans connexion statique
      retour;
    sinon // dynamique
      instance.debutReconfiguration();
      interfaceRequise.detruireConnexion(notification.fournisseur);
      si(interfaceRequise.connexions.taille==0) // faut il chercher substitut?
        si(interfaceRequise.typeCardinalité==’obligatoire’)
          fournisseur=chercherAutreFournisseur();
          si(fournisseur==null)
            instance.invalider(); // echec car une connexion est requise
            retour;
          sinon
            instance.lierAvec(fournisseur);
            interfaceRequise.connexions.ajouter(fournisseur);
            instance.finReconfiguration();
sinon // arrivée d’un nouveau service
  si(instance.valide==faux)
    configuration(instance); // l’instance n’est pas encore valide
    retour;
  sinon
    si(interfaceRequise.politique==’dynamique’)
      si(interfaceRequise.multCardinalite==’multiple’ ||
         interfaceRequise.connexions.taille==0)
        instance.debutReconfiguration();
        instance.lierAvec(notification.fournisseur);
        interfaceRequise.connexions.ajouter(notification.fournisseur);
        instance.finReconfiguration();
finfonction
```

FIG. 5.6 – Algorithme de reception de notifications

statique change, l'instance est non seulement invalidée mais aussi notifiée (`terminer`) du fait que l'invalidation est le résultat d'un changement dans une connexion statique.

Il faut noter que les interfaces requises caractérisées comme étant simples, obligatoires et dynamiques permettent de supporter la réalisation de substitutions. En effet, lorsqu'une connexion associée à une interface requise ayant ces caractéristiques est détruite, le gestionnaire de l'instance cherche un substitut (`chercherAutreFournisseur`) au lieu de procéder à l'invalidation de l'instance gérée. Le fait que la substitution soit réalisée par rapport à une interface de service et que les instances soient prévenues au moment de la reconfiguration permettent de supporter les deux premiers points liés à la préservation de la consistance décrits dans la section 5.2.1.1.

La table 5.2 décrit le comportement du gestionnaire d'instance vis à vis des étapes de configuration et de reconfiguration par rapport aux informations définies au niveau des interfaces requises.

5.5 Assemblage dynamique et auto-adaptation

La logique d'adaptation contenue dans les gestionnaires d'instances résulte dans le fait que l'assemblage dynamique d'une application commence au moment où des instances de composant sont validées.

5.5.1 Composition de composants

Une composition de composants est réalisée, au moment de l'exécution, à partir de la création d'un ensemble d'instances de composants dont les services fournis et requis donnent lieu à des connexions. Une composition peut résulter dans une application, et ceci a lieu typiquement lorsqu'il existe un composant 'principal', qui est responsable de diriger l'exécution de l'application.

Il est important de noter qu'il existe deux approches par rapport à la construction d'une application. Dans la première, appelée *noyau applicatif extensible*, le composant principal requiert des services fournis par d'autres composants (il peut éventuellement ne pas fournir de services lui même). Dans la deuxième, appelée *noyau applicatif fournisseur*, le composant principal fournit un ou plusieurs services qui sont utilisés par d'autres composants. Dans le premier cas, le noyau applicatif s'adapte par rapport à la présence des composants qu'il requiert, tandis que dans le deuxième cas, ce sont les composants qui s'adaptent par rapport à la présence du noyau.

Un exemple d'application à base de composants à services est montré dans la figure 5.7.a. Cette application, qui représente un éditeur de texte, est construite à partir d'une instance d'un composant principal (WordProcessor Component), qui a deux interfaces de ser-

Cardinalité, Politique	Comportement
`Obligatoire, Simple, Statique`	L'interface requise représente une seule connexion qui doit pouvoir être créée au moment de la la configuration de l'instance. Tout changement dans cette connexion invalide l'instance.
`Obligatoire, Simple, Dynamique`	L'interface requise représente une seule connexion qui doit pouvoir être créée au moment la configuration de l'instance. Si la connexion est détruite en raison d'un changement de l'instance vers laquelle elle est dirigée, l'environnement d'exécution cherche à créer une connexion de remplacement. Si la connexion ne peut être créée, l'instance du composant qui déclare la dépendance est invalidée.
`Obligatoire, Multiple, Statique`	L'interface requise représente un ensemble de connexions et au moins une connexion doit pouvoir être créée au moment de la configuration de l'instance. Une fois que l'instance est active, les connexions ne peuvent plus changer. Tout changement dans une connexion entraîne l'invalidation de l'instance.
`Obligatoire, Multiple, Dynamique`	L'interface requise représente un ensemble de connexions et au moins une connexion doit pouvoir être créée au moment de la configuration de l'instance. Des changements dans les connexions peuvent avoir lieu pendant l'exécution : des nouvelles connexions peuvent être créées et des connexions existantes peuvent être détruites du moment qu'il existe au moins une connexion. Si les changements résultent dans la destruction de toutes les connexions l'instance est invalidée.
`Optionnelle, Simple, Statique`	L'interface requise représente une connexion optionnelle. La connexion peut être créée lors de l'étape de configuration de l'instance si un fournisseur de service approprié est présent. Une fois que l'instance est activée et si la connexion a été créée, tout changement dans la connexion invalide l'instance. Si aucune connexion n'a été créée lors de la configuration de l'instance, celle-ci n'est jamais invalidée (jusqu'à la destruction).
`Optionnelle, Simple, Dynamique`	L'interface requise représente une connexion optionnelle. La connexion peut être créée soit lors de l'étape de configuration de l'instance soit après que l'instance a été activée. La connexion peut être détruite puis recréée pendant l'exécution de l'instance. L'invalidation de l'instance n'a lieu que lors de la destruction.
`Optionnelle, Multiple, Statique`	L'interface requise représente un ensemble de connexions optionnelles. Les connexions peuvent être créées lors de l'étape de configuration de l'instance. Une fois que l'instance a été activée, tout changement dans les connexions invalide l'instance. Si aucune connexion n'a été créée lors de la configuration de l'instance, celle-ci n'est jamais invalidée à cause de changements dans les connexions, l'invalidation n'a lieu que jusqu'à la destruction.
`Optionnelle, Multiple, Dynamique`	L'interface requise représente un ensemble de connexions optionnelles. Les connexions sont créées lors de l'étape de configuration de l'instance ou bien une fois que l'instance a été activée. Dans ce cas l'instance n'est jamais invalidée en raison de changements dans les connexions, l'invalidation n'a lieu que jusqu'à la destruction.

TAB. 5.2 – Comportement selon les caractéristiques d'une interface requise

vice requises[5]. La première correspond à un service d'impression (optionnelle, simple et dynamique) et la deuxième à un service de correction orthographique (obligatoire, simple et dynamique). Dans cette application, le composant principal est un noyau applicatif extensible. De plus, le composant de correction orthographique a une interface de service requise correspondant à un service de dictionnaire (obligatoire, multiple et dynamique).

5.5.2 Ordre d'assemblage

Du fait que chaque instance de composant est gérée de façon indépendante, l'assemblage d'une application est réalisé de façon progressive, au fur et à mesure que les instances de composants sont validées par leurs gestionnaires respectifs.

La validation des instances a lieu dans l'ordre suivant :

1. Premièrement, les instances de composant qui ne requièrent pas de services, ou qui requièrent des services de façon optionnelle.
2. Deuxièmement, les instances de composant qui requièrent des interfaces de services obligatoires et qui sont fournies par les premières.

Il est important de noter que la présence de dépendances circulaires peut empêcher qu'une application soit assemblée, cette situation est discutée dans la section 5.5.7.

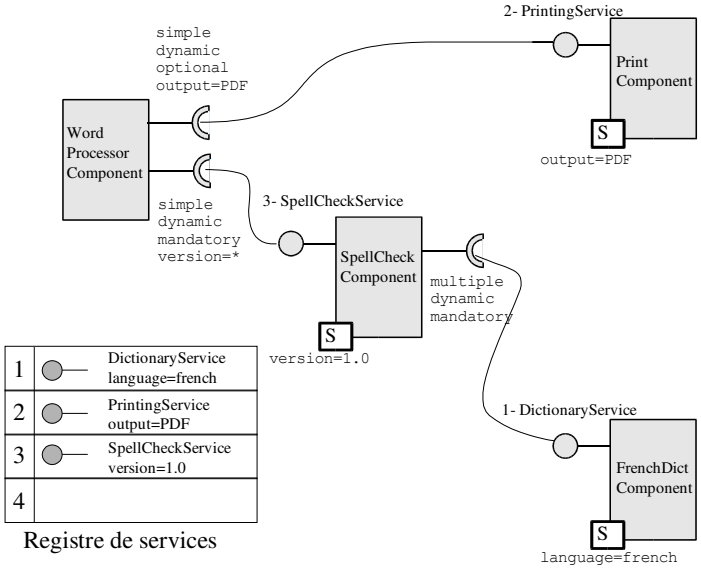
Dans l'exemple présenté dans la figure 5.7.a, en supposant que toutes les instances de composant sont créées de façon simultanée, ce seront d'abord les instances correspondant aux services de dictionnaire (DictionaryService) et au service d'impression (PrintingService) qui seront validées. Une fois que le service de dictionnaire devient disponible, l'instance du correcteur orthographique sera validée. Après que le service de correction orthographique devient disponible, l'éditeur de texte sera à son tour validé. Cet exemple permet d'apprécier comment l'application s'assemble dynamiquement au fur et à mesure de l'arrivée des services fournis par les instances.

5.5.3 Incorporation de nouvelles instances

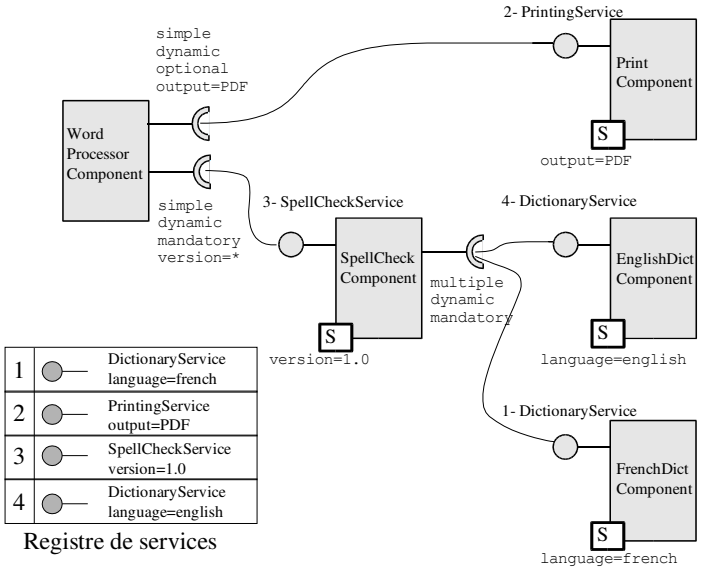
Au moment où de nouvelles instances deviennent disponibles pendant l'exécution d'une application, en raison de leur validation, les gestionnaires d'instances sont notifiés de l'arrivée de services qui peuvent être employés par les instances qu'ils gèrent. A ce moment là, des reconfigurations peuvent avoir lieu.

La figure 5.7.b présente la même application qu'avant au moment où une instance fournissant un service dictionnaire devient disponible. Au moment où ce deuxième service dictionnaire est enregistré, le gestionnaire de l'instance du correcteur orthographique

[5]Cette représentation montre les instances de composant uniquement à partir de leur vue externe



a) Application dans une configuration initiale



b) Après l'arrivée d'un nouveau composant dictionnaire

FIG. 5.7 – Assemblage et arrivée d'une instance de composant

reçoit une notification de l'arrivée d'un nouveau service. Du fait que l'interface de service requise par l'instance qu'il gère est concernée par ce service et qu'elle accepte des connexions multiples et dynamiques, le gestionnaire d'instance réalise une reconfiguration de son instance, dans ce cas, ceci résulte dans la création d'une nouvelle connexion vers l'instance qui fournit le deuxième service de dictionnaire.

5.5.4 Retrait d'instances utilisées

Le retrait d'une instance utilisée dans une application suit une procédure similaire à l'ajout d'une instance. Les gestionnaires des instances qui sont connectées à l'instance qui devient indisponible, car elle est invalidée, reçoivent une notification du retrait du service du registre de services. A ce moment là, les gestionnaires d'instance reconfigurent leurs instances par rapport aux caractéristiques définies au niveau des interfaces de service requis.

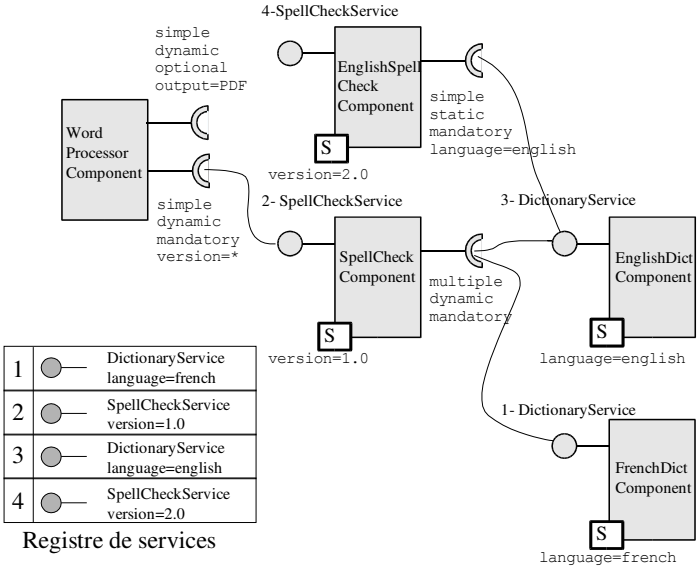
La figure 5.8.c présente un retrait d'instance au niveau du service d'impression. Du fait que l'interface de service requise par l'éditeur de texte est optionnelle et dynamique, la connexion peut être détruite sans que l'instance de l'éditeur de texte soit invalidée.

5.5.5 Substitution d'instances utilisées

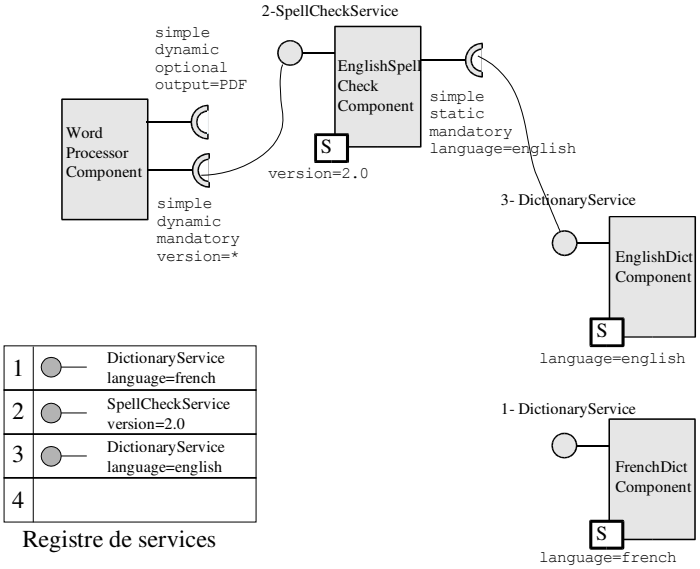
Comme il a été décrit précédemment, une interface requise caractérisée comme étant obligatoire, simple et dynamique permet de supporter la substitution pendant l'exécution. Dans l'exemple présenté dans la figure 5.8.d, une substitution est réalisée au niveau de l'instance du composant de correction orthographique. Cette substitution est possible du fait qu'au moment où l'instance du composant de correction orthographique est retirée, un autre service de correction orthographique est disponible (EnglishSpellCheckComponent). Les interfaces requises avec ces caractéristiques permettent de créer des applications capables d'auto-réparation à travers la substitution.

5.5.6 Invalidation d'une application par "réaction en chaîne"

Les changements qui ont lieu au niveau des instances de composants peuvent parfois résulter dans l'invalidation d'une application comme résultat d'une "réaction en chaîne". Ceci a lieu au moment où une instance de composant qui fournit un service indispensable devient indisponible et que le composant principal de l'application ne peut s'adapter à la modification. Cette situation pourrait se présenter dans l'application présentée auparavant si l'application se trouve dans l'état représente dans la figure 5.7.a et que l'instance du composant 'dictionnaire Français' est retirée. Ceci a pour conséquence que le correcteur orthographique soit invalidé ce qui entraîne à son tour l'invalidation de l'éditeur de textes.



c) Après le retrait du composant d'impression et arrivee d'un nouveau correcteur orthographique



d) Après une substitution du correcteur

FIG. 5.8 – Retrait et substitution d'une instance

Du fait que les gestionnaires d'instance cherchent à maintenir la validité des instances qu'ils gèrent, au moment où un nouveau dictionnaire devient disponible, le correcteur est validé à son tour ainsi que le composant principal. L'application peut alors redémarrer son exécution.

5.5.7 Limitations de l'assemblage dynamique

Le succès de l'assemblage dynamique d'une application ne peut être garanti que si l'acteur qui introduit les instances de composants dans le système a auparavant vérifié que les interfaces requises obligatoires des composants pourront être satisfaites. D'un autre côté, deux situations peuvent apparaître avec l'assemblage dynamique : l'*imprévisibilité* et les blocages.

5.5.7.1 Imprévisibilité

La technique de composition employée dans la construction d'applications avec le modèle à composants orienté services a pour conséquence qu'il puisse y avoir un certain degré d'imprévisibilité au moment où les connexions sont créées. L'imprévisibilité apparaît par exemple lorsqu'une instance a une interface de service requise simple et que deux instances fournissant le service requis sont présentes au moment de l'assemblage, car il n'est pas possible de connaître en avance vers quelle instance fournissant le service la connexion sera réalisée. Si les deux instances peuvent être utilisées par la première de façon indifférente, ceci ne pose pas de problème, cependant dans le cas où une liaison particulière doit être réalisée, il est nécessaire de spécifier un filtre dans l'interface requise. Ce filtre est décrit par rapport aux propriétés de l'instance vers laquelle la connexion doit être réalisée, et ceci permet de réduire l'imprévisibilité. L'introduction de filtres a pour désavantage de limiter les possibilités d'adaptation.

5.5.7.2 Blocage

Du fait que les gestionnaires d'instance ne possèdent qu'une information localisée qui concerne l'instance qu'ils gèrent, il est possible que des dépendances circulaires puissent donner lieu à des blocages. Une dépendance circulaire apparaît lorsqu'un composant requiert, de façon obligatoire, un service qui est fourni par un autre composant qui requiert lui même, de façon obligatoire, un service fourni par le premier composant. Cette dépendance circulaire résulte dans le fait qu'aucune des deux instances ne peut être validée, car les deux gestionnaires d'instance restent en attente de l'arrivée d'un service.

De façon simple, le blocage peut être résolu en modifiant un des composants pour que l'interface de service requise ne soit pas obligatoire mais optionnelle et dynamique. Ceci permettrait à l'instance de ce composant d'être validée, ce qui à son tour entraînerait la validation de l'instance de l'autre composant. Lorsque cette dernière a lieu, le premier

composant serait connecté au deuxième. Une solution plus générale nécessiterait d'un moyen de détecter des cycles pendant la configuration.

5.6 Conclusion

Ce chapitre a introduit les concepts d'une approche permettant d'introduire la disponibilité dynamique dans un modèle à composants et de permettre aux applications de s'adapter face à cette situation. Ceci est réalisé à partir d'un modèle à composants orienté services qui est un modèle à composants dans lequel des concepts de l'approche à services sont introduits ainsi que des mécanismes permettant de séparer la logique d'adaptation du code applicatif. Du fait que chaque instance de composant est gérée de façon indépendance à partir d'informations disséminées dans chaque composant, les applications sont auto-assemblées et auto-adaptatives.

Les chapitres suivants se focalisent dans la réalisation du modèle et la construction d'applications. Dans le chapitre suivant, la création et destruction d'instances résulte des activités de déploiement continu. Dans le chapitre 7, la création et destruction d'instances est réalisée en dehors des activités de déploiement.

Chapitre 6

Applications à base d'instances de déploiement

Ce chapitre présente le concept d'instance de déploiement, la réalisation du modèle à composants orienté services au dessus de la plate-forme de services OSGi, ainsi que des évaluations dont des applications construites à partir d'instances de déploiement.

6.1 Introduction

Ce chapitre présente le concept d'instance de déploiement ainsi que des aspects relatifs à l'implémentation de l'environnement d'exécution du modèle à composants orienté services au dessus de la plate-forme de services OSGi (qui a été présentée dans la section 4.2.4).

La construction d'applications à base d'instances de déploiement permet de créer des applications compatibles avec la plate-forme de services OSGi. De plus, l'utilisation du modèle à composants orienté services simplifie considérablement la construction d'applications dans cette plate-forme. Des exemples d'applications construites à base d'instances de déploiement sont décrits à partir d'évaluations industrielles dans lesquelles l'implémentation du modèle, appelée le ServiceBinder, a été utilisée.

Ce chapitre présente d'abord le concept d'instance de déploiement suivi des des aspects relatifs à l'implémentation. Ceci est suivi par une section d'évaluations qui inclut la description des applications réalisées dans le cadre d'évaluations industrielles ainsi qu'un comparatif avec la plate-forme OSGi.

6.2 Instances de déploiement

Une *instance de déploiement* représente une instance de composant à services dont la création et la destruction (opérations *create* et *destroy*) sont liées à l'introduction et au

retrait de composants à service dans le système (opérations *insert* et *remove* de la section 5.2.2.2). Une instance de déploiement est une instance de composant à services 'singleton', car une seule instance est créée par composant à services. Ces instances sont appelées de déploiement car l'introduction et retrait de composants à service sont le résultat des activités de déploiement continu (voir 5.2.2).

6.2.1 Applications à base d'instances de déploiement

Le concept d'instance de déploiement permet de construire des applications dans lesquelles la création d'instances multiples pendant l'exécution n'est pas nécessaire, et qui peuvent évoluer à partir du déploiement continu. Ceci est précisément le cas des applications construites avec la plate-forme de services OSGi, sur laquelle est réalisé le modèle. Des exemples de ces applications sont présentés dans la section 6.4.1.

6.2.2 Acteur responsable de la création

L'acteur responsable de la création et destruction des instances de déploiement est un acteur qui réalise le déploiement continu. Dans l'implémentation du modèle, cet acteur interagit avec des mécanismes fournis par la plate-forme de services OSGi et qui dirigent le cycle de vie des paquetages de composants à services, ceci est présenté dans la section 6.3.2.

6.3 Réalisation du modèle

Cette section présente la réalisation du modèle à composants orienté services au dessus de la plate-forme de services OSGi.

6.3.1 Descripteur de Composants

Un composant à services est déclaré dans un descripteur basé sur une syntaxe XML. La DTD correspondante au descripteur de composant à services ainsi qu'un exemple de descripteur sont présentés dans la figure 6.1. Un descripteur peut contenir plusieurs composants à services qui sont délimités par la balise `bundle.`

L'exemple présenté montre un composant à services fournissant un service de correction orthographique (`SpellCheckService`) qui possède une dépendance envers des services de dictionnaire (`DictionaryService`). Ce composant correspond à l'exemple donné dans le chapitre précédent, dans la section 5.5.

Les différents éléments présents au niveau de la DTD correspondent aux éléments présents sur la vue externe d'un composant à services (ces éléments ont été présentés graphiquement dans la figure 5.2). Ces éléments sont contenus à l'intérieur de la balise

DTD

```
<!ELEMENT bundle (component*)
>
<!ELEMENT component (provides*,property*,requires*)>
<!ATTLIST  component
  implementation CDATA #REQUIRED
>
<!-- interface de services fournie -->
<!ELEMENT provides EMPTY>
<!ATTLIST  provides
  service CDATA #REQUIRED -- nom d'interface Java --
>
<!-- propriété de service -->
<!ELEMENT property EMPTY>
<!ATTLIST  property
  name CDATA #REQUIRED
  value CDATA #REQUIRED
  type CDATA #REQUIRED -- types simples Java --
>
<!-- interface de services requise -->
<!ELEMENT requires EMPTY>
<!ATTLIST  requires
  service CDATA #REQUIRED -- nom d'interface Java --
  filter CDATA #REQUIRED  -- filtre en format LDAP --
  cardinality (0..1|0..n|1..1|1..n) #REQUIRED
  policy (static|dynamic) #REQUIRED
  bind-method CDATA #REQUIRED
  unbind-method CDATA #REQUIRED
>
```

Exemple

```
<bundle>
  <component implementation="org.examples.impl.SpellCheckComponent">
    <provides service="org.examples.interfaces.SpellCheckService"/>
    <property name="version" value="1.0" type="string"/>
    <requires
        service="org.examples.interfaces.DictionaryService"
        filter="(language=*)"
        cardinality="1..n"
        policy="dynamic"
        bind-method="addDictionary"
        unbind-method="removeDictionary"
    />
  </component>
</bundle>
```

FIG. 6.1 – Descripteur de Composant à Services

```
<filter> : := '(' <filtercomp> ')'
<filtercomp> : := <and> | <or> | <not> | <item>
<and> : := '&' <filterlist>
<or> : := '|' <filterlist>
<not> : := '!' <filter>
<filterlist> : := <filter> | <filter> <filterlist>
<item> : := <simple> | <present> | <substring>
<simple> : := <attr> <filtertype> <value>
<filtertype> : := <equal> | <approx> | <greater> | <less>
<equal> : := '='
<approx> : := '~='
<greater> : := '>='
<less> : := '<='
<present> : := <attr> '=*'
<substring> : := <attr> '=' <initial> <any> <final>
<initial> : := NULL | <value>
<any> : := '*' <starval>
<starval> : := NULL | <value> '*' <starval>
<final> : := NULL | <value>
```

FIG. 6.2 – Grammaire LDAP

`component`, dans laquelle est définie une classe Java qui correspond à l'implémentation du composant à services. Les éléments correspondants à la vue interne d'un composant à services, notamment les interfaces de contrôle, et les dépendances d'implémentation, ne sont pas décrites dans le descripteur de composant. Les interfaces de contrôle sont décrites dans l'implémentation du composant (voir 6.3.1.3), et les dépendances d'implémentation sont décrites dans un fichier associé aux paquetages, dans lesquels sont déployées les composants à services, ceci est décrit dans la section 6.3.2.

6.3.1.1 Interfaces de service fournies et propriétés de service

Le nom d'une interface de service correspond à une interface Java. Le ServiceBinder n'est pas concerné par des problématiques de versionnement au niveau des interfaces de service du fait que la plate-forme sous-jacente ne permet d'avoir qu'une seule version d'une interface de service présente à un moment donné.

Les propriétés, qui identifient les services d'un composant à services, ont un nom, un type et une valeur. Les types correspondent aux types simples de Java (`int`, `float`, `string`, etc...).

6.3.1.2 Interfaces de service requises

Les caractéristiques d'une interface de service requise sont décrites à l'intérieur de la balise `requires`, elles sont :

`service` le nom d'une interface de service, qui correspond à une interface Java.

`filter` le filtre qui est décrit dans une syntaxe LDAP dont la grammaire est présentée dans la figure 6.2.

`cardinality` la partie inférieure de la cardinalité permet d'exprimer le caractère optionnel ou obligatoire d'une interface de service requise et la partie supérieure

de la cardinalité permet d'exprimer la multiplicité. De cette façon, une interface de service requise avec une cardinalité `1..n` représente des connexions multiples dont au moins une est obligatoire.

`policy` la politique est statique (`static`), ou dynamique (`dynamic`).

`bind-method` et `unbind-method` noms des méthodes de liaison et de retrait, présentes au niveau de l'implémentation. Ces méthodes permettent aux instances de composant à services de participer lors de la reconfiguration. Ces méthodes sont décrites à la suite.

6.3.1.3 Implémentation du composant à services

Un composant à services est implémenté par une classe Java qui doit implémenter les interfaces de service fournies déclarées dans le descripteur de composant. Lorsque le composant à services déclare des interface de services requises, la classe d'implémentation doit supporter les caractéristiques décrites dans le descripteur du composant à services. Par exemple, si la cardinalité est multiple, l'implémentation doit être préparée à recevoir plusieurs références vers des instances fournissant le service requis.

La classe d'implémentation doit obéir à certaines conventions pour permettre aux gestionnaires des instances de gérer leur cycle de vie pendant l'exécution. Ces caractéristiques sont décrites à la suite.

Interface de contrôle Lifecycle

Une interface de contrôle appelée `Lifecycle` doit être implémentée en particulier lorsque le composant à services possède des interfaces de service requises. Cette interface permet au gestionnaire de l'instance de réaliser l'activation de l'instance, ainsi que de la notifier du fait qu'un changement va avoir lieu dans les connexions. Cette interface est présentée dans la figure 6.3.

Par rapport aux étapes du cycle de vie présentées dans le chapitre précédent (section 5.4.3.1), l'appel de la méthode `activate` est réalisé au moment de la validation d'une instance, avant que les services de l'instance soient enregistrés tandis que l'appel de la méthode `deactivate` est réalisé au moment de l'invalidation, après que les services de l'instance soient retirés. La méthode `suspend` correspond à la notification qui est envoyée à une instance de composant avant une reconfiguration tandis que la méthode `resume` correspond à la notification envoyée à la fin de la reconfiguration.

Méthodes de liaison et de retrait

Pour chaque interface de service requise définie au niveau du descripteur, la classe d'implémentation doit définir une paire de méthodes de liaison et de retrait. Ces méthodes

```
public interface Lifecycle
{
    /**
     * Méthode appelée pour notifier l'instance de son activation
     * cette méthode est appelée avant l'enregistrement des services
    **/
    public void activate();
    /**
     * Méthode appelée pour notifier l'instance de sa déactivation
     * cette méthode est appelée après le retrait des services
     *
     * @param fromstatic indique si l'invalidation résulte de changement
     *                   dans connexion statique
    **/
    public void deactivate(boolean fromstatic);

    /**
     * Méthode appelée avant tout changement au niveau des connexions.
    **/
    public void suspend();
    /**
     * Méthode appelée à la fin des changements au niveau des connexions
    **/
    public void resume();
}
```

FIG. 6.3 – Interface de contrôle

permettent au gestionnaire de l'instance de réaliser les connexions entre les instances de composant pendant l'exécution. Les méthodes de liaison et de retrait sont considérées comme étant des méthodes de contrôle car elles ne sont visibles que pour le gestionnaire des instances, elles ne sont cependant pas définies dans une interface de contrôle du fait que leur nombre et leur nom peuvent varier dans chaque composant à services.

Passage de contexte

Le constructeur de la classe d'implémentation peut recevoir, de façon optionnelle, un paramètre contenant un contexte qui permet à une instance de composant d'interagir avec l'environnement d'exécution. L'interaction inclut l'accès aux informations définies dans le descripteur, ainsi que l'accès au registre de services de la plate-forme OSGi. La possibilité d'accéder au registre de services est fournie pour permettre de garder une compatibilité avec la plate-forme de services OSGi.

6.3.2 Paquetages de composant

Les composants à services sont conditionnés dans des fichiers JAR qui correspondent aux paquetages de la plate-forme OSGi appelés *bundles*. Le paquetage contient un fichier *manifest* ainsi qu'un descripteur de composants à services, dans lequel sont dé-

```
Bundle-Activator : org.examples.impl.Activator
Import-Package : org.examples.interfaces,
 org.ungoverned.gravity.servicebinder
Bundle-Classpath : .,library.jar
Bundle-Name : Spell checker component
Bundle-Description : A spell check service provider
Bundle-Version : 1.0.0
Metadata-Location : org/examples/impl/res/metadata.xml
```

FIG. 6.4 – Exemple de fichier manifest

crits un ou plusieurs composants à services. Le fichier manifest, dont un exemple est donné dans la figure 6.4, contient une clé (`Bundle-Classpath`) qui définit l'emplacement, à l'intérieur du paquetage, de ressources nécessaires aux composants à services. Ces ressources correspondent à des dépendances d'implémentation et sont des fichiers binaires (par exemple des images) ou bien des librairies. Le fichier manifest contient aussi l'emplacement du descripteur de composants à services, donné par la clé `Metadata-Location`. En plus de ces fichiers, le paquetage contient le code binaire correspondant aux interfaces de services et aux implémentations des composants.

Un paquetage contient aussi une classe d'activation, définie à travers la clé `Bundle-Activator`, et une liste de dépendances de code, définie comme une liste de packages Java dans la clé `Import-Package`. Lorsqu'un paquetage de composant est déployé dans la plate-forme, le code qu'il contient est considéré comme étant privé au paquetage et n'est pas accessible à d'autres paquetages déployés simultanément dans la plate-forme. Certaines parties de code peuvent nécessiter cependant d'être partagées ; c'est notamment le cas des définitions des interfaces de service ainsi que des librairies. Pour supporter ce partage, un paquetage peut déclarer qu'il exporte ou qu'il importe des espaces de noms (`packages`) de classes Java. Il faut noter qu'à l'exception de `Metadata-Location`, qui référence l'emplacement du descripteur de composants, l'ensemble des clés présentées dans cette description correspondent à des clés standard OSGi.

Cycle de vie de déploiement

Un paquetage a un cycle de vie bien défini qui est illustré dans la figure 6.5. Les étapes de ce cycle de vie sont les suivantes :

`INSTALLED` Lorsqu'il est installé, un paquetage de composant à services est introduit dans la plate-forme. L'installation est réalisée à partir d'une URL.

`RESOLVED` Quand un paquetage est installé, la plate-forme vérifie de façon automatique que ses dépendances de code soient valables, c'est à dire que les espaces de noms importés par le paquetage soient exportés à ce moment là par un autre paquetage installé dans la plate-forme ; si les dépendances de code sont

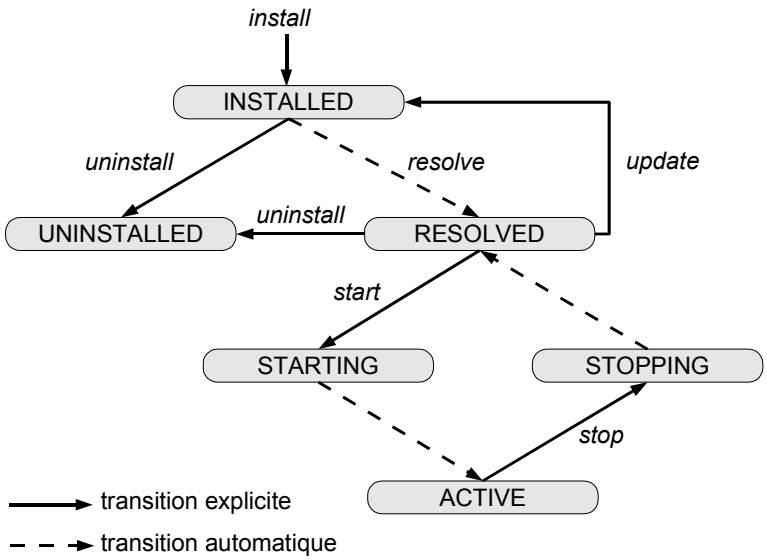


FIG. 6.5 – Cycle de vie d'un paquetage de composant

valables, le paquetage est considéré comme étant résolu (`RESOLVED`) et il peut alors être activé. La transition vers l'état `RESOLVED` à partir de l'état `INSTALLED` est réalisée de façon automatique.

`STARTING` Lorsqu'un paquetage est activé, il entre dans une étape de démarrage. Une instance de la classe d'activation est créée à ce moment là. Une méthode d'activation (`start`) de cette classe est appelée.

`STARTED` Lorsque l'étape d'activation a conclu, le paquetage de composants est considéré comme étant actif.

`STOPPING` Lorsqu'un paquetage actif est désactivé, il entre dans une étape de dé-activation. Lors de cette étape, la classe d'activation est appelée à nouveau par la plate-forme (méthode de dé-activation `stop`). Une fois que l'arrêt conclut, le paquetage retourne à l'état `RESOLVED`.

`UNINSTALLED` Lorsqu'un paquetage est dés-installé, il ne peut plus être utilisé à moins d'être ré-installé à nouveau.

6.3.3 Instances de déploiement

La création et destruction des instances de déploiement a lieu au moment de l'activation (`STARTING`) et de la dé-activation (`STOPPING`) du paquetage de composants, respectivement. La plate-forme de services OSGi fournit des mécanismes permettant d'administrer le cycle de vie des paquetages. Dans la pratique l'administration des paquetages

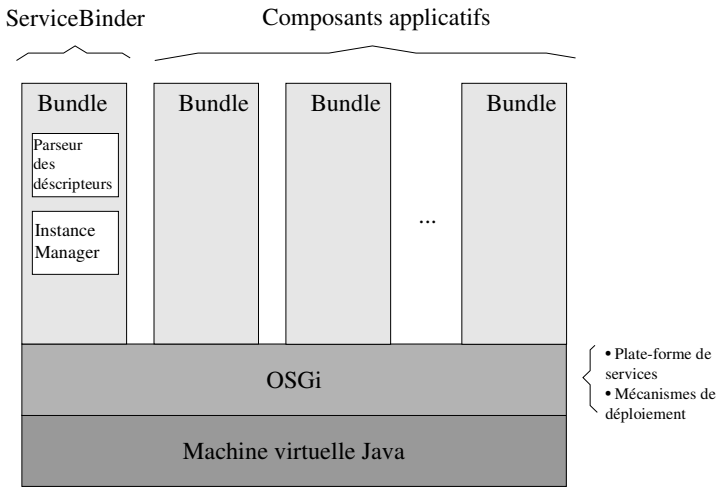


FIG. 6.6 – Vue globale du système

peut être réalisée de façon locale, par exemple à travers une console, ou bien de façon distante.

6.3.4 Environnement d'exécution

La figure 6.6 montre une représentation des différents éléments qui composent le système. L'environnement d'exécution est constitué par la machine virtuelle Java, la plate-forme de services OSGi, ainsi qu'un bundle correspondant au ServiceBinder. Le ServiceBinder réalise d'un côté l'interprétation et la réification de l'information contenue au niveau de descripteurs de composants et d'un autre la logique d'adaptation. Les applications sont construites à partir des autres bundles installés dans le système.

Il est important de souligner que ce système est orienté à la construction d'applications non distribuées. Ce choix est imposé par le fait que la plate-forme de services OSGi est elle même non-distribuée et est orientée à des environnements restreints. Cette contrainte permet cependant de se concentrer dans les aspects concernant l'adaptation sans devoir prendre en compte des problématiques propres aux systèmes distribués, comme les failles dans la réception de notifications ou la communication distribuée. Un bénéfice additionnel est que l'implémentation est d'une taille réduite (~70K), ce qui permet son utilisation dans le type d'environnements pour lesquels OSGi a été conçu.

Le ServiceBinder profite du mécanisme d'OSGi permettant d'exporter des espaces de noms pour exporter un framework, qui contient notamment l'interface `Lifecycle`, mais aussi une classe d'activation générique qui doit simplement être sous-classée dans

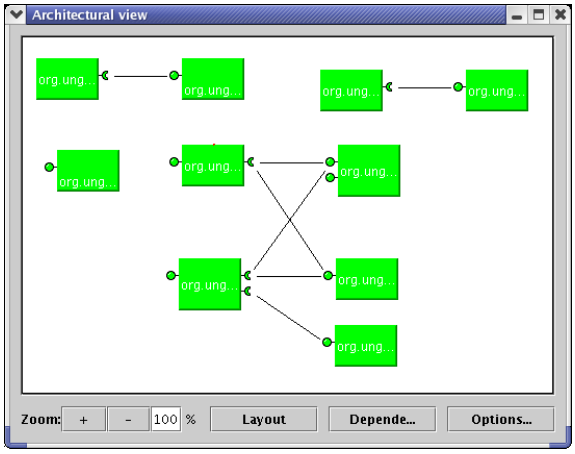


FIG. 6.7 – Vision de l'architecture lors de l'exécution

les paquetages contenant des composants à services. Le fait de déployer le ServiceBinder à l'intérieur d'un bundle permet l'indépendance par rapport à une implémentation particulière de la plate-forme de services OSGi.

6.3.4.1 Gestionnaire d'instances

Le code correspondant aux gestionnaires d'instance, appelés `InstanceManager`, implémente principalement les algorithmes présentés dans la section 5.4.3. Pendant l'exécution, les gestionnaires s'exécutent dans un fil d'exécution (thread) commun et utilisent le mécanisme de synchronisation de Java pour éviter d'avoir des problèmes dûs à la concurrence par rapport à des appels provenant d'autres fils d'exécution (comme celui des swing, par exemple). Ceci permet de synchroniser les activités de reconfiguration et d'éviter des blocages à l'intérieur des gestionnaires d'instances.

6.3.4.2 Service d'introspection architecturale

Si le bundle correspondant au ServiceBinder est activé, celui-ci fournit un service permettant de réaliser de l'introspection par rapport à l'architecture de l'application qui s'exécute. Ceci est avantageux car une telle information n'est pas présente dans OSGi.

Le service fourni par le ServiceBinder peut ensuite être utilisé par différents clients, par exemple pour réaliser des activités de débogage. Un exemple de client est un visualisateur graphique de l'architecture, ce visualisateur est présenté dans la figure 6.7.

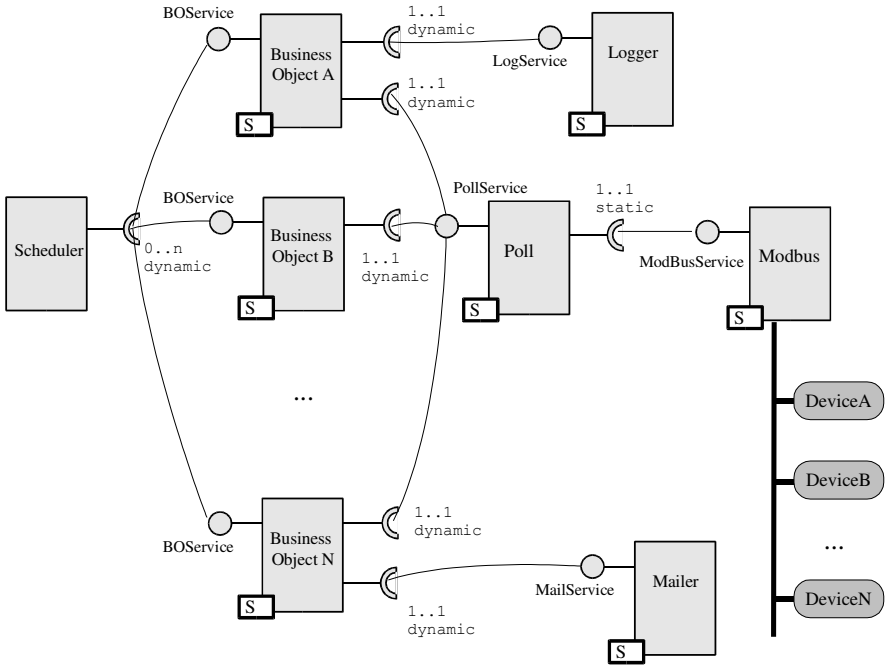


FIG. 6.8 – Architecture de l'application de Schneider Electric

6.4 Évaluations

Le ServiceBinder a été publié comme un projet de source libre[1] qui jusque aujourd'hui a été évalué dans deux projets réalisés dans un contexte industriel. Ces projets présentent des exemples réels d'applications construites à partir d'instances de déploiement. Cette section présente ces applications ainsi qu'un positionnement du ServiceBinder par rapport à la plate-forme OSGi standard.

6.4.1 Exemples d'applications

Le ServiceBinder a été évalué dans deux projets industriels qui sont présentés à la suite.

6.4.1.1 ServiceBinder à Schneider Electric

Le ServiceBinder est utilisé au sein de Schneider Electric dans la construction d'une application s'exécutant de façon continue sur une passerelle reliée à des appareils électriques à travers un *bus de terrain*. Le but de cette application est de surveiller les appareils électriques et de réagir par rapport à des changements dans l'état des divers appareils.

[1]La page du projet se trouve à l'adresse suivante : http ://gravity.sourceforge.net/servicebinder/

L'application est formée par un certain nombre d'*objets métier* ("BusinessObject") qui sont chargés d'interroger les appareils électriques et de réagir de diverses manières par rapport à certaines situations, par exemple en envoyant un courrier électronique à un administrateur, ou bien de façon continue, en écrivant une trace. Un besoin dans l'application est celui de permettre à un administrateur de pouvoir rajouter ou retirer les objets métier pendant l'exécution pour pouvoir introduire ou retirer des nouveaux moyens de surveillance des appareils électriques.

La combinaison OSGi/ServiceBinder a été favorisée dans ce projet du fait que l'application s'exécute dans un environnement restreint. L'architecture de l'application construite par Schneider Electric est représentée dans la figure 6.8 en termes d'instances de composants (montrées à travers leur vue externe). Les instances principales de cette architecture sont :

Scheduler Le scheduler est un composant qui coordonne l'application. Son objectif est d'activer de façon périodique les objets métier qui se trouvent présents à un moment donné dans le système. Ce composant représente le composant principal de l'application, qui joue ici le rôle de noyau applicatif extensible[2].

Business Object Les objets métier contiennent la logique qui réalise la supervision des équipements électriques. Lors de la supervision d'un équipement, certaines conditions peuvent nécessiter d'une action, par exemple de notifier un administrateur du système par courrier électronique ou bien de l'écriture d'une trace. Dans l'architecture présentée dans la figure 6.8, l'objet métier A dépend d'un service de traçage, tandis que l'objet métier N dépend d'un service permettant d'envoyer un courrier électronique. Le nombre d'objets métiers varie pendant l'exécution de l'application.

Poll Le Poll est un intermédiaire entre le ModBus et les objets métier. Son but est d'interroger le ModBus à partir de critères donnés par chaque objet métier et de traduire l'information obtenue à partir de celui ci pour la transmettre aux objets métier.

ModBus Le ModBus est une interface vers le bus de terrain qui est connecté aux équipements électriques (devices).

L'architecture présentée dans la figure permet d'apprécier les caractéristiques des dépendances qui sont utilisées pour supporter l'ajout et le retrait d'objets métier. En particulier, le *Scheduler* possède une dépendance `0..n` dynamique envers un service fourni par chaque objet métier. Chaque objet métier est ensuite connecté au *Poll*, qui à son tour est connecté au *ModBus*.

[2]Dans l'implémentation originale, le scheduler était plutôt un noyau fournissant un service, mais ceci a été modifié pour des fins d'illustration.

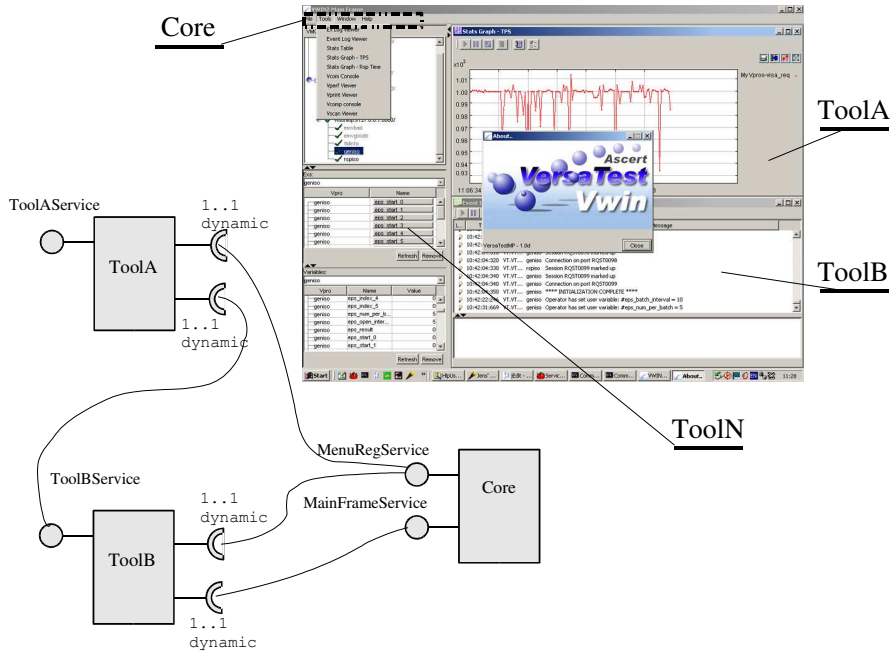


FIG. 6.9 – Client VersaTest

Les retours de la part de Schneider Electric sur l'utilisation du ServiceBinder ont été très positifs ; de plus, un des responsables du projet (Marc Chachereau) a contribué par des améliorations au code du ServiceBinder. Le prototype réalisé par Schneider a été testé sur l'implémentation d'OSGi réalisée par IBM.

6.4.1.2 Client VersaTest

Un autre cas d'évaluation industrielle du ServiceBinder a été réalisée au sein d'une entreprise, nommée Ascert, dédiée à la création d'applications de test. Concrètement, le ServiceBinder a été utilisé pour construire un client d'une application de monitoring et de contrôle appelée VersaTest[3]. Le besoin concret de Ascert était de pouvoir construire un système extensible [Szy96] permettant de créer différentes configurations du client. Chaque configuration est basée sur un ensemble différent d'outils qui permettent de réaliser et de visualiser des tests réalisés par VersaTest. Après une évaluation de diverses plateformes extensibles telles que NetBeans, Ascert a décidé de construire son application à partir du ServiceBinder et d'utiliser OSCAR comme implémentation OSGi sous-jacente.

Le client VersaTest est construit comme un noyau applicatif fournissant un ensemble de services permettant à divers outils de s'intégrer dans l'application. L'intégration des outils inclut par exemple le rajout de sous-menus dans la barre de menu principale. Chaque

[3]http ://www.ascert.com/versatest.html

outil est modélisé comme un composant ServiceBinder qui requiert d'un côté les services fournis par le noyau, et qui peut fournir et demander des services fournis par d'autres outils. Une représentation schématique de l'architecture ainsi qu'une capture d'écran du client VersaTest sont présentés dans la figure 6.9.

La réalisation de différentes configurations du client est réalisée au démarrage du framework OSGi. A ce moment là, la plateforme est démarrée avec un ensemble de bundles correspondant au noyau et aux outils relatifs à une configuration particulière (ceci est réalisé à partir de fichiers de propriétés dans OSCAR). Dans cette application le ServiceBinder a surtout pour rôle de permettre aux différentes configurations de l'application de s'assembler dynamiquement. Ce système supporte aussi la réalisation de mises à jour des outils ou du noyau, ainsi que l'ajout ou retrait d'outils de façon dynamique[4]. Les retours de la part de Ascert sur l'utilisation du ServiceBinder ont été très positifs.

6.4.2 ServiceBinder par rapport à OSGi

Cette section discute divers aspects qui positionnent le ServiceBinder par rapport à OSGi.

6.4.2.1 Améliorations introduites

Dans la plate-forme OSGi, il n'existe pas de concept de composant. Un bundle contient simplement un ensemble de fournisseurs et de demandeurs de service qui sont créés depuis la classe d'activation. Les relations qui peuvent exister entre les services fournis et requis ne sont pas explicites et sont contenues à l'intérieur du code déployé par le bundle. La classe d'activation pourrait être considérée comme un composant. Cependant une seule classe d'activation, dont une seule instance est créée, peut être déployée dans un bundle tandis que le ServiceBinder permet de déployer plusieurs composants à services.

Dans OSGi, toutes les activités concernant l'enregistrement de services, l'assemblage d'une application et l'adaptation par rapport aux changements doivent être réalisés à travers la programmation. De plus, la logique d'assemblage et d'adaptation est souvent mélangée à la logique applicative. Le ServiceBinder simplifie le développement à travers la séparation entre la logique d'adaptation et la logique applicative. Un exemple comparatif de programmation OSGi standard et de programmation avec le ServiceBinder est présenté dans l'annexe B.

6.4.2.2 Compatibilité avec OSGi

Les concepts introduits par le ServiceBinder sont complètement compatibles avec ceux de la plate-forme sous-jacente. Les demandeurs de service programmés en OSGi

[4]Ces dernières possibilités ne sont cependant pas encore exploitées dans la version commerciale.

Tâche	OSGi Standard (en octets)	ServiceBinder (en octets)	Delta
Enregistrement de service	156	537	244%
Enregistrement de Service plus une dépendance de service	864	1344	55%
Dépendances additionnelles	768	1032	34%

TAB. 6.1 – Consommation Mémoire ServiceBinder vs. OSGi Standard

'standard' peuvent utiliser les services fournis par les instances de composant à services et vice-versa.

Le ServiceBinder est indépendant d'une implémentation particulière de la plate-forme OSGi, du fait qu'il est livré comme un bundle standard. Par ailleurs, le ServiceBinder peut être obtenu à travers le catalogue de bundles (OBR - Oscar Bundle Repository) de l'implémentation OSCAR de la plate-forme OSGi [5].

6.4.2.3 Limitations par rapport à OSGi

Actuellement, le concept de `ServiceFactory`, qui permet d'avoir une politique de création d'instances d'un objet de service par demandeur n'est pas supporté dans le ServiceBinder. La décision de la manière d'implémenter ce concept n'a pas encore été prise du fait que dans OSGi standard, tout code contenu dans un bundle est considéré comme faisant partie du même demandeur. Cependant, dans le ServiceBinder, un bundle permet de déployer différents composants à services, qui peuvent être effectivement considérés comme des étant des demandeurs indépendants. Le souci de compatibilité avec la plate-forme OSGi laisse penser que le premier cas serait plus approprié, de plus les composants ServiceBinder qui doivent être considérés comme des demandeurs distincts peuvent être déployés indépendamment.

6.4.2.4 Performances par rapport à OSGi

Bien que l'utilisation du ServiceBinder donne lieu à plusieurs avantages par rapport à la programmation dans la plate-forme OSGi standard, son utilisation implique un surcoût au niveau de la consommation de mémoire en raison de la création d'un certain nombre d'objets nécessaires à la gestion de chaque instance de composant et chaque connexion entre les instances. Pour connaître le surcoût de l'utilisation du ServiceBinder

[5] Cette implémentation est disponible à l'adresse suivante : http ://oscar-osgi.sourceforge.net

avec plus de détail, une étude a été menée à partir des techniques d'étude de performance de programmes Java décrites dans [WK00] ; ces techniques ont permis de réaliser des estimations sur la consommation de mémoire introduite par le ServiceBinder. Ces tests ont été réalisés en calculant la consommation de mémoire à partir de la création d'un grand nombre d'instances de composant et en divisant le résultat par le nombre d'instances. Chaque test a été réalisé dix fois chacun en déactivant le ramasse-miettes de la machine virtuelle Java sur un PC Linux à 700MHz avec 512 Mb de Mémoire. Les tests ont été réalisés sur OSCAR.

Le premier test de consommation de mémoire a permis de comparer l'entête au niveau de la mémoire qui est rajoutée par le ServiceBinder lors d'un enregistrement de service dans la plate-forme OSGi. Le test a été réalisé par rapport à une instance implémentant un service simple sans dépendances et avec un seul attribut. La différence de consommation de mémoire dans ce cas a été de 381 octets, soit 244% de plus que lorsque ceci est réalisé directement dans OSGi.

Le deuxième test a comparé la consommation de mémoire utilisée par le ServiceBinder lors de l'enregistrement d'un service et la gestion d'une dépendance. Ce test a été réalisé en créant une instance fournissant un service et ayant une dépendance de service unique ; cette dépendance est de cardinalité `0..n` et dynamique. La différence de consommation de mémoire a été dans ce cas de 480 octets, ce qui représente une consommation additionnelle de mémoire de 55% lors de l'utilisation du ServiceBinder.

Le troisième test a été orienté à l'étude de la consommation de mémoire correspondante à chaque dépendance additionnelle par rapport au deuxième test. Dans ce cas, la différence de consommation de mémoire a été de 264 octets par dépendance, ce qui représente une augmentation de 34%.

Les résultats de ces tests sont présentés dans la table 6.1. Bien que le surcoût présenté dans le premier test peut sembler excessif, il peut être expliquée par le fait que l'enregistrement d'un service nécessite de la réification du descripteur du composant ainsi que de la création d'objets comme le gestionnaire d'instance. Les autres tests montrent qu'au moment où des interfaces requises sont introduites, le surcoût imposée par le ServiceBinder n'est pas excessif. L'analyse postérieure de ces tests a permis en plus de déterminer qu'une grande partie de la consommation de la mémoire ne provenait pas spécifiquement du ServiceBinder mais provient plutôt du mécanisme de gestion des filtres de l'implémentation de OSGi utilisée dans les tests.

6.4.3 Limitations

Le ServiceBinder possède certaines limitations par rapport au blocage des appels entrants, aux propriétés de configuration et au filtrage côté client. Ces limitations sont discutées à la suite.

6.4.3.1 Blocage des appels entrants

Dans le ServiceBinder, une instance est prévenue de la réalisation d'une reconfiguration, c'est à dire d'un changement dans ses connexions, à travers les méthodes `suspend` et `resume` de l'interface `Lifecycle`. L'interruption de l'exécution nécessite cependant de bloquer des appels entrants vers l'instance du composant pendant que celle-ci n'est pas active. Ceci n'est actuellement pas pris en charge par le ServiceBinder et il doit être réalisé à l'intérieur des composants. Ceci pourrait cependant être réalisé en par exemple en plaçant le gestionnaire d'instance comme intermédiaire entre un client et une instance de composant. Au moment d'une reconfiguration, l'intermédiaire serait chargé de bloquer les appels entrants vers l'instance. La génération dynamique d'intermédiaires est réalisable en Java en utilisant le mécanisme de *proxy dynamiques*[6].

6.4.3.2 Propriétés de configuration

Dans l'implémentation courante du ServiceBinder, il n'est pas possible d'exprimer des propriétés de configuration permettant de configurer une instance au moment de sa création. Celles ci pourraient être décrites à partir d'une balise dans le descripteur de composant et leur valeur pourrait être modifiée avant l'introduction du paquetage de composant dans le système.

6.4.3.3 Filtrage côté client

Actuellement le ServiceBinder utilise un algorithme très simple pour résoudre la problématique du filtrage côté client. Lorsqu'un gestionnaire d'instance reçoit plusieurs réponses de la part du registre de services et qu'il doit en choisir une en particulier, il prend simplement la première réponse. Des algorithmes plus sophistiqués pourraient être envisagés, il pourraient se baser sur les propriétés de l'instance qui implémente le service.

6.5 Conclusion

Ce chapitre a présenté le concept d'instances de déploiement ainsi que les aspects relatifs à la réalisation du modèle à composants orienté services. Ce concept permet de créer des applications compatibles avec la plate-forme de services OSGi.

Bien que l'environnement d'exécution du modèle à composants a été réalisé au dessus de la plate-forme de services OSGi, une autre plate-forme de services, telle que Jini, aurait pu être utilisée. Le concept d'instance de déploiement nécessite cependant de la présence de mécanismes permettant de réaliser le déploiement continu.

[6] http ://java.sun.com/j2se/1.4.1/docs/api/java/lang/reflect/Proxy.html

Les évaluations réalisées à l'extérieur du laboratoire ont permis d'avoir des exemples applicatifs réels ainsi que des retours importants.

Chapitre 7

Applications à base d'instances dynamiques

7.1 Introduction

Dans le chapitre précédent, la construction d'applications à base d'instances de déploiement a été présentée. Cette politique suppose que l'introduction et retrait d'instances dans le système, c'est à dire les opérations *create* et *destroy*, sont causées par un acteur externe qui réalise l'administration de paquetages de composants de façon continue. Bien que cette politique de création d'instances soit suffisante pour développer un certain type d'applications, elle ne permet pas de réaliser des applications qui nécessitent de la création d'instances *dynamiques*, c'est à dire d'instances créées pendant l'exécution. Par ailleurs, cette politique n'est pas en accord avec les caractéristiques de l'approche à composants présentées dans le chapitre 3, car un composant a été défini comme pouvant donner lieu à de multiples instances.

Ce chapitre introduit les concepts de *fabrique* et d'*espaces de résolution*, qui permettent de supporter la création d'instances dynamiques ainsi que de limiter l'imprévisibilité résultante de cette situation. Ce chapitre décrit aussi un moyen de description et gestion de compositions ainsi que des évaluations qui ont été réalisées par rapport à ces concepts.

7.2 Fabriques

Un cas d'étude qui permet de mieux comprendre le besoin de pouvoir créer des instances pendant l'exécution, ainsi que la description du concept de *fabrique* sont présentés à la suite.

7.2.1 Cas d'étude : un lanceur d'applications XLet de TV interactive

Un exemple particulier qui permet de comprendre les limitations dans la construction d'applications à base d'instances de déploiement est le système OSGiTV, un sous-projet faisant partie du projet CompiTV[1] qui a été réalisé entre des acteurs industriels (Canal + Technologies et Gemplus) et des acteurs du milieu académique (Universités de Valenciennes et Lille). OSGiTV avait pour but d'étudier le moyen de déployer des applications, appelées *XLets*, dans des terminaux, c'est à dire des passerelles contenant une plate-forme OSGi, connectés à un téléviseur [CD03]. Les XLets peuvent représenter par exemple des applications interactives de jeu (PMU, Casino), des applications de gestion de comptes bancaires, ou autres. Un terminal possède un lecteur de cartes à puce et les cartes sont utilisées pour déployer certains composants à services qui ne sont disponibles que pendant que la carte est insérée dans le terminal. De nouvelles applications XLet et des mises à jour des applications XLet existantes sont obtenues périodiquement à partir d'informations reçues par la passerelle.

Au moment où le terminal est allumé, un menu proposé par un lanceur d'applications (XLetLauncher) présente la liste d'applications disponibles ; si l'utilisateur décide de lancer une application, une instance de celle-ci est créée puis contrôlée par un gestionnaire d'applications (XLetManager). Une application XLet peut dépendre des services fournis par les composants contenus dans la carte à puce, si cette carte est retirée, l'application peut fonctionner soit en mode dégradé soit ne plus fonctionner. Dans ce système, la présence de la carte à puce est à l'origine de la disponibilité dynamique de certains composants, et les applications XLet doivent s'adapter de façon autonome par rapport à la présence de la carte.

Une représentation simplifiée de l'architecture de OSGiTV lors de l'exécution est montrée dans la figure 7.1. Cette figure représente le lanceur et gestionnaire d'applications, et diverses applications XLet (instances de composant fournissant des services `XLetService`). Le schéma représente aussi une instance d'un composant déployé à partir de la carte à puce, qui dans ce cas est un moteur de jeu sécurisé (`CasinoEngineService`).

Ce cas d'étude est propice à la réalisation à partir de composants à services ; la dépendance entre le gestionnaire d'applications et les instances pourrait être modélisée par une dépendance dynamique `0..n`, et des dépendances entre les applications et les services déployés à partir de la carte à puce permettraient, par exemple, d'invalider une application au moment où la carte à puce est retirée.

Bien que le ServiceBinder a été employé dans ce projet, il n'a pas été utilisé pour gérer les dépendances entre le lanceur d'applications et les applications XLet, entre les applications XLet elles mêmes et entre les applications XLet et les composants de la carte

[1] http ://www.telecom.gouv.fr/rnrt/projets/res_01_47.htm

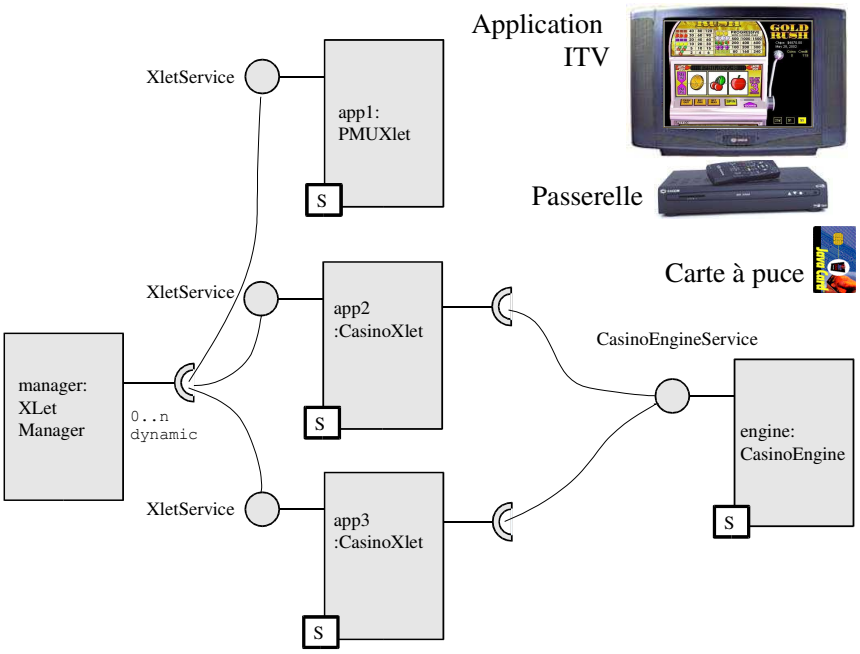


FIG. 7.1 – Architecture de OSGiTV

à puce[2]. Deux raisons sont à l'origine de ceci :

1. Dans ce système, le XLetManager et les instances d'applications XLet ne sont pas créés en même temps, le XLetLauncher déclenche la création des instances d'applications XLet à partir d'interactions avec un utilisateur.
2. Plusieurs instances d'une même application XLet doivent pouvoir exister de façon simultanée.

Finalement, il faut noter que le gestionnaire d'application et l'ensemble d'application XLet forment eux mêmes l'application "globale".

7.2.2 Principes

Dans les modèles à composants, le concept de fabrique (présenté dans la section 5.4.3) permet de supporter la création et destruction d'instances de composants pendant l'exécution. Ce concept est repris dans le modèle à composants orienté services pour supporter les opérations *create* et *destroy* présentées dans la section 5.2.2.2.

[2]Le ServiceBinder a seulement été utilisé pour faciliter la liaison statique d'instances de composant correspondant à des fonctionnalités de l'environnement d'exécution.

7.2.2.1 Service de fabrique

Dans un modèle à composants classique, une fabrique est typiquement un objet qui offre une interface fournissant des méthodes de création et de destruction d'instances de composant (par exemple les "homes" dans le modèle EJB). Dans le modèle à composants orienté services, une fabrique est un service présent dans le registre de services [CH00]. L'interface du service contient les méthodes de création et destruction d'instances de composants à services, permettant de réaliser les opérations *create* et *destroy*.

Une fabrique est associée à un seul composant à services. Cependant, toutes les fabriques de composants à services partagent la même interface de service. L'identification des fabriques est réalisée à partir des propriétés de service. Ces propriétés contiennent des informations relatives aux services et aux propriétés de service du composant à services associé à la fabrique.

7.2.2.2 Fournisseur du service

Le service de fabrique est fourni par des instances de déploiement n'ayant pas d'interfaces de services requis. Ceci permet au gestionnaire de ces instances de rendre disponible le service de fabrique immédiatement après que le composant à services soit introduit dans le système. Au contraire, le retrait du composant à services entraîne le retrait du service de fabrique.

Les opérations *insert* et *remove*, présentées dans la section 5.2.2.2 et qui correspondent à l'introduction et au retrait de composants à services dans le système, se traduisent par l'enregistrement ou le retrait des services de fabrique.

7.2.2.3 Création et validité des instances

Alors que dans l'approche à composants classique, l'acteur responsable de la création d'une instance de composant est souvent responsable aussi de réaliser la composition de cette instance, ceci est différent dans le modèle à composants orienté services. Dans ce modèle, la composition d'une instance dynamique, c'est à dire créée à partir d'une fabrique, est la responsabilité de l'environnement d'exécution à travers les gestionnaires d'instance. En raison des caractéristiques du cycle de vie des instances de composants à services présentées dans la section 5.4.3.1, il n'existe pas une garantie que la création d'une instance dynamique résulte dans une instance qui sera immédiatement validée par son gestionnaire.

Un acteur qui créé une instance dynamique reçoit une poignée (handle) qui représente l'instance et qui lui permet ensuite de détruire l'instance. Cette poignée ne lui permet cependant pas d'interagir avec les fonctionnalités de l'instance. Si cet acteur désire utiliser des services fournis par l'instance créée, il doit se lier avec celle-ci en obtenant ses services.

7.2.2.4 Destruction des instances

La destruction d'une instance dynamique peut être déclenchée par deux actions :
- Une demande de destruction explicite à travers la fabrique.
- Le retrait de la classe de composant à services à partir de laquelle l'instance a été créée. Ceci est nécessaire pour supporter l'hypothèse formulée dans la section 5.2.2.1.

7.2.3 Réalisation

Cette section discute la réalisation du concept de fabrique, qui inclut l'interface du service, la description d'une fabrique et le fournisseur du service.

7.2.3.1 Interface de service Factory

La figure 7.2 présente les interfaces `Factory`, correspondant au service de fabrique, ainsi que `InstanceReference` correspondant à la poignée qui représente l'instance créée. La méthode create de l'interface `Factory` reçoit comme paramètre une clé identifiant l'instance. Cette clé est associée à l'instance créée comme une propriété de service. De cette manière le créateur peut réaliser une liaison vers l'instance qu'il crée à travers les services qu'elle fournit même si d'autres instances fournissant le même service sont présentes dans le système.

7.2.3.2 Description d'une fabrique

Une fabrique est déclarée comme une classe de composant à services standard avec un attribut additionnel au niveau de la balise `component` et qui déclare que les instances du composant à services sont créées à partir d'une fabrique (Cet attribut est `factory="yes"`). La présence de l'attribut qui identifie une fabrique cause la création d'une instance de déploiement qui fournit le service `Factory` et dont les propriétés de service correspondent aux propriétés du composant à services associé à la fabrique.

La figure 7.3 présente la modification introduite au niveau de la DTD présentée dans le chapitre précédent et montre un exemple de descripteur représentant une fabrique ; cet exemple correspond plus précisément à une fabrique d'instances de XLets représentant une application XLet Casino qui utilise un moteur de jeu fourni par un autre service.

7.2.3.3 Fabriques et OSGi

Le concept de fabrique est compatible avec la plate-forme OSGi. Un concept similaire est celui de `ServiceFactory` qui a déjà été discuté dans la section 6.4.2.3. Un service factory est un service qui implémente une politique de création d'un objet de service par demandeur, le niveau de granularité du demandeur étant le bundle. Le concept de

```
public interface Factory
{
  /**
   * Méthode permettant d'obtenir une propriété de service
   * @param name nom de la propriété
  **/
  public Object getProperty(String name);
  /**
   * Méthode permettant de créer une instance
   * @param key clé identifiant l'instance
   * @return un InstanceReference représentant l'instance créée
  **/
  public InstanceReference createInstance(String key);
  /**
   * Méthode permettant de détruire une instance
   * @param ref InstanceReference de l'instance qui doit être
   *            détruite
  **/
  public destroyInstance(InstanceReference ref);
}



public interface InstanceReference
{
  /**
   * Méthode permettant d'obtenir une propriété de service
   * @param name nom de la propriété
  **/
  public Object getProperty(String name);
  /**
   * Méthode permettant de tester la validité de l'instance
   * @return true si l'instance est valide, false sinon
  **/
  public boolean isValid();
}
```

FIG. 7.2 – Interfaces Factory et InstanceReference

Modification dans DTD

```
<!ELEMENT component (provides*,property*,requires*)>
<!ATTLIST  component
  implementation CDATA #REQUIRED
  factory (yes|no) "no"
>
```

Exemple

```
<component implementation="org.examples.impl.CasinoImpl"
  factory="yes">
  <provides service="org.examples.interfaces.XLetService"/>
  <property name="appname" value="CasinoXLET" type="string"/>
  <property name="version" value="1.0" type="string"/>
  <requires
      service="org.examples.interfaces.CasinoEngineService"
      filter="(provider=TVCasinos S.A.)"
      cardinality="1..1"
      policy="static"
      bind-method="addEngine"
      unbind-method="removeEngine"
  />
</component>
```

FIG. 7.3 – Fabrique d'instances

fabrique implémente une politique de création d'instances multiples et permet de créer des instances de composant qui ne fournissent pas de services, ce qui n'est pas possible à travers le concept `ServiceFactory`.

7.2.4 Réalisation du cas d'étude avec fabriques

A l'aide du concept de fabrique, l'application OSGiTV aurait pu être réalisée comme le montre la figure 7.4. Dans cette figure, les instances nommées représentent des instances dynamiques, tandis que les autres représentent des instances de déploiement (signalées avec un paquetage), les instances représentant des fabriques sont identifiés par une usine. Bien que ce schéma représente l'ensemble des instances de composants à services présentes pendant l'exécution, celles correspondant aux fabriques et celles correspondant à l'application se trouvent à différents niveaux conceptuels.

Le lanceur d'applications, XLetLauncher, possède une dépendance optionnelle, multiple et dynamique vers le service fabrique (service `Factory`). Du fait que différentes fabriques peuvent être présentes dans le système de façon simultanée, le filtre `appname=*` permet de spécifier une liaison uniquement envers des fabriques de XLets (ceci suppose qu'une convention est établie dans laquelle tout XLet doit avoir une propriété `appname`). Pendant l'exécution, le lanceur d'applications est lié aux différentes fabriques de Xlets au fur et à mesure de l'enregistrement de leurs services. Le lanceur d'applications peut en-

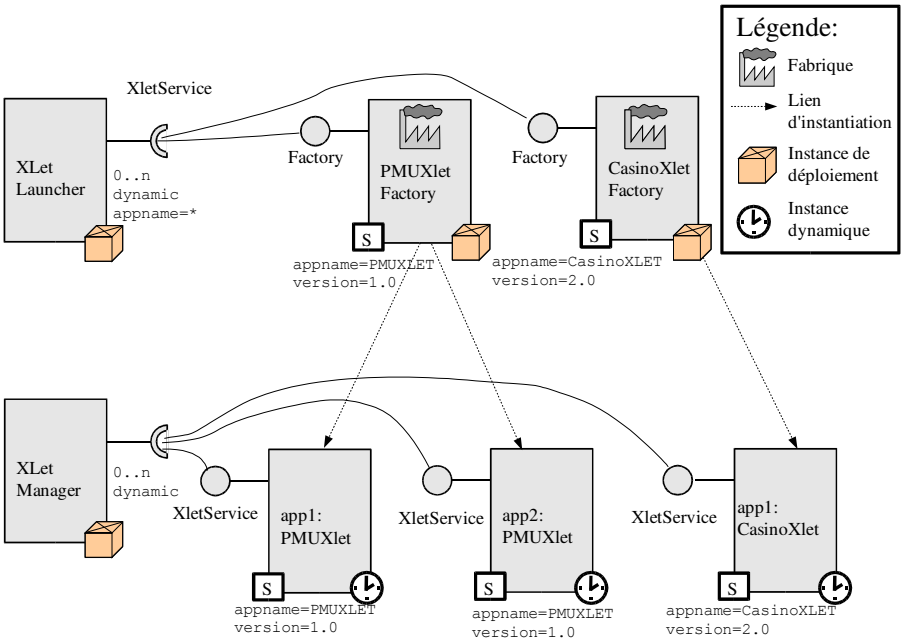


FIG. 7.4 – Réalisation de l'application OSGiTV avec fabriques

suite proposer par exemple dans un menu, la liste des applications disponibles qui est obtenue à partir des fabriques vers lesquelles il est connecté.

Le lancement d'une application résulte d'une interaction entre un utilisateur et le lanceur d'applications. Lors du lancement, le lanceur d'applications appelle la méthode de création dans l'interface du service de fabrique. Une instance du composant à services de l'application XLet est alors créée. Si cette instance peut être validée par son gestionnaire, son service XLetService devient disponible. Ceci a pour conséquence que le gestionnaire d'applications XLet, le XLetManager, qui déclare une interface requise multiple et dynamique vers l'interface de service XLetService, soit connecté à la nouvelle instance.

7.2.5 Synthèse

Cette section a présenté le concept de fabrique qui permet d'introduire ou de retirer des instances dynamiques dans un système pendant l'exécution. Ce concept se différentie de celui de l'approche à composants classique par le fait que la méthode de création ne permet pas d'accéder directement aux fonctionnalités de l'instance créée ou de composer cette dernière.

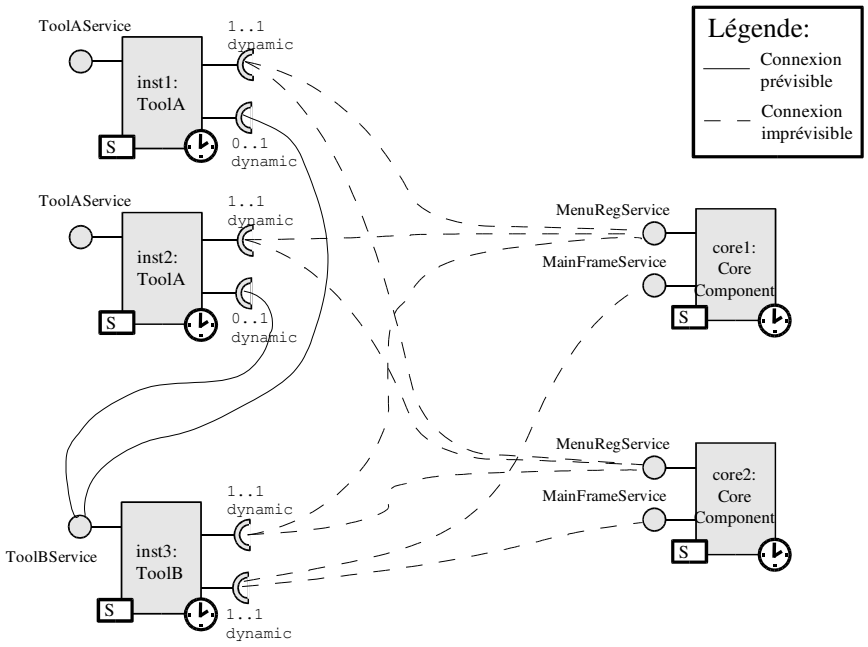


FIG. 7.5 – Exemple d'imprévisibilité

7.3 Espaces de résolution

Cette section présente le concept d'espace de résolution, qui limite le problème de l'imprévisibilité au moment de la création d'instances dynamiques.

7.3.1 Le problème de l'imprévisibilité

L'imprévisibilité, qui a été présentée dans la section 5.5.7, fait référence au fait que dans certaines situations les connexions entre instances de composant sont réalisées de façon imprévisible par les gestionnaires des instances. Cette situation a lieu au moment où plusieurs instances de composants requièrent des interfaces de service qui sont fournies par diverses instances de composants de façon simultanée. Un exemple d'imprévisibilité est présenté dans la figure 7.5. Cet exemple est basé sur le client VersaTest présenté dans la section 6.4.1.2. La situation représentée pourrait se présenter par exemple si deux configurations différentes de l'outil étaient créées de façon simultanée.

Dans cet exemple, et à différence du cas réel, il est supposé que les instances de composant sont des instances dynamiques. Dans la figure, deux instances du composant noyau applicatif sont présentes ainsi que deux instances d'un outil ToolA et une instance d'un outil ToolB (les fabriques ne sont pas montrées pour simplifier le diagramme). Cet exemple présente le fait qu'au moment de l'assemblage, il n'est pas possible de prédire quelle instance de quel outil sera connectée à quelle instance du noyau. Les connexions en

ligne solide représentent des connexions prévisibles, tandis que les connexions en pointillés représentent des connexions imprévisibles.

Bien que l'imprévisibilité est présente lors de la création d'instances non-multiples (par exemple les instances de déploiement), cette problématique peut être limitée à travers l'utilisation de filtres au niveau des interfaces de service requises. L'introduction d'instances dynamiques multiples pose cependant un problème par rapport à cette solution, du fait que chaque instance partage la même information au niveau des interfaces requises, ce qui inclut le filtre.

7.3.2 Principes

Les principes associé au concept d'espace de résolution incluent le concept lui même et la connexion à travers les espaces de résolution.

7.3.2.1 Concept d'espace de résolution

Un *espace de résolution* ("scope") contient une ensemble d'instances de composant. Ce concept permet de limiter la problématique de l'imprévisibilité, lors de la présence d'instances dynamiques multiples, en contraignant la réalisation de connexions entre les instances de composant qu'il contient à l'intérieur d'une frontière.

Conceptuellement, à chaque espace correspond un registre de services indépendant qui ne contient que les services correspondant aux instances contenues dans l'espace de résolution. Un gestionnaire d'instance travaille par rapport aux services correspondant à l'espace dans lequel l'instance qu'il gère a été placée.

Le registre de services de la plate-forme de services est considéré comme appartenant à un espace *global*. A l'exception de l'espace global, les espaces de résolution peuvent être créés et détruits pendant l'exécution et l'acteur responsable de créer les espaces de résolution est typiquement le même acteur qui créé les instances de composant. La création des espaces de résolution est hiérarchique, car un espace de résolution est toujours créé à l'intérieur d'un autre. La destruction d'un espace parent a pour conséquence la destruction des espaces qu'il contient.

Au moment de sa création, une instance est associée à un espace de résolution. L'instance est retirée de l'espace de résolution au moment où elle est détruite, et la destruction d'un espace de résolution entraîne la destruction des instances qu'il contient. Le déplacement d'une instance entre espaces de résolution n'est pas considéré.

7.3.2.2 Connexions à travers des espaces

Pour supporter la connexion entre espaces de résolution, il est nécessaire d'*exporter* des interfaces de service fournies ou requises vers l'extérieur d'un espace. La possibilité

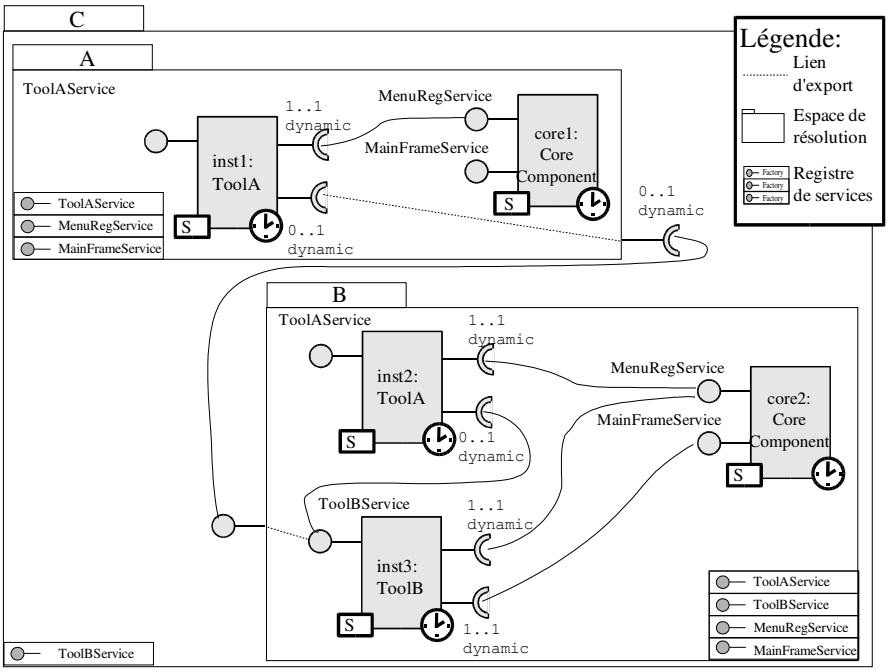


FIG. 7.6 – Connexion à travers des espaces de résolution

d'exporter des interfaces de service requises ou fournies signifie que pour une instance qui est créée à l'intérieur d'un espace, des interfaces fournies ou requises peuvent être marquées comme étant exportées en dehors de l'espace où elle est créée. La sélection des interfaces exportées est faite au moment où une instance est créée et ne sont pas modifiables postérieurement.

Lorsqu'une interface fournie est exportée, elle est ajoutée au registre de l'espace parent (l'espace global si aucun espace parent n'a été créé). Lorsqu'une interface requise est exportée, le gestionnaire de l'instance qui exporte cette interface utilise non seulement le registre associé à l'espace dans lequel a été créée l'instance mais aussi celui de l'espace parent pour la création de connexions. L'export d'interface et la création d'espaces de résolution imbriqués permettent de réaliser, en temps d'exécution, des compositions hiérarchiques.

La figure 7.6 présente un exemple de hiérarchie d'espaces de résolution ainsi qu'une connexion à travers des espaces réalisée à partir d'interfaces de service exportées. Dans cet exemple les espaces A et B sont créés à l'intérieur d'un espace parent C. Une interface fournie est exportée de l'espace A et une interface de service requise est exportée de l'espace B. Ceci a pour résultat que l'instance qui implémente l'interface de service exportée soit enregistrée dans le registre de l'espace C et qu'une connexion puisse être réalisée entre une instance de l'espace A avec celle de l'espace B.

7.3.3 Réalisation

Par manque de temps, l'implémentation du concept d'espace de résolution n'a été réalisé que de façon partielle. Cette section résume la réalisation de ce concept.

7.3.3.1 Création et destruction d'espaces de résolution

La création et destruction d'espaces de résolution est réalisée à partir d'une classe fournie par le framework exporté par le ServiceBinder. Les opérations fournies par cette classe sont :

`Scope createScope(Scope parent);` permet de créer un espace de résolution. La création d'espaces hiérarchiques est rendue possible à travers le paramètre passé à cette méthode, qui référence l'espace de résolution à l'intérieur duquel l'espace est créé. Si aucun espace de résolution parent n'est spécifié, l'espace est crée à l'intérieur de l'espace global.

`destroyScope(Scope scope);` détruit un espace de résolution. La destruction d'un espace de résolution entraîne la destruction des instances ainsi que d'autres espaces de résolution qu'il contient. L'espace de résolution global ne peut être détruit.

La classe `Scope` est une poignée et ne fournit pas de fonctions permettant de manipuler les instances contenues dans un espace.

7.3.3.2 Création d'instances dans un espace de résolution

Pour supporter la création d'instances à l'intérieur d'un espace de résolution, la méthode `create` de l'interface de service `Factory` doit être modifiée pour recevoir en paramètre une référence vers l'espace de résolution dans lequel doit résider l'instance ainsi qu'une liste des interfaces de service fournies et requises exportées par l'instance. La modification sur cette méthode est la suivante :

```
InstanceReference create(String key, Scope scope,
     String[] exportedProvided, String[] exportedRequired);
```

La liste des interfaces de services fournis et requis qui sont exportés est basée sur le nom des interfaces de service. Si aucun espace de résolution n'est donné, l'instance est créée dans l'espace global et les interfaces exportées sont ignorées. Le retrait d'une instance de l'intérieur d'un espace se fait à travers la destruction de l'instance.

7.3.3.3 Registres de services

La réalisation du concept d'espace de résolution nécessite de l'existence de plusieurs registres de services. Le registre de services de l'espace de résolution global correspond

au registre de services de la plate-forme OSGi.

La création des registres correspondant aux espaces de résolution est simplifiée grâce a classe `Filter`[3] de la plate-forme de services OSGi. Cette classe encapsule la mécanique autour des filtres LDAP et permet de tester si un filtre coïncide avec un ensemble de propriétés. Cette mécanique est la partie la plus compliquée du registre de services, le reste étant essentiellement la gestion de la liste de services enregistrés et des notifications.

7.3.3.4 Espaces de résolution et OSGi

Le concept d'espace de résolution n'est compatible avec la plate-forme de services OSGi que par rapport aux services présents dans l'espace global, les autres services étant inaccessibles depuis le registre de services OSGi.

Ce concept n'a pas d'équivalent dans la plate-forme OSGi, cependant, il pourrait être utilisé dans cette plate-forme de services non seulement pour limiter l'imprévisibilité mais aussi pour réaliser l'isolation de certains services dont l'accès publique n'est pas souhaitable. Un exemple de ceci serait un système d'alarme ; tous les services internes au système pourraient être créés à l'intérieur d'un espace de résolution et seulement un service de gestion du système d'alarme serait exporté vers l'espace global. Ainsi, les services internes au système d'alarme ne seraient pas accessibles publiquement. Le concept le plus proche à ceci dans la plate-forme OSGi est celui de la restriction d'accès aux services à partir des mécanismes de sécurité de la plate-forme. Ces mécanismes ne permettent cependant pas de limiter l'imprévisibilité lors de la présence d'instances dynamiques multiples.

7.3.4 Solution du problème de l'exemple

La figure 7.7 présente une solution au problème d'imprévisibilité décrit auparavant[4]. Dans cette figure, trois espaces de résolution sont présents. L'espace global, dans lequel sont enregistrés les services correspondant aux instances de déploiement, qui contient deux espaces à l'intérieur desquels sont créées les deux configurations de l'outil VersaTest. Dans cet exemple, l'instance de composant BootstrapComponent, qui est une instance de déploiement, est responsable de la création des espaces de résolution ainsi que des instances formant les applications.

7.3.5 Synthèse

Le concept d'espace de résolution permet de limiter l'imprévisibilité au moment de la création d'instances dynamiques multiples. La possibilité de création de hiérarchies d'es-

[3] voir http ://ivadmin.vwh.net/devzone/members/library/javadoc/org/osgi/framework/Filter.html
[4] Dans la figure, certains liens d'instantiation ont été omis pour simplifier.

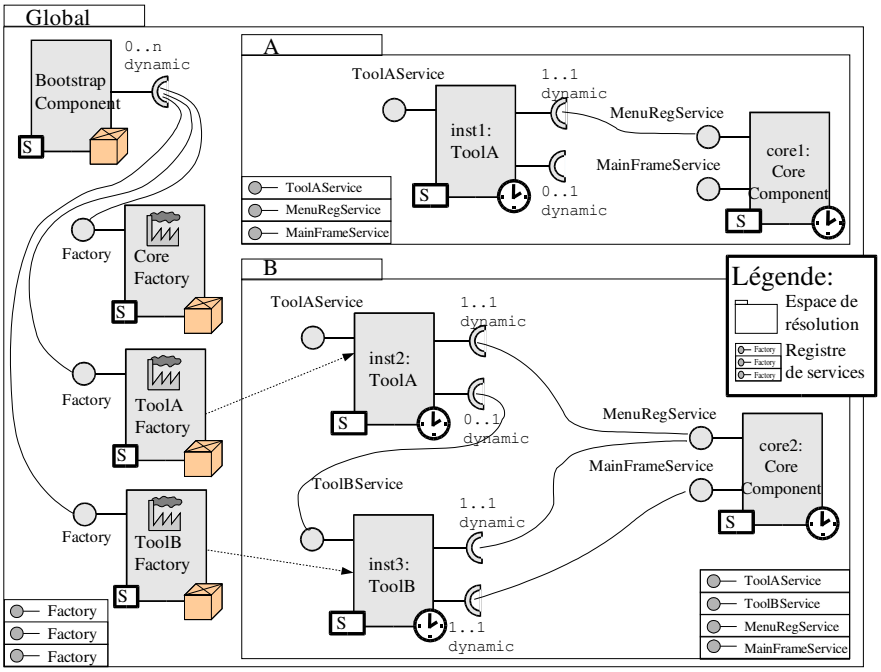


FIG. 7.7 – Instances créées à l'intérieur d'espaces de résolution

paces de résolution permet de réaliser des compositions hiérarchiques en temps d'exécution.

7.4 Gestion de compositions

Les exemples décrits précédemment supposent l'existence d'une instance de composant qui est liée aux fabriques et qui est responsable de créer, à la fois, les espaces de résolution ainsi que les instances qui sont créées à l'intérieur de ces espaces. Ce composant est représenté par le BootStrapComponent dans la figure 7.7.

La disponibilité dynamique des composants à services implique cependant que, pendant l'exécution, les fabriques de ces composants peuvent être ajoutées ou retirées du système (comme résultat des opérations *insert* et *remove*, réalisées par un acteur externe). Lors du retrait d'une fabrique, les instances créées à partir de celle-ci sont détruites, ce qui peut entraîner des invalidations "en chaîne" (voir 5.5.6) à l'intérieur des différents espaces de résolution dans lesquels se trouvaient les instances retirées.

L'instance de composant responsable de la création doit gérer ces situations. Elle peut, par exemple, décider de créer des nouvelles instances pour substituer les instances détruites, à partir d'autres fabriques, pour tenter de restituer les compositions présentes dans les différents espaces de résolution. Cette instance peut aussi tenter de créer des instances à partir de fabriques qui deviennent disponibles après la création d'une composition. Ces

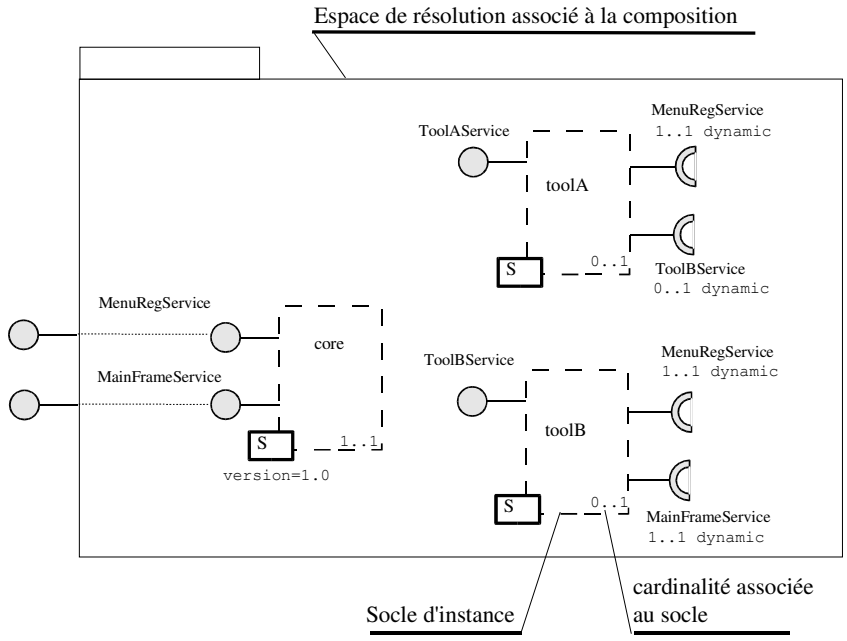


FIG. 7.8 – Représentation schématique d'un descripteur de composition

activités représentent des adaptations qui sont réalisées par l'instance de composant par rapport aux reconfigurations réalisées par son gestionnaire. Cette adaptation est comparable à celle que réalise un gestionnaire d'instance par rapport à la présence de services à travers la création et destruction de connexions. Dans ce cas, cependant, l'adaptation doit être faite par rapport à la présence de fabriques de composants à services à travers la création d'instances de composants.

7.4.1 Principes

Cette section décrit les principes permettant de décrire et de gérer une composition d'instances. Ces principes incluent le descripteur de composition et le gestionnaire de composition.

7.4.1.1 Socles d'instance et descripteur de composition

Lors de la création d'une composition d'instances à l'intérieur d'un espace de résolution, un certain degré d'évolution de la composition pendant l'exécution est nécessaire pour supporter les changements résultant de la disponibilité dynamique. Cette évolution concerne la possibilité de substitution, incorporation et retrait d'instances de composant de l'espace de résolution.

Pour supporter cette évolution, une composition est décrite comme un ensemble de

socles (placeholder). Un socle représente un nombre variable d'instances qui seront créés (à partir de fabriques) et gérées à l'intérieur de l'espace de résolution correspondant à la composition au moment de l'exécution. Un socle est décrit à partir des éléments présents dans la vue externe d'une instance de composant à services. Ces éléments incluent les interfaces de services fournies, requises et les propriétés de service.

Un socle est associé à une cardinalité. Cette cardinalité reprend les éléments de la cardinalité définie au niveau des instances de service requise, elle permet de définir si un socle représente une instance optionnelle ou obligatoire ainsi que la multiplicité de cette instance, qui peut être simple ou multiple. La cardinalité permet de supporter l'incorporation et le retrait d'instances de composant de l'espace de résolution pendant l'exécution. Finalement, des instances de services fournies et requises par des socles peuvent être marquées comme étant exportées.

La figure 7.8 présente de façon graphique les éléments contenus dans un descripteur de composition. Cette composition permet de créer une configuration du client VersaTest avec un noyau et deux outils. Dans cet exemple, le noyau est considéré comme étant obligatoire tandis que les deux outils sont considérés comme étant optionnels. Les services fournis par le noyau sont exportés.

7.4.1.2 Critère de correspondance

Le critère permettant de décider si une instance correspond à un socle est un critère simple. Toute instance qui fournit et requiert les mêmes interfaces de services est acceptée. Dans le cas ou une instance fournit des services additionnels à ceux déclarés au niveau du socle, l'instance est acceptée. Dans le cas ou une instance requiert des services additionnels à ceux déclarés au niveau du socle, l'instance est acceptée seulement si les services additionnels sont catégorisés comme étant optionnels. Les services fournis ou requis additionnels sont ignorés au moment de la création de connexions. Cette décision évite la création de connexions non envisagées ou indésirables dans la composition.

7.4.1.3 Gestionnaire de composition

La création d'une composition suit une approche similaire à celle proposée pour les instances de composant. Un gestionnaire de composition est responsable de créer et de faire évoluer une composition à partir du descripteur. De façon similaire aux instances de composants, les compositions peuvent être valides ou invalides. Une composition est valide si toutes les instances obligatoires sont présentes. La validité d'une composition n'implique cependant pas que les instances qu'elle contient sont valides. Une composition est gérée suivant un cycle de vie identique à celui présenté dans la section 5.4.3.

Configuration

L'étape de configuration correspond à la localisation de fabriques correspondant aux socles déclarés dans le descripteur. Une fois que toutes les fabriques correspondant aux socles obligatoires sont localisées, des instances de composants à services sont créées. Une fois qu'une composition est valide, le gestionnaire de composition entre alors dans la phase d'exécution.

Exécution

Pendant cette phase, le gestionnaire de composition reçoit des notifications concernant l'arrivée et le départ de fabriques. Si une fabrique dont les instances peuvent être utilisées dans la composition devient disponible, le gestionnaire de la composition peut alors décider de créer une nouvelle instance correspondante à un socle, si la cardinalité de celui-ci le permet.

Si une fabrique devient indisponible, ses instances sont détruites et donc retirées de l'espace de résolution correspondant à la composition de façon automatique. Dans le cas ou une instance obligatoire est détruite, le gestionnaire de composition cherche à créer une instance remplaçante. Si ceci n'est pas possible, la composition est invalidée, dans quel cas toutes les instances qu'elle contient sont détruites.

Après l'invalidation de la composition, le gestionnaire de composition entre dans un nouveau cycle de configuration de la composition. Comme le gestionnaire d'instance, le gestionnaire de composition tente de maintenir la validité de la composition qu'il gère.

7.4.2 Réalisation

Cette section décrit la réalisation du gestionnaire de composition.

7.4.2.1 Descripteur de composition

La DTD ainsi qu'un exemple de descripteur de composition sont montrés dans la figure 7.9 ; cet exemple décrit la création d'une composition contenant un navigateur web et un ensemble de plug-ins. Dans cet exemple là, la composition contient une instance unique du navigateur (cardinalité `1..1`), ainsi qu'un ensemble variable d'instances d'extensions optionnelles (cardinalité `0..n`). Une interface de service fournie par le navigateur est exportée en dehors de l'espace de résolution.

Un descripteur de composition est déployé de la même manière qu'un composant à services à l'intérieur d'un paquetage.

DTD

```
<!ELEMENT composition (exports*,property*,placeholder+)>
  <!ATTLIST composition
    factory (yes|no) "no"
  >
<!ELEMENT exports EMPTY>
  <!ATTLIST exports -- services exportés --
    service CDATA #REQUIRED -- nom du service --
    type (provided|required) #REQUIRED -- fourni ou requis --
    from IDREF #REQUIRED -- a partir de quelle instance --
  />
<!ELEMENT property EMPTY>
  <!ATTLIST  property
    name CDATA #REQUIRED
    type CDATA #REQUIRED
    value CDATA #REQUIRED
  >
<!ELEMENT placeholder (provides*,requires*)>
  <!ATTLIST placeholder
    id ID #REQUIRED
    filter CDATA #REQUIRED
    cardinality (0..1|0..n|1..1|1..n) #REQUIRED
  >
<!ELEMENT provides EMPTY>
  <!ATTLIST  provides
    service CDATA #REQUIRED
  >
<!ELEMENT requires EMPTY>
  <!ATTLIST  requires
    service CDATA #REQUIRED
    filter CDATA #REQUIRED
    cardinality (0..1|0..n|1..1|1..n) #REQUIRED
    policy (static|dynamic) #REQUIRED
  >
```

Exemple

```
<composition factory="yes">
  <exports
    service="org.examples.services.Application"
    type="provided"
    from="core"
  >
  <placeholder id="core"
    filter="(name=WebBrowserCore)"
    cardinality="1..1">
      <provides service="org.examples.services.Application"/>
      <requires service="org.examples.services.BrowserPlugin"
        filter=""
        cardinality="0..n"
        policy="dynamic"
      />
  </placeholder>
  <placeholder id="plugin"
    filter=""
    cardinality="0..n"
  >
    <provides service="org.examples.services.BrowserPlugin"/>
  </placeholder>
</composition>
```

FIG. 7.9 – Descripteur de composition

7.4.2.2 Gestionnaire

Le code correspondant au gestionnaire de composition est contenu dans bundle du ServiceBinder. Au moment ou la balise composition est trouvée dans un fichier descripteur, un gestionnaire de composition est créée. Pour supporter la localisation de fabriques, des informations relatives aux services fournis et requis par les instances de composant créées par une fabrique sont rajoutés comme des propriétés de service au niveau de la fabrique.

7.4.3 Synthèse

Le gestionnaire de composition fournit un moyen de créer et de maintenir une composition d'instances à l'intérieur d'un espace de résolution à partir d'un descripteur de composition. Ceci simplifie la tâche d'écriture d'un composant dédié à ces activités. Le descripteur de composition permet de plus de supporter un degré partiel d'évolution dans la composition pendant l'exécution.

7.5 Évaluations

Les propositions qui ont été présentées dans ce chapitre ont été évaluées à l'intérieur du laboratoire à l'aide de prototypes qui sont présentés à la suite.

7.5.1 Environnement de conception et d'exécution

Pour évaluer et expérimenter autour du concept de fabrique, un prototype d'environnement de conception et d'exécution d'applications orientées utilisateur (ayant une interface graphique) a été réalisé. Le but de cet environnement était de permettre de réaliser la construction d'applications à base de composants à services de façon visuelle, et de supporter l'exécution de ces applications, particulièrement au moment où un composant visuel devient indisponible. Pour cela, l'environnement d'exécution permet d'alterner entre un mode *conception* et un mode *exécution*.

Le concept de construction d'applications de façon visuelle diffère dans cet environnement de ce qui est réalisée dans des environnements tels que la BeanBox (voir 3.1.4), dans ce dernier des instances de composant sont créées puis configurées et connectées graphiquement. Dans l'environnement d'assemblage, un utilisateur crée des instances de composant, à partir d'un catalogue de composants, en réalisant du *glisser-déposer* ("drag-and-drop") depuis le catalogue dans une zone d'assemblage et les instances de composant ainsi créées sont connectées par le ServiceBinder automatiquement. Cette technique de construction implique bien sûr que l'utilisateur est au courant des dépendances entre les composants et est plutôt orientée à faciliter la disposition des composants visuels.

7.5.1.1 Scénario d'utilisation

La figure 7.10 présente un scénario d'utilisation de cet environnement. Le scénario présente initialement l'environnement dans un mode de conception (figure A). La partie gauche de l'environnement montre une catalogue de composants (qui est obtenu à partir des fabriques enregistrées à un moment donné) et la partie droite montre la zone d'assemblage (qui dans cette capture d'écran est vide).

La figure B présente l'environnement dans le mode de conception après que diverses instances de composant aient été créées à travers des opérations de glisser-déposer à partir du catalogue. L'environnement de conception permet de changer les propriétés des instances à travers un mécanisme d'introspection (de façon similaire à ce qui est réalisé dans les JavaBeans). Au fur et à mesure que les instances sont créées, leurs connexions le sont aussi.

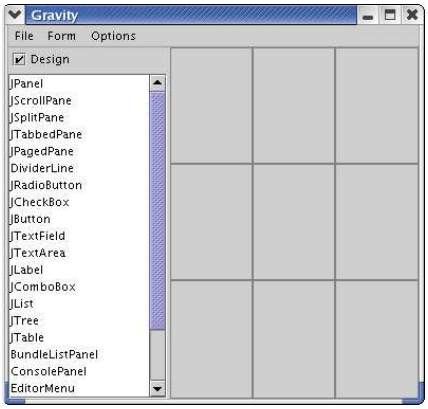
La figure C présente l'environnement dans le mode exécution. A ce moment là, le catalogue de composants n'est plus présent. L'exemple construit dans l'environnement est un éditeur de texte qui est assemblé à partir de plusieurs composants fournissant diverses fonctionnalités telles qu'un menu, un éditeur, un buffer multiple (permettant de changer entre divers documents dans l'éditeur) et un explorateur de fichiers. Les instances de ces composants sont connectées entre elles.

La figure D présente une adaptation de l'application par rapport au départ d'un composant qui réalise le rendu de l'explorateur de fichiers. La dépendance entre l'explorateur de fichiers et le service réalisant le rendu est de type `1..1` dynamique, donc au départ du service de rendu utilisé, le ServiceBinder cherche un remplaçant. Cette substitution est visible dans cette figure car le rendu du système de fichiers est réalisé de façon différente.

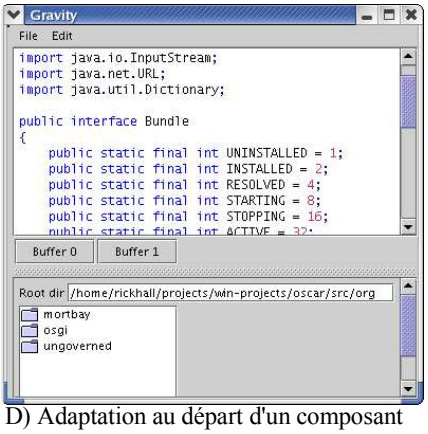
La figure E présente le résultat du départ du deuxième service de rendu. A ce moment là, la dépendance du composant explorateur de fichiers n'est plus valide et l'instance est invalidée. L'environnement d'exécution remplace l'explorateur de fichiers par une icône signifiant que l'instance a été invalidée. Le reste de l'éditeur continue à fonctionner normalement car la dépendance entre l'éditeur et l'explorateur de fichier est de type `0..1` dynamique. Finalement la figure F présente le résultat de l'arrivée du service de rendu.

7.5.1.2 Discussion

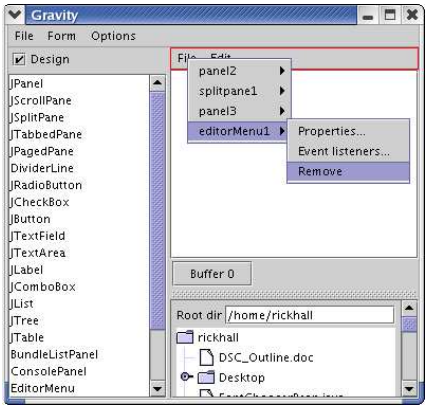
Un aspect intéressant de ce prototype est l'idée d'assembler des applications en réalisant du glisser-déposer sans devoir créer des connexions explicites entre les composants. Cette approche peut paraître peu conventionnelle, car elle suppose que l'assembleur de l'application a une connaissance sur les services qui sont requis et fournis par les instances de composant qu'il utilise. Cependant, il pourrait être envisagé que l'environnement de conception suggère à l'utilisateur, au moment où il décide de créer l'instance d'un composant, des fabriques à partir desquelles il pourrait créer d'autres instances qui



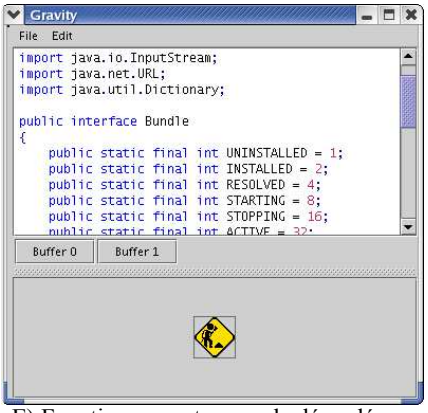
A) Environnement de conception



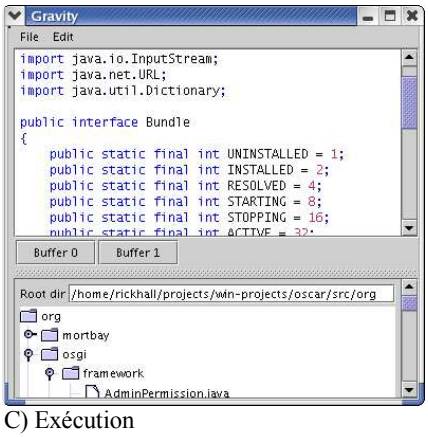
D) Adaptation au départ d'un composant



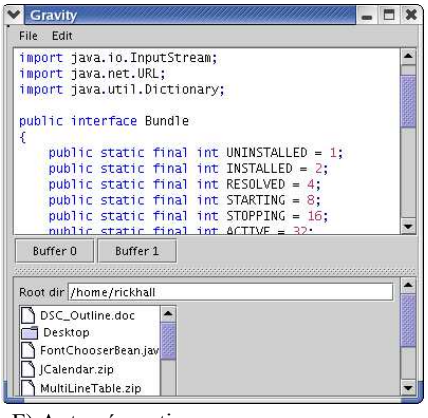
B) Configuration des instances



E) Fonctionnement en mode dégradé



C) Exécution



F) Auto-réparation

FIG. 7.10 – Environnement de conception et exécution

seraient connectées avec l'instance qui vient d'être été créée. L'environnement pourrait réaliser ces suggestions en interrogeant les fabriques des composants enregistrées pour savoir si les instances de ces composants fournissent des services pertinents. D'un autre côté, cet environnement gère de façon très simple l'invalidation d'une instance de composant, en plaçant un icône qui représente le fait que le gestionnaire tente de valider l'instance. D'autres recherches pourraient être réalisées autour de cette problématique pour réaliser des interfaces utilisateur *plastiques*, c'est à dire qui peuvent s'adapter face aux changements dans l'application tout en préservant l'usabilité [GCT01].

Dans ce prototype, le concept d'espace de résolution n'a pas été utilisé bien que son utilité devient évidente au moment où plusieurs instances de composants connectées entre elles sont créées. Une question qui se pose concerne la possibilité de créer des espaces de résolution de façon automatique, par exemple par rapport à l'emplacement où sont déposés les composants au moment de la création d'une instance. L'introduction du concept d'espace de résolution d'environnement pourrait être couplée avec un générateur de descripteurs de compositions permettant de sauvegarder l'assemblage.

Finalement il faut remarquer que ce type d'environnements pourrait être employé pour d'autres domaines que ceux des applications supportant l'intéraction. Un environnement de ce type pourrait permettre à un administrateur d'introduire des composants de supervision à travers le glisser-déposer, dans une application comme celle qui a été présentée dans la section 6.8.

7.5.2 Gestionnaire de composition

Pour expérimenter autour des concepts d'espace de résolution et de gestion de composition, le gestionnaire de composition a été implémenté ainsi qu'un prototype simple permettant de réaliser des tests. Dans ce prototype, un lanceur d'applications permet de créer différentes instances d'applications qui sont déployées dans la plate-forme suivant une technique similaire à celle qui a été proposée pour le projet de télévision interactive. La composition utilisée est similaire à celle présentée dans la figure 7.9, et une application qui a été développée est un navigateur web extensible à base de plug-ins. La figure 7.11 présente des captures d'écran montrant l'intégration d'un plug-in permettant de visualiser des fichiers PDF pendant l'exécution du navigateur. La figure A montre le lanceur d'application avec un menu qui montre la liste des applications disponibles ainsi qu'une instance de la composition qui donne lieu au navigateur. Dans la figure B, un lien dirigé sur un fichier PDF est choisi, et comme aucun plug-in permettant de gérer ce format n'est disponible à ce moment là, un message d'erreur est affiché. La figure C montre le même navigateur web après qu'un composant à services offrant un service de plug-in permettant d'afficher des PDF est installé pendant l'exécution de l'application. A ce moment là, le choix du même lien suivi précédemment résulte dans l'affichage correct du fichier par le

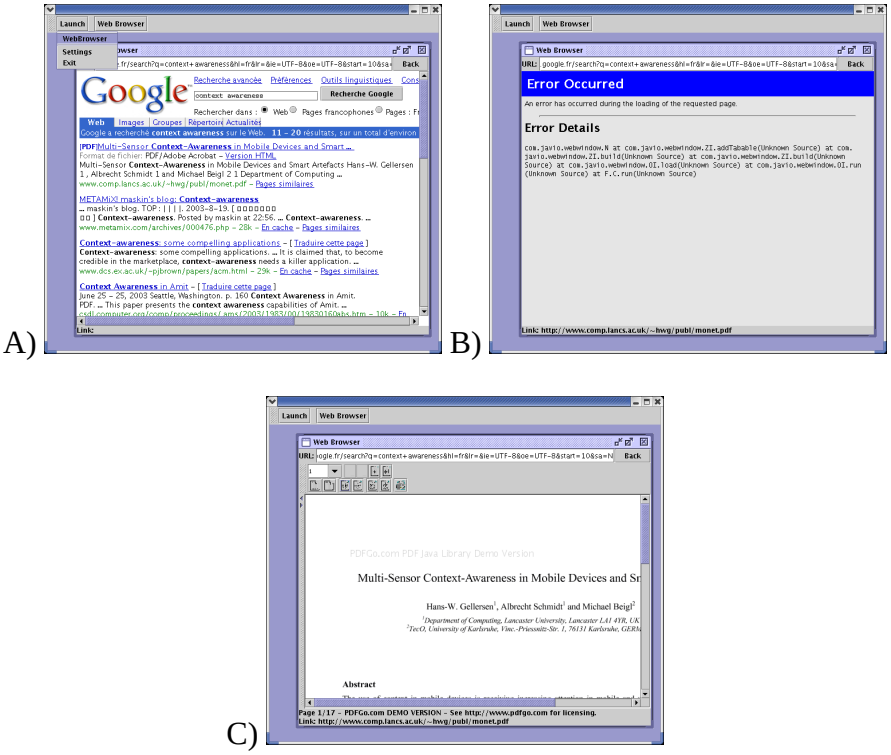


FIG. 7.11 – Prototype de test des espaces de résolution et gestionnaire de composition

plug-in. Cet exemple est très simple et n'a seulement servi que pour tester le gestionnaire de compositions.

7.6 Conclusion

Ce chapitre a présenté les concepts nécessaires à la construction d'applications à partir d'instances de composants à services dynamiques. Ces concepts incluent notamment les fabriques et les espaces de résolution. Bien que les fabriques et les espaces de résolution permettent de créer des instances dynamiques et de limiter l'imprévisibilité au niveau de leurs connexions, il reste à la charge du programmeur de gérer la création et le maintien de compositions par rapport à la disponibilité dynamiques des classes de composants à services.

Ce problème est traité à partir de la proposition d'un mécanisme cherchant à faciliter la création et la gestion de compositions à partir de descripteurs. Le gestionnaire de composition est similaire au gestionnaire d'instances, cependant le premier est chargé de gérer des instances de composants à services par rapport à des socles au lieu de connexions par rapport à des interfaces de services requises.

Finalement des prototypes réalisés au sein du laboratoire pour évaluer les concepts présentés dans ce chapitre ont été décrits. Un environnement d'assemblage et d'exécution

a été réalisé pour tester le concept de fabrique ainsi que pour étudier la possibilité de construire des applications à partir de composants interactifs de façon visuelle. Par rapport au gestionnaire d'instances, un prototype simple permettant de tester le gestionnaire a été réalisé.

Troisième partie

Conclusions et Perspectives

Chapitre 8

Conclusions et Perspectives

8.1 Synthèse

Ce document a présenté les travaux qui ont été réalisés au long de la thèse pour introduire et supporter la disponibilité dynamique dans un modèle de composants. Les points principaux de ce travail sont résumés à la suite.

Étude des approches à composants et à services

Dans la première partie de cette thèse, une étude des approches à composants et à services a été présentée. Cette étude représente non seulement un état de l'art mais aussi une contribution qui a cherché à présenter les concepts essentiels de chacune des deux approches en restant éloignés d'une technologie particulière. Cette étude a été réalisée à partir des concepts disponibles dans un ensemble de modèles à composants et de plateformes de services existant actuellement. Une description et une comparaison de ces modèles à composants et plateformes de services a ensuite été réalisée par rapport aux concepts identifiés. Cette étude permet d'apprécier les points de coïncidence et de divergence des deux approches.

Modèle à composants orienté services

La deuxième partie de cette thèse a traité les aspects relatifs à l'introduction et au support de la disponibilité dynamique dans un modèle à composants. La disponibilité dynamique représente le fait qu'à tout moment pendant l'exécution d'une application construite à base de composants, des classes ou des instances de composants peuvent être introduits ou retirés dans le système où s'exécute une application, potentiellement par un acteur autre que l'application. Du fait que ces activités peuvent avoir lieu en dehors du contrôle de l'application, la disponibilité dynamique nécessite que les applications soient capables de s'adapter de façon autonome par rapport aux changements.

Ceci est pris en charge à travers l'introduction de concepts de l'approche à services dans le modèle à composants, ainsi qu'à partir de la gestion, par l'environnement d'exécution du modèle à composants, d'une logique d'adaptation qui est configurée par rapport à des informations associées aux composants.

Composants à services

Un composant à services fournit et requiert des interfaces de service et contient des propriétés de service. Les interfaces de service fournies ainsi que les propriétés de service sont utilisées pour publier les instances des composants à services dans un registre de services. Les interfaces de service requises, présentes lorsque l'implémentation du composant contient une composition de services, sont caractérisées par des informations qui sont la cardinalité, la politique et un filtre. La cardinalité définit le nombre de connexions qui peuvent être réalisées vers une interface de services requise ainsi que le fait que l'existence d'une connexion soit obligatoire ou non. La politique définit si les connexions peuvent changer ou non pendant l'exécution et le filtre permet de contraindre la réalisation de connexions vers des sous-ensembles de fournisseurs. Ces informations définissent la manière suivant laquelle une instance du composant est adaptée par rapport à des changements dans les instances qui fournissent des services.

Une fois créée, une instance de composant à services se trouve dans un de deux états : valide ou invalide. Lorsqu'elle est valide, ses services sont enregistrés dans le registre de services et sa logique s'exécute. Lorsqu'elle est invalide, ses services ne sont pas enregistrés et sa logique ne s'exécute pas.

Gestion de l'adaptation

Les informations associées aux interfaces de service requises configurent une logique d'adaptation qui fait partie de l'environnement d'exécution du modèle, qui est bâti sur une plate-forme de services. Cette approche permet de séparer le code applicatif, qui se trouve à l'intérieur des composants, du code adaptatif.

Pendant l'exécution, chaque instance de composant à services est placée dans un gestionnaire d'instance qui, d'une façon similaire à un conteneur dans un modèle classique, gère le cycle de vie de l'instance, ce qui inclut la réalisation de reconfigurations et le maintien de la validité. La reconfiguration de l'instance gérée est réalisée à travers la création et la destruction de connexions (opérations *bind* et *unbind* de la reconfiguration dynamique). L'adaptation réalisée par le gestionnaire est réalisée par rapport à la supervision de changements ayant lieu dans le registre de services qui concernent l'arrivée ou le départ des services fournis par les instances. Le maintien de la validité représente le fait que lors de l'échec d'une reconfiguration, le gestionnaire d'instance tente de configurer l'instance gérée à nouveau pour la faire retourner à l'état valide.

Le fait que chaque instance de composant soit gérée de façon indépendante résulte dans des applications capables de s'assembler et de s'adapter dynamiquement et de façon autonome.

Applications à base d'instances de déploiement

Le concept d'instance de déploiement représente une instance de composant singleton qui est créée et détruite au moment de l'introduction ou retrait d'un composant à services dans le système. Ce concept permet de construire un type d'application dans lequel la création d'instances multiples à partir d'un même composant à services n'est pas nécessaire. Des exemples de telles applications ont été obtenus à partir d'évaluations industrielles.

Applications à base d'instances dynamiques

Les instances dynamiques sont créées pendant l'exécution et de multiples instances peuvent être créées à partir d'un même composant à services. La création des instances dynamiques est réalisée à partir de fabriques, qui permettent d'introduire ou de retirer des instances de composants à services dans le système pendant l'exécution. Une fabrique retourne une poignée qui représente une instance, car la création d'une instance ne garantit pas qu'elle puisse être validée immédiatement.

La création d'instances dynamiques augmente la problématique d'imprévisibilité, qui représente le fait que certaines connexions peuvent être créées de façon aléatoire, du fait que toutes les instances partagent la même information au niveau des interfaces de service requises. Cette problématique est limitée à travers le concept d'espace de résolution, qui permet de contraindre la création de connexions entre instances de composants à l'intérieur d'un registre de services indépendant associé à chaque espace. Les espaces sont créés de façon hiérarchique, ce qui permet de créer des compositions hiérarchiques pendant l'exécution.

Pour simplifier la construction d'applications à base d'instance dynamiques, une proposition d'un moyen de décrire une composition de façon déclarative a été présentée. Dans cette proposition, une composition d'instances de composants à services est décrite par rapport à des socles, qui représentent des ensembles d'instances de composants à services vues à partir de leur vue externe. Pendant l'exécution, un gestionnaire de composition est chargé de trouver des fabriques de composant et de créer des instances à partir de ces fabriques à l'intérieur d'un espace de résolution.

Réalisation du modèle au dessus de OSGi

L'environnement d'exécution du modèle à composants orienté services, appelé ServiceBinder, a été réalisé au dessus de la plate-forme de services OSGi. Cette réalisation est

contrainte à la construction d'applications non-distribuées. Les concepts introduits par le modèle à composants orienté services facilitent de façon considérable la création d'applications dans la plate-forme OSGi ; ceci est particulièrement valide dans le cas de la construction d'applications à base d'instances de déploiement. Les surcoûts au niveau de la consommation de la mémoire introduits par le ServiceBinder ne sont pas excessifs et sont compensés par la simplification au niveau de l'écriture du code. Le ServiceBinder a été publié comme un projet de source ouverte qui a été testé en milieu industriel. Les retours obtenus des évaluations ont été très positifs.

8.2 Conclusions

Les travaux présentés dans ce document permettent de conclure que l'approche choisie pour introduire et supporter la disponibilité dynamique dans un modèle de composants est adéquate. Les apports principaux de cette thèse concernent d'un côté la combinaison des approches à composants et à services et les bénéfices apportés au niveau du développement d'applications pour la plate-forme OSGi.

Combinaison des concepts de l'approche à composants et à services

Un apport important de cette thèse concerne l'étude des approches à services et à composants ainsi que la proposition de moyens de combiner les deux approches. Les concepts présentés dans ce travail pourraient éventuellement être employés dans le domaine des service web, dans lequel il existe déjà des recherches, telles que [Bur00] et [Yan03], qui cherchent à combiner des concepts appartenant aux deux approches. La recherche dans les service web est cependant contrainte par la lourdeur de l'infrastructure, ce qui rend difficile l'implémentation des concepts. L'utilisation d'une plate-forme de services 'légère' permet de tester des concepts plus facilement. Il serait cependant intéressant d'implémenter le modèle à composants orienté services au dessus d'une plate-forme de services distribuée, telle que Jini, pour supporter la création de systèmes distribués.

Construction d'applications dans la plate-forme OSGi

Un deuxième apport important de cette thèse concerne les bénéfices qu'apporte le ServiceBinder par rapport au développement d'applications pour la plate-forme OSGi. Au fur et à mesure où cette plate-forme de services deviendra plus populaire, il est possible que des problématiques traitées dans cette thèse seront rencontrées, par exemple la limitation de l'existence d'un registre unique, ou le besoin de création d'instances dynamiques.

La popularité de la plate-forme de services est en plein essor et son utilisation ne se limite plus aux environnements restreints, car elle vient d'être incorporée à l'environnement de développement Eclipse de IBM comme une infrastructure de déploiement des

plug-ins. L'utilisation d'OSGi en dehors du domaine d'application original pour lequel il a été conçu est une idée que le Dr. Hall et l'auteur de cette thèse partagent depuis un certain nombre d'années. Le fait de voir OSGi employé dans Eclipse confirme que cette vision était correcte.

Un souhait du Dr. Hall et de l'auteur serait de pouvoir avoir une influence sur les gens qui définissent la spécification de la plate-forme de services OSGi. Pour cela, les travaux présentés dans cette thèse sont promus de façon active.

8.3 Perspectives

Diverses perspectives de ce travail sont présentées à la suite.

8.3.1 Résolution des problèmes techniques

Parmi mes problèmes techniques, les plus importants concernent ceux relatifs à la préservation de la consistance dûs à la reconfiguration dynamique (voir 5.2.1.1). La réalisation du transfert de l'état ainsi que l'interruption des appels entrants vers une instance sont indispensables, cependant ces tâches doivent couramment être réalisées à travers la programmation. Il serait désirable que l'environnement d'exécution du modèle à composants puisse prendre en charge une partie ou la totalité de ces tâches. Bien que l'interruption des appels entrants peut être réalisée de façon relativement simple, à travers l'introduction d'intermédiaires, une solution performante doit être recherchée du fait qu'il est désirable de pouvoir utiliser l'environnement d'exécution du modèle à composants dans des environnements restreints. Pour le transfert d'état il est nécessaire de définir un moyen de décrire l'état qui doit être transféré par l'environnement d'exécution.

8.3.2 Environnement d'exécution extensible

Une idée qui mérite d'être explorée est celle de rendre l'environnement d'exécution du modèle à composants orienté services une entité extensible qui utilise elle même des services. L'environnement d'exécution pourrait alors être réduit à un noyau de petite taille pouvant être adapté à divers contextes d'utilisation à travers l'ajout d'extensions. Ces extensions pourraient d'ailleurs être déployées dans la plate-forme au même titre que les composants à services standard et pourraient être incorporées dans l'environnement d'exécution à partir de services fournis par ces composants. Des points qui ont été identifiés comme pouvant accommoder des services d'extension du noyau sont les suivants :

Parseur Couramment les descripteurs de composants à services sont décrits dans une syntaxe XML, et un parseur XML de taille réduite est inclus dans le ServiceBinder. Un service de parsing pourrait être fourni par un composant externe

ce qui permettrait de pouvoir réaliser la description des composants dans des syntaxes différentes.

Filtrage Comme il a été mentionné dans la section 6.4.3.3, le ServiceBinder contient un algorithme très simple pour résoudre la problématique du filtrage côté client. Des algorithmes plus sophistiqués pourraient être offerts à travers des services qui seraient invoqués au moment où plusieurs réponses sont retournées par le registre de service. Un algorithme de filtrage côté client pourrait par exemple se guider par rapport à une description architecturale 'globale' de l'application.

Téléchagement à la demande Une extension possible pourrait concerner un service permettant de télécharger et d'installer des composants à services lorsqu'aucune connexion ne peut être réalisée et que l'interface de service requise est qualifiée comme étant obligatoire. Ce service pourrait contacter un catalogue et à partir du service demandé obtenir un composant adéquat. Cette extension est, à priori, facilement réalisable du fait que OSGi permet d'accéder aux mécanismes de gestion des bundles à travers le contexte.

Aspects non fonctionnels Le gestionnaire d'instance pourrait être utilisé pour introduire du support d'aspects non-fonctionnels dans la plate-forme, notamment la distribution, dans le cas où le modèle serait implémenté au dessus d'une plate-forme de services distribuée. L'utilisation de services pour réaliser les aspects non-fonctionnels pourrait être envisagée.

8.3.3 Construction d'applications sensibles au contexte

Une vision a long terme qui à servi de motivation dans la réalisation de ce travail est la construction d'applications sensibles au *contexte*, qui dans [DA00] est défini comme étant "toute information qui peut être utilisée pour caractériser la situation d'une entité".

La vision de la manière de réaliser de telles applications est présentée dans la figure 8.1. Cette figure montre un système (à droite, machine virtuelle Java avec OSGi et ensemble de bundles) qui contient un ensemble d'instances de composants à services connectées entre elles. Le 'nuage' représente un acteur qui utilise l'information contextuelle et agit sur le système en dirigeant des activités de déploiement, par exemple l'installation, retrait ou mise à jour de composants, ainsi que la création et destruction d'instances de composants et d'espaces de résolution. Cet acteur peut par exemple être contenu dans un paquetage et être déployé dans le système, ce qui lui permet d'interagir avec les fabriques ou avec la plate-forme OSGi pour administrer les paquetages de composants.

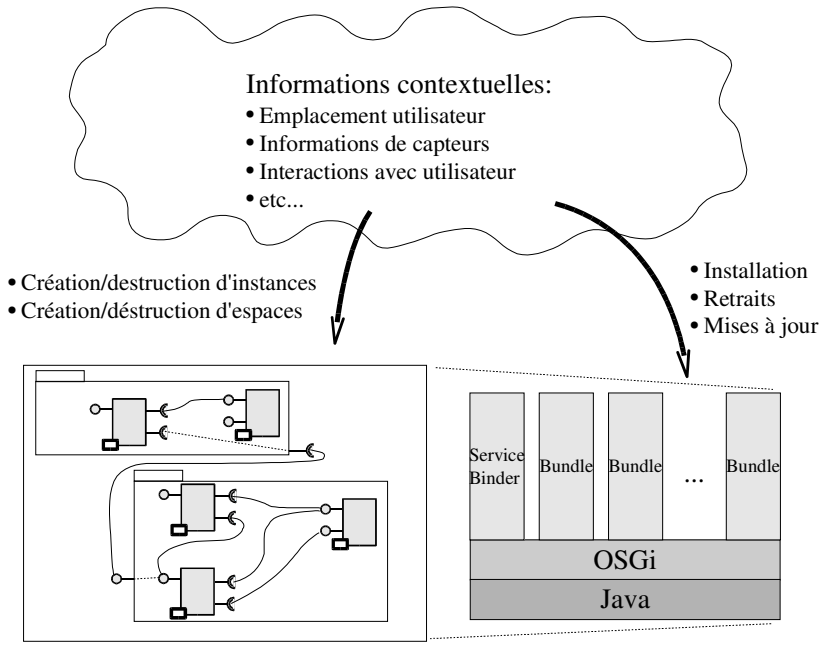


FIG. 8.1 – Applications sensibles au contexte

Quatrième partie

Bibliographie et Annexes

Bibliographie

[AA01] J. Almeida et Al. An approach to dynamic reconfiguration of distributed systems based on object-middleware. Dans *Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC)*, 2001.

[ACN02] J. Aldrich, C. Chambers, et D. Notkin. Archjava : Connecting software architecture to implementation. Dans *Proceedings of ICSE*, Mai 2002.

[AF01] L.F. Andrade et J.L. Fiadeiro. Coordination technologies for web-services. Dans *Workshop on Object-Oriented Web Services (at OOPSLA)*, 2001.

[AGD97] R. Allen, D. Garlan, et R. Douence. Specifying dynamism in software architectures. Dans *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Suisse, Septembre 1997.

[AN00] F. Achermann et O. Nierstrasz. Applications = components + scripts - a tour of piccola. Dans Mehmet Aksit, editeur, *Software Architectures and Component Technology*. Kluwer, 2000, 2000.

[AOWW99] K. Arnold, B. O'Sullivan, R. W. Waldo, et A. Wollrath. *The Jini Specification*. Addison-Wesley, Reading, Mass., 1999.

[BA00a] F. Bachman et Al. Volume ii : Technical concepts of component-based software engineering. Rapport technique ESC-TR-2000-007, Software Engineering Institute, Mai 2000.

[BA00b] L. Bass et Al. Volume i : Market assessment of component-based software engineering. Rapport technique CMU/SEI-2001-TN-007, Software Engineering Institute, Mai 2000.

[BAKR95] L. Bellisard, S.B. Atallah, A. Kerbrat, et M Riveill. Component Based Programming And Application Management with OLAN. Dans *Proceedings of Wokshop on Object-based Parallel and Distributed Computation*, 1995.

[Bar98] J.G.P. Barnes. *Programming in Ada*. Addison-Wesley, second edition, 1998.

[BBRVD98] R. Balter, L. Bellissard, M. Riveill, et J.-Y. Vion-Dury. Architecting and Configuring Distributed Applications with OLAN. Dans *Proceedings de IFIP International Conference on Distributed Systems Platforms and Open*

Distributed Processing (Middleware'98), pages 241–256. Springer, Septembre 1998.

[BC01] G. Bieber et J. Carpenter. Introduction to service-oriented programming (rev 2.1). Document en ligne, Avril 2001. http ://www.openwings.org/download.html.

[BCS02] E. Bruneton, T. Coupaye, et J.B. Stefani. *The Fractal Composition Framework Version 1.0*. Object Web Consortium, Juillet 2002.

[Bir01] D. Birngruber. A software composition language and its implementation. *Perspectives of System Informatics*, Juillet 2001.

[Box98] D. Box. *Essential COM*. Addison-Wesley, Janvier 1998.

[Bur00] S. Burbeck. Evolution of web applications into service-oriented components with web services. Document en ligne, Octobre 2000. http ://www-106.ibm.com/developerworks/library/ws-tao/index.html.

[CA02] F. Curbera et Al. *Business Process Execution Language (BPEL) for Web Services, Version 1.0*, Juillet 2002.

[Cab98] L. Cable. *Extensible Runtime Containment and Services Protocol for JavaBeans Version 1.0*. Sun Microsystems, Decembre 1998.

[CD03] S. Chomat et D. Donsez. OSGiTV : une plate-forme de deploiement d'applications de television interactive basee sur OSGi. Dans *Proceedings de CFSE*, 2003.

[Cer00] H. Cervantes. Analyse de dependances et decoupe dans un logiciel de grande taille. These de Master, Universite Joseph Fourier, Grenoble, 2000.

[CFD02] H. Cervantes, J. M. Favre, et F. Duclos. Describing hierarchical compositions of java beans with the beanome language. Dans *Workshop on Software Composition (SC2002)*, 2002.

[CH00] H. Cervantes et R. S. Hall. Beanome : A component model for the osgi framework. Dans *proceedings of the Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices*, Lausanne, Suisse, Septembre 2000.

[Cle98] P. C. Clements. A survey of architecture description languages. Dans *Eighth International Workshop on Software Specification and Design*, Allemagne, 1998.

[CNW01] F. Curbera, A. Nagy, et S. Weerawarana. Web-services : Why and how. Dans *Workshop on Object-Oriented Web Services (in OOPSLA)*, Aout 2001. http ://www.research.ibm.com/people/b/bth/OOWS2001.html.

[Com03] ComponentSource. The evolution of components. Document en ligne, 2003. http ://www.componentsource.com/.

[CPFD01] D. Conan, E. Putrycz, N. Farcet, et M. DeMiguel. Integration of non-functional properties in containers. Dans *Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[DA00] A. Dey et G. Abouwd. Towards a better understanding of context and context-awareness. Workshop on the what, who, where, when and how of context awareness at CHI, 2000.

[Dav02] M. Davidson. The Bean Builder Tutorial. Document en ligne, 2002. http ://java.sun.com/products/javabeans/beanbuilder/1.0/docs/guide/tutorial.html.

[DC01] J. Dowling et V. Cahill. The k-component architecture meta-model for self-adaptive software. Rapport technique, Trinity College Dublin, Computer Science Department, 2001.

[DK76] F. DeRemer et H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2) :80–87, Juin 1976.

[DLB01] N. DePalma, P. Laumay, et L. Bellisard. Ensuring dynamic reconfiguration consistency. Dans *Proceedings of the Sixth International Workshop on Component Oriented Programming*, 2001.

[DVdHT02] E.M. Dashofy, A. Van der Hoek, et R. Taylor. Towards architecture-based self-healing systems. Dans *Proceedings of the Workshop on Self-Healing Systems (WOSS)*, 2002.

[GA95] E. Gamma et Al. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GCT01] Calvary. G, J. Coutaz, et D. Thevenin. Supporting context changes for plastic user interfaces : A process and a mechanism. *Proceedings of IHM-HCI*, 2001.

[Geo02] I. Georgiadis. *Self-Organising Distributed Component Software Architectures*. These de doctorat, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 2002.

[GMK02] I. Georgiadis, J. Magee, et J. Kramer. Self-organising software architectures for distributed systems. Dans *Proceedings of the first workshop on Self-healing systems*, pages 33–38. ACM Press, 2002.

[GMW00] D. Garlan, R. Monroe, et D. Wile. *ACME : Architectural Description of Component-Based Systems in Foundations of Component-Based Systems*. Cambridge University Press, 2000. pp. 47-68.

[Hal97] R. Hall. An architecture for post-development configuration management in a wide-area network. Dans *Proceedingos of the International Conference on Distributed Computing Systems*, Mai 1997.

[Hal99] R.S. Hall. *Agent-based Software Configuration and Deployment*. These de doctorat, University of Colorado, 1999.

[Hay03] B. Hayes. The post-oop paradigm. *American Scientist*, 91(2) :106–110, Mars 2003.

[HG98] G. Hjalmtysson et R. Gray. Dynamic c++ classes : A lightweight mechanism to update code in a running program. Dans *USENIX Annual Technical Conference (No 98)*, 1998.

[Hof93] C. R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. These de doctorat, Computer Science Department, University of Maryland, 1993.

[IBR93] J. Indulska, M. Bearman, et K. Raymond. A Type Management System for an ODP Trader. Dans *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing ICODP*, 1993.

[JF88] R. E. Johnson et B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2) :22–35, Juin/Juillet 1988.

[KBC02] A. Ketfi, N. Belkhatir, et P. Y. Cunin. Adaptation dynamique, concepts et experimentations. Dans *Proceedings of ICSSEA*, 2002.

[KC99] F. Kon et R. H. Campbell. Supporting automatic configuration of component-based distributed systems. Dans *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX, 1999.

[LMMG02] S. Leblanc, R. Marvie, P. Merle, et J.M. Geib. TORBA : contrats de courtage pour CORBA. *Les intergiciels, developpements recents dans CORBA, Java RMI et les agents mobiles*, pages 47–72, 2002.

[LR97] F. Leymann et D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1) :102, 1997.

[LSSG02] M. Lumpe, J-G. Schneider, B. Schonhage, et T. Genssler. Composition languages. Dans *Proceedings of the Second International Workshop on Composition Languages*, 2002.

[Luc96] D. Luckham. Rapide : A language and toolset for simulation of distributed systems by partial orderings of events. DIMACS Partial Order Methods Workshop IV, Princeton University, Juillet 1996.

[Mar01] R. Marvie. CODeX : proposition pour la description dynamique d'architectures a base de composants logiciels. Dans *Acte des Journées Composants*, Besancon, Octobre 2001. in French.

[MB00] M. Mattson et J. Bosch. Stability assessment of evolving industrial object-oriented frameworks. *Journal Of Software Maintenance : Research and Practice*, 12 :79–102, 2000.

[Med95] N. Medvidovic. Formal definition of the Chiron-2 software architectural style. Rapport technique UCI-ICS-95-24, Irvine, CA, USA, Aout 1995.

[MGG97] P. Merle, C. Gransart, et J.-M. Geib. Using and implementing CORBA objects with CorbaScript. Dans *Object Based Parallel and Distributed Computing Workshop*, Toulouse, Octobre 1997.

[MK96] J. Magee et .J. Kramer. Dynamic structure in software architectures. Dans *Proceedings of the Fourth Symposium of the Foundations of Software Architecture (FSE4)*, 1996.

[MM01] R. Marvie et P. Merle. CORBA Component Model : Discussion and Use with OpenCCM. *Special issue of the Informatica : "Component Based Software Deployment"*, 25(4), 2001.

[MN97] T. Meijler et O. Nierstrasz. Beyond objects : Components. Dans *Cooperative Information Systems : Current Trends and Directions*. Academic Press, 1997.

[MS98] L. Mikhajlov et E. Sekerinski. A study of the fragile base class problem. Dans *Object-Oriented Programming 12th European Conference, ECOOP*, pages 355–382, 1998.

[MT00] N. Medvidovic et R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–96, Janvier 2000.

[MW99] P. Milne et K. Walrath. Long term persistence for javabeans. Document en ligne, Novembre 1999. http ://java.sun.com/products/jfc/tsc/articles/persistence/.

[Nel91] G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[New01] J. NewMarch. *A Programmer's Guide to Jini Technology*. APress, 2001.

[OA99] P. Oreizy et Al. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, Mai/Juin 1999.

[Obj99] Object Management Group. Corba components : Joint revised submission. Aout 1999.

[Ope03] Open Services Gateway Initiative. *OSGi Service Platform (3d Release)*, Mars 2003.

[Ore96] P. Oreizy. Issues in the runtime modification of software architectures. Rapport technique, Department of Information and Computer Science, University of California, Irvine, 1996.

[OT98] P. Oreizy et R. N. Taylor. On the role of software architectures in runtime system reconfiguration. *IEE Proceedings-Software*, 145(5), Octobre 1998.

[Ous98] J.K. Ousterhout. Scripting : Higher-level programming for the 21st century. *Computer*, 31(3) :23–30, 1998.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), Decembre 1972.

[PDN86] R. Prieto-Diaz et J.M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4) :307–334, Novembre 1986.

[Pel03] C. Peltz. Web services orchestration and choreography. *Computer*, Octobre 2003.

[PF03] S.R. Ponnenkanti et A. Fox. Application-service interoperation without standardized interfaces. Dans *Proceedings IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2003.

[Pla02] D.S. Platt. *Introducing Microsoft .NET*. Second edition, 2002.

[RAJ02] E. Roman, S. Ambler, et T. Jewell. *Mastering Enterprise JavaBeans*. Wiley Computer Publishing, second edition, 2002.

[RS02] M. Riveill et A. Senart. *Coopération dans les systèmes à objets*, volume 8 of *RSTI - L'Objet*, chapter Aspects dynamiques des langages de description d'architecture logicielle. Hermes, 2002.

[San01] R. Sanlaville. *An Architectural Environment for Dassault Systemes*. These de doctorat, University of Grenoble, 2001.

[Sen00] A. Senart. Aspects dynamiques dans les architectures logicielles en environnement reparti. These de Master, Universite Joseph Fourier, 2000.

[SG96] M. Shaw et D. Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, Avril 1996.

[SLB00] D. Sevilla, M. Lacage, et D. C. Binnerna. *GNOME and CORBA*. The GNOME Project, 2000.

[Ste03] M. Stevens. *Java Web Services Architecture*, chapter Chapter 2 : Service-Oriented Architecture. Morgan Kaufmann Publishers, 2003.

[Str98] D. Stratton. The OMG CORBA Trader Service. Rapport technique, School of Information Technology and Mathematical Sciences, University of Ballarat, Australie, 1998.

[Sun97] Sun Microsystems. Java beans specification. Document en ligne, 1997. http ://java.sun.com/products/javabeans/docs/.

[Sun01] Sun Microsystems. Enterprise javabeans specification version 2.0. Document en ligne, Aout 2001. http ://java.sun.com/products/ejb/docs.html.

[Szy96] C. Szyperski. Independently extensible systems – software engineering potential and challenges. Dans *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australie, 1996.

[Szy98] C. Szyperski. *Component software : beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., 1998.

[Tha99] T.L. Thai. *Learning DCOM*. O'Reilly, 1st edition, Avril 1999.

[udd02] uddi.org. UDDI Version 3.0 Specification. Document en ligne, Juillet 2002. http ://uddi.org/.

[Vin97] S. Vinoski. Corba : Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, Fevrier 1997.

[WD01] R. Wuyts et S. Ducasse. Composition languages for black-box components. Dans *First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, 2001.

[Weg87] P. Wegner. Dimensions of object-based language design. Dans *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA)*, 1987.

[WK00] S. Wilson et J. Kesselman. *Java Platform Performance, Strategies and Tactics*. Addison Wesley, Mai 2000.

[Wor01] World Wide Web Consortium (W3C). Web services description language (wsdl) 1.1. W3C Note, Mars 2001. http ://www.w3.org/TR/wsdl.

[Yan03] J. Yang. Web service componentization. *Communications of the ACM*, 46(10) :35–40, Octobre 2003.

Annexe A

Glossaire

architecture : une description de la façon suivant laquelle un système est composé à partir de ses composants [SG96]

assemblage de composants : voir composition

conteneur : dans un modèle à composants, entité faisant partie de l'environnement d'exécution qui gère le cycle de vie des instances de composant et qui prend en charge les aspects non-fonctionnels.

composant : une 'brique' logicielle pré-fabriquée qui est conçue pour être composée, c'est à dire assemblée, avec d'autres composants. Un composant est séparé en classes, instances et paquetages.

composition (ou assemblage) de composants : En dehors du développement de composants, il s'agit de l'autre activité fondamentale de l'approche à composants. Le composition de composants fait référence à la configuration et connexion composition d'instances de composant à partir des éléments présents sur leur vue externe.

demandeur de service : Cet acteur est le client d'un service particulier. Pour pouvoir interagir avec le service, il doit d'abord découvrir dans le registre un *fournisseur du service* et ensuite se lier avec lui pour obtenir un *objet de service*.

descripteur de service : description caractérisant un service composée d'une *interface de service* et d'un ensemble de propriétés.

disponibilité dynamique : dans un modèle à composants, fait référence au fait qu'une classe ou une instance de composant puisse devenir indisponible ou disponible à tout moment pendant que l'application est en train d'être exécutée.

fournisseur de service : Le fournisseur de services a pour rôle de fournir des *objets de service* qui implémentent une ou plusieurs *interfaces de services*. Cet acteur est responsable de publier, auprès du *registre de services*, les descripteurs correspondant aux services qu'il fournit pour permettre leur découverte. Les

fournisseurs de services ont une *disponibilité dynamique*, ils peuvent à tout moment commencer à fournir ou arrêter de fournir leurs services.

interface de service : interface permettant l'interaction avec un service et faisant partie du *descripteur de service*. L'interface de service est considérée un contrat et peut être définie par une organisation.

modèle à composants : un modèle à composants définit un ensemble de caractéristiques des composants, de leurs assemblages et est accompagné par un support d'exécution.

objet de service : Implémentation d'une *interface de service*. Un objet de service est crée par un *fournisseur de services* au moment ou un *demandeur de services* se lie avec lui.

plate-forme de services : réalisation particulière de l'approche à services. Des exemples de plate-formes sont Jini, OSGi et le courtier CORBA.

registre de services : Le registre de services est l'intermédiaire entre les *fournisseurs* et les *demandeurs* de services. Le registre contient un ensemble de *descripteurs de services* publiés par les fournisseurs de services et il fournit des mécanismes permettant de l'interroger pour obtenir des références envers les fournisseurs de ces services. L'ensemble de descripteurs de services qui se trouvent dans le registre change constamment car à tout moment un fournisseur de services peut demander la publication où le retrait de ses services du registre.

service : Un comportement défini de façon contractuelle qui est réalisé par un *objet de service* qui implémente une *interface de service*. Un *fournisseur de services* est chargé de créer les objets de service lorsqu'un *demandeur de services* se lie avec lui après l'avoir découvert dans un *registre de services*.

vue externe de composant : La vue externe d'un composant représente les différents éléments d'une classe de composant (interfaces fonctionnelles, propriétés de configuration) qui sont visibles pour les clients des instances du composant.

vue interne de composant : La vue interne d'un composant représente les différents éléments d'une classe de composant qui sont visibles pour l'environnement d'exécution au niveau des instances du composant.

Annexe B

Exemple comparatif OSGi vs. ServiceBinder

Cette annexe montre un exemple de programmation d'un fournisseur d'un service ayant des dépendances, ce qui équivaut à un composant à services qui fournit un service et qui à des dépendances. Concrètement, ce composant implémente un service de correction orthographique qui requiert d'au moins un dictionnaire pour réaliser sa tâche. Le composant supporte pendant l'exécution l'ajout et retrait de dictionnaires, ce qui revient à modéliser sa dépendance comme ayant une cardinalité `1..n` et dynamique. L'implémentation du composant est présentée à la suite. Dans le premier cas, elle est réalisée de façon directe dans le framework OSGi et dans le deuxième elle est réalisée à l'aide du ServiceBinder. Le code correspondant à la gestion des dépendances dans le premier cas est montré en gras. Ces exemples font partie d'un tutorial qui a été réalisé dans le cadre d'un cours d'utilisation d'OSGi[1].

Programmation OSGi standard

```
package tutorial.example5;
import java.io.*;
import java.util.*;
import org.osgi.framework.*;
import tutorial.example2.service.DictionaryService;
import tutorial.example5.service.SpellCheckService;
/**
  * This class implements a bundle that implements a spell
  * check service. The spell check service uses all available
  * dictionary services to check for the existence of words in
  * a given sentence. This bundle not only monitors the dynamic
  * availability of dictionary services, but it manages the
  * aggregation of all available dictionary services as they
  * arrive and depart. The spell check service is only registered
  * if there are dictionary services available, thus the spell
```

[1]Le tutorial complet est accessible à l'addresse suivante : http ://oscar-osgi.sourceforge.net/tutorial/

```
  * check service will appear and disappear as dictionary
  * services appear and disappear, respectively.
**/
public class Activator implements BundleActivator, ServiceListener {
    // Bundle's context.     private BundleContext m_context = null;
    // List of available dictionary service references.
    private ArrayList m_refList = new ArrayList();
    // Maps service references to service objects.
    private HashMap m_refToObjMap = new HashMap();
     // The spell check service registration.
    private ServiceRegistration m_reg = null;
    /**
     * Implements BundleActivator.start(). Adds itself
     * as a service listener and queries for all currently
     * available dictionary services. Any available dictionary
     * services are added to the service reference list. If
     * dictionary services are found, then the spell check
     * service is registered.
     * @param context the framework context for the bundle.
    **/
    public void start(BundleContext context) throws Exception
    {
        m_context = context;
         // Listen for events pertaining to dictionary services.
        m_context.addServiceListener(this,
            "(&(objectClass=" + DictionaryService.class.getName() + ")" +
            "(Language=*))");
        // Query for all dictionary services.
        ServiceReference[] refs = m_context.getServiceReferences(
            DictionaryService.class.getName(), "(Language=*)");
        // Add any dictionaries to the service reference list.
        if (refs != null)
        {
            // Lock the list.
            synchronized (m_refList)
            {
                for (int i = 0; i < refs.length; i++)
                {
                    // Get the service object.
                    Object service = m_context.getService(refs[i]);
                    // Make that the service is not being duplicated.
                    if ((service != null) &&
                        (m_refToObjMap.get(refs[i]) == null))
                    {
                        // Add to the reference list.
                        m_refList.add(refs[i]);
                        // Map reference to service object for easy look up.
                        m_refToObjMap.put(refs[i], service);
                    }
                }
                // Register spell check service if there are any
                // dictionary services.
                if (m_refList.size() > 0)
                {
                    m_reg = m_context.registerService(
                        SpellCheckService.class.getName(),
                        new SpellCheckServiceImpl(), null);
                }
            }
```

```
    }
}
/**
 * Implements BundleActivator.stop(). Does nothing since
 * the framework will automatically unregister any registered services,
 * release any used services, and remove any event listeners.
 * @param context the framework context for the bundle.
**/     public void stop(BundleContext context)
{         // NOTE : The services automatically released.     }
 /**
 * Implements ServiceListener.serviceChanged(). Monitors
 * the arrival and departure of dictionary services, adding and
 * removing them from the service reference list, respectively.
 * In the case where no more dictionary services are available,
 * the spell check service is registered. As soon as any dictionary
 * spell check becomes available, the spell check service is
 * reregistered.      * @param event the fired service event.
**/     public void serviceChanged(ServiceEvent event)
{         String[] objectClass =
        (String[]) event.getServiceReference().getProperty("objectClass");
    // Add the new dictionary service to the service list.
    if (event.getType() == ServiceEvent.REGISTERED)
    {
        synchronized (m_refList)
        {
            // Get the service object.
            Object service = m_context.getService(event.getServiceReference());
            // Make that the service is not being duplicated.
            if ((service != null) &&
                (m_refToObjMap.get(event.getServiceReference()) == null))
            {
                // Add to the reference list.
                m_refList.add(event.getServiceReference());
                // Map reference to service object for easy look up.
                m_refToObjMap.put(event.getServiceReference(), service);
                // Register spell check service if necessary.
                if (m_reg == null)
                {
                    m_reg = m_context.registerService(
                        SpellCheckService.class.getName(),
                        new SpellCheckServiceImpl(), null);
                }               }
            else if (service != null)
            {
                m_context.ungetService(event.getServiceReference());
            }
        }
    }
    // Remove the departing service from the service list.
    else if (event.getType() == ServiceEvent.UNREGISTERING)
    {
        synchronized (m_refList)
        {
            // Make sure the service is in the list.
            if (m_refToObjMap.get(event.getServiceReference()) != null)
            {
                // Unget the service object.
                m_context.ungetService(event.getServiceReference());
                // Remove service reference.
```

```
                m_refList.remove(event.getServiceReference());
                // Remove service reference from map.
                m_refToObjMap.remove(event.getServiceReference());
                // If there are no more dictionary services,
                // then unregister spell check service.
                if (m_refList.size() == 0)
                {
                    m_reg.unregister();
                    m_reg = null;
                }
            }
        }
    }
}
/**
 * A private inner class that implements a spell check service;
 * see SpellCheckService for details of the service.
**/
private class SpellCheckServiceImpl implements SpellCheckService
{
    /**
     * Implements SpellCheckService.check(). Checks the
     * given passage for misspelled words.
     * @param passage the passage to spell check.
     * @return An array of misspelled words or null if no
     *         words are misspelled.
    **/
    public String[] check(String passage)
    {
        // No misspelled words for an empty string.
        if ((passage == null) || (passage.length() == 0))
        {
            return null;
        }
        ArrayList errorList = new ArrayList();
        // Tokenize the passage using spaces and punctionation.
        StringTokenizer st = new StringTokenizer(passage, " ,.!?; :");
        // Lock the service list.
        synchronized (m_refList)
        {
            // Loop through each word in the passage.
            while (st.hasMoreTokens())
            {
                String word = st.nextToken();
                boolean correct = false;
                // Check each available dictionary for the current word.
                for (int i = 0; (!correct) && (i < m_refList.size()); i++)
                {
                    DictionaryService dictionary =
                        (DictionaryService) m_refToObjMap.get(m_refList.get(i));
                    if (dictionary.checkWord(word))
                    {
                        correct = true;
                    }
                }
                // If the word is not correct, then add it
                // to the incorrect word list.
                if (!correct)
                {
```

```
                        errorList.add(word);
                    }
                }
            }
            // Return null if no words are incorrect.
            if (errorList.size() == 0)
            {
                return null;
            }
            // Return the array of incorrect words.
            return (String[]) errorList.toArray(new String[errorList.size()]);
        }
    }
}
```

Avec le ServiceBinder

Activator :

```
package tutorial.example7;
import org.ungoverned.gravity.servicebinder.GenericActivator;
/**
  * This example re-implements the spell check service of
  * Example 5 using the Gravity Service Binder.
**/
public class Activator extends GenericActivator
{ }
```

Descripteur de composant

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
   <component implementation="tutorial.example7.SpellCheckServiceImpl">
     <provides service="tutorial.example5.service.SpellCheckService"/>
     <requires
       service="tutorial.example2.service.DictionaryService"
       filter="(Language=*)"
       cardinality="1..n"
       policy="dynamic"
       bind-method="addDictionary"
       unbind-method="removeDictionary"
     />
   </component>
</bundle>
```

Code applicatif

```
package tutorial.example7;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;
import tutorial.example2.service.DictionaryService;
import tutorial.example5.service.SpellCheckService;
/**
  * This class re-implements the spell check service of Example 5.
```

```
 * This service implementation behaves exactly like the one in
 * Example 5
**/
public class SpellCheckServiceImpl implements SpellCheckService
{
    // List of service objects.
    private ArrayList m_svcObjList = new ArrayList();
     /**
      * Bind and unbind methods
      * @param dictionary the dictionary to add to the spell
      *                   check service.
     **/
     public void addDictionary(DictionaryService dictionary)
     {
         synchronized (m_svcObjList) // Lock list and add service object.
         {
             m_svcObjList.add(dictionary);
         }
     }
     public void removeDictionary(DictionaryService dictionary)
     {
         synchronized (m_svcObjList) // Lock list and remove service object.
         {
             m_svcObjList.remove(dictionary);
         }
     }
     /**
      * Checks a given passage for spelling errors. A passage is any
      * number of words separated by a space and any of the following
      * punctuation marks : comma (,), period (.), exclamation mark (!),
      * question mark (?), semi-colon (;), and colon( :).
      * @param passage the passage to spell check.
      * @return An array of misspelled words or null if no
      *         words are misspelled.
     **/
     public String[] check(String passage)
     {
         // No misspelled words for an empty string.
         if ((passage == null) || (passage.length() == 0))
         {
             return null;
         }
         ArrayList errorList = new ArrayList();
         // Tokenize the passage using spaces and punctionation.
         StringTokenizer st = new StringTokenizer(passage, " ,.!?; :");
         // Lock the service list.
         synchronized (m_svcObjList)
         {
             // Loop through each word in the passage.
             while (st.hasMoreTokens())
             {
                 String word = st.nextToken();
                 boolean correct = false;
                 // Check each available dictionary for the current word.
                 for (int i = 0; (!correct) && (i < m_svcObjList.size()); i++)
                 {
                     DictionaryService dictionary =
                         (DictionaryService) m_svcObjList.get(i);
                     if (dictionary.checkWord(word))
```

```
                    {
                        correct = true;
                    }
                }
                // If the word is not correct, then add it
                // to the incorrect word list.
                if (!correct)
                {
                    errorList.add(word);
                }
            }
        }
        // Return null if no words are incorrect.
        if (errorList.size() == 0)
        {
            return null;
        }
        // Return the array of incorrect words.
        return (String[]) errorList.toArray(new String[errorList.size()]);
    }

}
```