HAL
open science

iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques

Clement Escoffier

To cite this version:

Clement Escoffier. iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2008. Français. NNT : . tel-00347935

HAL Id: tel-00347935

https://theses.hal.science/tel-00347935v1

Submitted on 17 Dec 2008

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

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

UNIVERSITÉ JOSEPH FOURIER (GRENOBLE I)

THÈSE

Pour obtenir le grade de

DOCTEUR de l'Université JOSEPH FOURIER

Discipline : INFORMATIQUE

Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Présentée et soutenue publiquement par

CLEMENT ESCOFFIER

Le 3 décembre 2008

---

iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques

---

Directeur de thèse:
Philippe LALANDA

JURY

Présidente	Laurence Nigay, Professeur à l'Université Joseph Fourier, Grenoble
Rapporteurs	Alexander Wolf, Professeur à l'Imperial College, Londres Michel Riveill, Professeur à Polytech'Nice, Sophia Antipolis
Examinateur	François Exertier, Bull SAS
Encadrants	Philippe Lalanda, Professeur à L'Université Joseph Fourier, Grenoble Richard S. Hall, Sun Microsystems

Thèse préparée au sein du Laboratoire Informatique de Grenoble (LIG)

Résumé

La récente évolution de l'informatique a ouvert la voie à de nouveaux types d'applications. En effet, la convergence de l'Internet et de l'informatique ubiquitaire permet le développement d'applications intégrant le monde virtuel dans le monde physique. Cependant, cette convergence a vu émerger de nouveaux besoins tel que le dynamisme. Bien que de nombreux travaux aient étudié cette propriété, la création d'applications pouvant évoluer et réagir aux changements environnementaux et contextuels reste très complexe.

Cette thèse s'intéresse à la conception, au développement et à l'exécution d'applications dynamiques. L'approche proposée s'appuie sur les récents efforts réalisés autour des modèles à composant. En effet, iPOJO, le modèle réalisé, combine l'approche à service et la programmation par composant afin d'introduire des caractéristiques de dynamisme au sein d'un modèle à composant. iPOJO propose un langage de composition permettant la conception d'applications nativement dynamiques, tout en facilitant le développement des ces applications. Celles-ci sont décrites de manière à réduire le couplage avec des implémentations de composants spécifiques. Enfin, iPOJO fournit une machine d'exécution. Cette machine propose des mécanismes d'introspection, de reconfiguration et d'extensibilité permettant la supervision d'applications ainsi que l'adaptabilité de cette plate-forme iPOJO à différents domaines.

L'implémentation d'iPOJO est hébergée sur le projet Apache Felix. Elle est actuellement utilisée dans différents projets industriels tel que des plates-formes domestiques, le serveur d'applications JEE JOnAS ainsi que l'intergiciel pour téléphone mobile OW2 uGASP.

Mots-clés : applications dynamiques, approche à service, modèles à composant à service, architecture logicielle, OSGi™, iPOJO.

Summary

Recent evolution in software paves the way for new kinds of applications. The confluence between Internet-based and pervasive applications allows developers to create applications that blur the boundary between the virtual and physical worlds. However, the result of this confluence is the emergence of new requirements, such as dynamic evolution. Despite several works studying this property, creating applications that support dynamic evolution, such as environmental or contextual adaptations, remains a challenging task.

This thesis investigates the design, development, and execution of dynamic applications. As part of this investigation, the thesis proposes a novel approach for creating dynamic applications based on the recent work around component models. iPOJO, the proposed model, combines concepts from service-oriented computing and component-based software engineering. iPOJO provides a composition language for designing applications that natively support dynamism, while simplifying their development. The resulting application compositions are described abstract in order to reduce the coupling with specific component implementations. Finally, iPOJO provides an execution framework. This framework exhibits capabilities such as introspection, reconfiguration, and extensibility.

The iPOJO implementation is hosted at the Apache Felix project. It is used in different industrial projects such as home gateways, the JOnAS JEE server, and the OW2 uGASP mobile phone middleware.

Keywords: dynamic applications, service-oriented computing, service-oriented component models, software architecture, OSGi™, iPOJO.

Remerciements

Je tiens à remercier toutes les personnes m'ayant soutenu et encourager pendant ces trois ans de thèse et qui ont, à leur manière, permis à ce travail d'aboutir.

Tout d'abord, je tiens à remercier le jury. Je remercie Alexander Wolf et Michel Riveill pour avoir accepté de rapporter ma thèse. Je remercie également Laurence Nigay d'avoir accepté de présider le jury de cette thèse et François Exertier pour avoir accepté d'examiner mon travail.

Je tiens également à remercier tous ceux qui m'ont encadré durant ces travaux, Philippe Lalanda et Richard S. Hall. Philippe, pour ces conseils, sa disponibilité, ses encouragements et ses remarques qui ont permis d'améliorer cette thèse. Richard S. Hall pour ces idées, nos innombrables et longues discussions, les opportunités qu'il m'a offert dont la chance de pouvoir héberger iPOJO sur Apache Felix et pour m'avoir invité à Tufts.

Je remercie également l'équipe Adèle qui m'a accueilli pendant plus de 5 ans, et tous ces membres passés et actuels avec qui j'ai partagé cafés et discussions. Je remercie tous ceux avec qui j'ai travaillé et plus particulièrement Johann Bourcier qui, grâce à ses idées et nos nombreuses discussions, a joué un grand rôle dans iPOJO.

Je souhaite aussi exprimer ma reconnaissance envers tous les utilisateurs d'iPOJO qui grâce à leurs soutiens et leurs retours ont contribués à son amélioration.

J'adresse également une pensée aux « Péons », à Thomas et à tous mes amis pour leur soutien, les franches rigolades et les soirées interminables. Un grand merci à eux.

Enfin, j'aimerais remercier mes parents pour m'avoir toujours soutenu et bien évidemment Déborah, qui a réussi à me supporter, ce qui n'est pas une chose facile et qui n'a cessé de me soutenir pendant ces longues années.

Merci, Thanks, Danke

Table des matières

Chapitre 1 Introduction 11

1. Contexte 11

2. Problématique 12

3. Contribution 12

4. Cadre de travail 13

5. Plan du document 13

Chapitre 2 Emergence de Nouvelles Applications 17

1. Les déclencheurs 18

1.1. Vers un Web 3.0 18

1.2. Vers une intégration transparente des systèmes informatiques dans le monde réel 19

2. De nouveaux types d'applications 20

2.1. Les applications Machine to Machine (M2M) 20

2.2. Les applications résidentielles 22

2.3. Une mise en place complexe 23

3. Des propriétés indispensables 23

3.1. Le passage à l'échelle 24

3.2. La sécurité 24

3.3. L'autonomie 25

3.4. Gestion de l'hétérogénéité 26

3.5. L'évolutivité 27

4. Synthèse 28

Chapitre 3 Applications dynamiques 29

1. Des besoins importants d'adaptation 30

2. Applications dynamiques : définition & généralités 31

2.1. Applications adaptables et dynamisme 32

2.2. Reconfiguration dynamique 33

2.3. Gestionnaire d'adaptation *35*

2.4. Degré de variation *35*

3. Caractérisation des approches 36

3.1. Critères *36*

3.2. Qu'est-ce qui initialise l'adaptation ? *37*

3.3. Où est décrite la logique d'adaptation ? *39*

3.4. Gestion des politiques d'adaptation *41*

3.5. Positionnement du gestionnaire d'adaptation *42*

3.6. Mécanismes utilisés pour la reconfiguration dynamique *46*

3.7. Gestion des interruptions de service *48*

3.8. Synthèse *49*

4. Application de la caractérisation 50

4.1. Modèles à composant supportant la reconfiguration dynamique *50*

4.2. Architectures logicielles dynamiques *54*

5. Conclusion 58

Chapitre 4 Architectures à service étendues dynamiques 61

1. Approche à service et architectures à service 62

1.1. Principes de l'approche à service *62*

1.2. Approche à service et dynamisme *64*

2. SOA supportant l'approche à service dynamique 66

2.1. Caractérisation des SOA supportant l'approche à service dynamique *66*

2.2. CORBA *68*

2.3. Jini *70*

2.4. OSGi™ *72*

2.5. Les Services Web *74*

2.6. UPnP & DPWS *77*

2.7. Synthèse *79*

3. Modèles à composant à service 79

3.1. Principes des modèles à composant à service *80*

3.2. Modèles à composant à service & SOA étendus dynamiques *81*

3.3. Caractérisation des modèles à composant à service et études de l'existant *82*

4. Synthèse 90

Chapitre 5 iPOJO: Un modèle à composant à service 93

1. Approches pour la création d'applications dynamiques 94

1.1. Gestion du dynamisme dans le code *94*

1.2. Modèle à composant & Reconfiguration dynamique *94*

1.3. Approche à service dynamique *94*

1.4. Problématique & Approche *95*

2. iPOJO : un modèle à composant à service 96

2.1. Un modèle et une machine d'exécution *97*

2.2. Une architecture à service dynamique et isolation *97*

2.3. Un modèle de développement simple *98*

2.4. Un langage de composition structurelle *98*

2.5. Des fonctionnalités d'introspection et reconfiguration dynamique *99*

2.6. Des mécanismes d'extension *99*

3. Organisation de ce manuscrit 100

Chapitre 6 Principes & Caractéristiques 101

1. Types de composants et Instances 102

2. Spécification de service & Composition 103

2.1. Dépendance de service *104*

2.2. Modèle de l'état *105*

2.3. Propriétés de service *106*

3. Un modèle de dépendance riche et flexible 107

3.1. Simple ou agrégée *108*

3.2. Optionalité *108*

3.3. Filtrage *109*

3.4. Ordre, Tri et Sélection *109*

3.5. Politique de liaison *109*

4. Contexte de service & SOA hiérarchique 110

5. Cohérence entre dépendances de spécification et d'implémentation 111

6. Synthèse 112

Chapitre 7 Gestion du dynamisme dans les composants atomiques et composites 115

1. Objectifs d'un modèle de développement supportant le dynamisme 116

1.1. Un modèle simple et non intrusif mansquant le dynamisme 116

1.2. Gestion des primitives de l'approche à service dynamique 116

2. Modèle des types de composants atomiques 117

3. Types de composants atomiques pour Java 118

3.1. Type de composant atomique implémenté en Java 118

3.2. Une machine d'injection et d'interception 119

3.3. Démonstration du modèle de développement pour Java 121

4. Principes et concepts des types de composants composites pour supporter le dynamisme 124

4.1. Sous-services 125

4.2. Instances internes 126

4.3. Fourniture de service et export de service 126

4.4. Cohérence des compositions 128

4.5. Source de contexte et gestion du dynamisme contextuel 129

5. Gestion du dynamisme dans les instances de composants composites 130

6. Synthèse 132

Chapitre 8 Introspection, reconfiguration & extensibilité 135

1. Introspection d'applications à service dynamiques 136

2. Reconfiguration dynamique d'applications à service 138

2.1. Reconfiguration des dépendances de service 138

2.2. Reconfiguration des instances « concrètes » 140

3. Un modèle extensible 140

3.1. Composition des conteneurs et handlers 141

3.2. Extensibilité des composants atomiques 143

3.3. Extensibilité des composants composites 144

4. Synthèse 145

Chapitre 9 Implémentation 149

1. La plate-forme à service OSGi™ 150

2. Principes de fonctionnement 151

2.1. Analyse des bundles 152

2.2. Création et Gestion des fabriques 152

2.3. Contexte de service 152

2.4. Instances 153

3. Support des composants atomiques 153

3.1. Implémentation de la machine d'injection 154

3.2. Gestion du dynamisme 155

3.3. Autres handlers fournis 156

4. Support des composants composites 157

4.1. Gestion des sous-services 157

4.2. Gestion des services fournis 158

4.3. Reconfiguration de l'assemblage 159

5. Extensibilité et handlers externes 160

5.1. Handler pour composants atomiques et composites 160

5.2. Développement de handler 161

5.3. Déploiement et Exécution 161

6. Quelques chiffres et Synthèse 162

Chapitre 10 Validation 165

1. Evaluation de la machine d'injection et d'interception 166

1.1. Description du banc d'essai 166

1.2. Protocole de test 166

1.3. Ordre de grandeur 167

1.4. Plates-formes et technologies testées 168

1.5. Résultats et analyses 172

2. Temps de démarrage de la plate-forme 174

2.1. Description du banc d'essai *175*

2.2. Plates-formes et technologies testées *175*

2.3. Résultats et analyses *176*

3. Utilisation d'iPOJO sur une passerelle résidentielle 176

3.1. Contexte et problématique des passerelles résidentielles *176*

3.2. Vers un serveur d'applications pour les applications résidentielles *177*

4. Utilisation d'iPOJO dans le serveur d'application JEE JOnAS 178

4.1. Un serveur d'applications JEE au dessus d'OSGi™ *178*

4.2. Gestion du dynamisme *179*

4.3. Perspectives et attentes *179*

5. Utilisation d'iPOJO sur les téléphones mobiles 179

5.1. Contexte *179*

5.2. Pourquoi iPOJO *180*

5.3. Architecture choisie *181*

5.4. Exemple d'application *181*

5.5. Perspectives *181*

6. Conclusion 182

Chapitre 11 Conclusion & Perspectives 183

1. Synthèse 184

1.1. L'émergence d'un nouveau besoin *184*

1.2. Approche & Exigences *185*

1.3. iPOJO : un modèle à composant à service *186*

2. Perspectives 187

2.1. Support du déploiement *187*

2.2. Plate-forme d'exécution pour applications autonomiques et sensibles au contexte *188*

2.3. Mise en place d'infrastructure centrée domaine *189*

Chapitre 12 Bibliographie 191

Chapitre 1
Introduction

1. Contexte

Dans les 30 dernières années, le domaine informatique a beaucoup évolué. Les systèmes logiciels sont devenus indispensables dans de nombreux domaines. La taille et la complexité des logiciels sont en croissance continue. Cela est dû à des besoins plus nombreux et complexes. Cependant, l'arrivée de l'Internet et l'émergence d'objet communicant flouant la frontière entre le monde virtuel et physique rendent les applications encore plus complexes à réaliser. Les développeurs doivent ainsi faire face à de nouveaux défis tels que:

- Évolution - qui s'exprime comme une combinaison de plusieurs facteurs. En premier, il s'agit de l'évolution des plates-formes d'exploitation et d'exécution, mais aussi de celle concernant les technologies d'implémentation des logiciels. À cela s'ajoute, l'évolution de l'application elle-même. Les exigences des utilisateurs évoluent aussi. Ils désirent que les logiciels soient de plus en plus évolutifs et que ces évolutions soient réalisées le plus rapidement possible ;
- Complexité - la complexité des logiciels est liée aux besoins propres des domaines qu'ils adressent. Un autre facteur qui a une importance influence est la complexité de la plate-forme technologique. C'est le cas des applications reparties, où la mise en place de services non fonctionnels tels que la sécurité, les transactions et la reprise sur pannes s'ajoute à la complexité propre aux domaines applicatifs ;
- Hétérogénéité - un autre défi est de prendre en compte l'importance de la grande diversité des technologies et des services mis en œuvre au sein d'une application. Un développeur doit alors comprendre et savoir utiliser un grand nombre de technologies différentes loin de son domaine métier.
- Dynamisme – Ce nouveau défi a un gros impact sur le cycle de vie du logiciel. Celui-ci n'est plus conçu comme une brique monolithique fixe, mais comme un ensemble de briques flexibles et pouvant apparaître, disparaître ou évoluer au cours du temps. La complexité engendrée par le dynamisme est grande et s'applique à plusieurs niveaux comme la conception, le développement et l'exécution.

En même temps, les entreprises imposent certains critères stricts pour le développement des logiciels, comme la productivité et l'efficacité. Un logiciel doit être développé et adapté beaucoup plus rapidement qu'auparavant afin de prendre en compte les fluctuations de la demande.

L'approche à service est récemment apparue pour répondre à ces besoins. Elle propose de construire des applications à partir d'entités logicielles qui peuvent être fournies par des organisations tierces et qui peuvent évoluer dynamiquement. Ces entités sont nommées services. Un service fournit une fonctionnalité qui, pour être mise à disposition, est spécifiée, de façon syntaxique et/ou sémantique, par une description de services. La description de services permet aux éventuels clients de rechercher, découvrir et par la suite appeler un service. De manière idéale, ce protocole (découverte, sélection, invocation) peut être réalisé à l'exécution, cela donnant lieu à la création d'applications dynamiques utilisant des services disponibles à un moment précis, et donc donnant une grande flexibilité à l'application.

Popularisés principalement avec le développement des services Web, les principes de l'approche à service ne sont pas tout à fait nouveaux. Par exemple, le découplage entre le fournisseur et le consommateur (le client d'un service) de service trouve ses racines dans l'appel de méthodes à distance. L'intergiciel CORBA permettait avec le langage IDL de réaliser la séparation entre l'interface d'un objet et son implémentation, en décrivant dans un langage standard, indépendant de la technologie d'implémentation, la partie visible d'un objet, ses méthodes et propriétés.

Nos travaux partent de la conviction que le développement d'infrastructure de développement et d'exécution gérant le dynamisme est primordial pour remplir les besoins des applications innovantes. Dans ce contexte, nos travaux proposent un modèle à composant à service et une plate-forme d'exécution permettant de masquer le dynamisme lors de la conception et du développement de l'application. Le dynamisme sera ensuite géré par la machine d'exécution.

La section suivante présente notre problématique et les objectifs de cette thèse.

2. Problématique

Les différentes applications émergentes ont des besoins importants que les infrastructures de développement et d'exécution doivent supporter. L'un de ces besoins est le dynamisme. En effet, de plus en plus d'applications ont besoin de s'adapter en fonction des différentes évolutions, et du dynamisme issu de l'environnement et du contexte d'exécution.

Cette thèse étudie une nouvelle approche afin de concevoir, développer et exécuter des applications dynamiques. Cependant, une telle approche a de nombreux objectifs tels que :

- Fournir un modèle et une machine d'exécution proche permettant de manipuler les mêmes concepts lors de la conception et lors de l'exécution.
- Fournir un modèle de développement simple et masquant le dynamisme afin de faciliter le travail des développeurs.
- Fournir un modèle de composition permettant d'architecturer des applications qui supporteront le dynamisme.
- Permettre l'introspection afin de pouvoir connaitre l'état d'un système durant son exécution.
- Être extensible pour permettre l'adaptation du modèle et de la machine d'exécution a des environnements particuliers ayant d'autres besoins que le dynamisme.

De nombreux travaux proposent des solutions pour créer et exécuter des applications dynamiques. Cependant, ces travaux sont très souvent accompagnés de contraintes limitant leur utilisation. De plus, l'apparition de l'approche à service semble ouvrir de nouvelle voie pour la création d'applications dynamiques.

3. Contribution

Cette thèse propose une nouvelle approche concevoir, développer et exécuter des applications dynamiques. Cette thèse présente iPOJO, un modèle à composant à service flexible. iPOJO est basé sur l'introduction de concepts provenant de l'approche à service dans un modèle à composant. Celui-ci propose entre autres :

- D'architecturer des applications grâce à un langage de composition permettant le support du dynamisme lors de l'exécution.
- Un modèle de développement simple masquant intégralement le dynamisme pour le développeur.
- Une machine d'exécution gérant le dynamisme de manière transparente.

- Des mécanismes d'introspection et de reconfiguration dynamique permettant de remonter l'état d'un système et d'agir sur celui-ci.
- Des mécanismes d'extension permettant d'étendre le langage de composition, le modèle à composant ainsi que la machine d'exécution.

Le travail de cette thèse est intégralement implémenté et est disponible sous licence Apache. Le projet est hébergé comme un sous-projet du projet Apache Felix. De plus, iPOJO est actuellement utilisé dans différents domaines tels que les plates-formes résidentielles, les serveurs d'applications JEE et les plates-formes pour téléphones portables.

4. Cadre de travail

Cette thèse s'est déroulée au sein de l'équipe Adèle du laboratoire LIG. Cette équipe s'intéresse principalement à l'évolution dynamique des logiciels pour répondre à des besoins de mise à jour dynamique [1], de déploiement incrémental, mais aussi d'adaptation au contexte logiciel, matériel (mobilité entre autres) ou à l'évolution des besoins des utilisateurs.

Deux grands axes sont issus de cette problématique: à un grain assez fin, l'approche orientée service répond aux besoins de dynamisme de nombreuses applications actuelles, car les services peuvent apparaître et disparaître de façon imprévisible et se substituer les uns aux autres sans arrêter les applications. À plus gros grains, la réutilisation des nombreuses applications logicielles existantes, conçues indépendamment les unes des autres, qu'elles soient ad hoc, patrimoniales ou sur étagère est considérée comme critique.

5. Plan du document

Outre l'introduction, ce rapport de thèse est divisé en 3 grandes parties:

- La première partie propose une présentation du contexte ainsi qu'un état de l'art sur le dynamisme et sur l'approche à service. Trois chapitres composent cette partie:

  ♦ Le chapitre 2 introduit les nouvelles tendances des applications innovantes. Elles nous amènent de nouvelles problématiques que ce travail contribue à résoudre.

  ♦ Le chapitre 3 présente le dynamisme et les approches permettant de créer des applications dynamiques. Ce chapitre proposera entre autres une caractérisation des différentes approches pour créer et exécuter des applications dynamiques.

  ♦ Le chapitre 4 présente l'approche à service et les modèles à composant à service. Plus particulièrement, ce chapitre introduira l'architecture à service dynamique étendue permettant de concevoir, exécuter et administrer des applications dynamiques basées sur l'approche à service.

La seconde partie du document concerne la problématique abordée dans le document et la contribution apportée par cette thèse. Elle se compose de quatre chapitres:

♦ Le chapitre 5 expose la problématique abordée dans le document, c'est-à-dire la conception et l'exécution d'applications dynamiques, et introduit iPOJO, le modèle à composant à service proposé et les exigences devant remplir l'approche proposée.

♦ Le chapitre 6 présente les principes d'iPOJO. Seront abordés dans ce chapitre les éléments clés introduits dans iPOJO afin de supporter le dynamisme.

♦ Le chapitre 7 présente la gestion du dynamisme. À la fois la gestion du dynamisme dans les composants atomiques et dans les compositions est présentée dans le chapitre.

♦ Le chapitre 8 expose les mécanismes d'introspection, de reconfiguration dynamique et d'extensibilité proposée par iPOJO.

La troisième et dernière partie du document concerne l'implémentation, l'expérimentation et les résultats obtenus par la mise en œuvre de la proposition dans différents domaines d'applications. Elle se compose de trois chapitres:

♦ Le chapitre 9 décrit l'implémentation actuelle d'iPOJO.

♦ Le chapitre 10 présente deux bancs d'essai testant l'implémentation d'iPOJO face à ces principaux concurrents. De plus, l'utilisation d'iPOJO dans trois domaines d'application sera présentée.

♦ Le chapitre 11 conclut ce document et propose des pistes de recherche soulevées par ce travail.

Première partie

Etat de l’art

Chapitre 2
Emergence de Nouvelles Applications

Récemment, de nouveaux phénomènes ont bouleversé profondément le monde de l'informatique. Le développement d'Internet a transformé et transforme encore l'informatique moderne. Internet a révolutionné les technologies de l'information. Aujourd'hui, ce réseau planétaire est devenu indispensable pour les entreprises, profitant de ce nouveau média à la fois pour se faire connaitre, mais également pour fournir de nouveaux services. Des nombreuses applications reposent aujourd'hui sur Internet tels que les jeux massivement multi-joueurs. L'évolution des moyens de communication illustre parfaitement cette révolution et cette dépendance. Tout d'abord avec la popularisation du courrier électronique (apparu en 1961, bien avant Internet) puis l'explosion de la messagerie instantanée depuis 1996 et aujourd'hui avec le développement de la voix et la vidéo sur IP. Internet est partout et est en constante évolution. Le « World Wide Web » ou « web » a commencé par permettre de lier des documents statiques grâce à des liens hypertextes. Au début des années 2000, est apparu le web dynamique où les sites web ont pu diffuser du contenu dynamique. Récemment est apparu le Web 2.0, où dorénavant, les sites web peuvent proposer du contenu provenant d'autres sites. Alors que ce Web 2.0 vient juste d'apparaître, déjà le Web 3.0 (ou « Web Sémantique ») se profile à l'horizon.

Parallèlement à l'explosion d'Internet, l'informatique ambiante et l'informatique mobile connaissent également de grands succès. Les récents progrès en électronique ont permis cette émergence. Avec l'apparition des réseaux sans fil et la miniaturisation des composants électroniques, il a été possible de rendre mobile de nombreux appareils. Ces objets cohabitent aujourd'hui avec nous. Ainsi, nos téléphones portables aux multiples fonctions, les assistants personnels, les baladeurs numériques, les ordinateurs embarqués de voitures, les jouets de plus en plus perfectionnés peuvent être considérés comme des objets intelligents et communicants [2, 3]. Ils sont devenus indispensables pour de nombreuses personnes et rendent la frontière monde réel – monde virtuel de plus en plus diffuse. Ces appareils tendent à réaliser la vision de Marc Weiser, c'est-à-dire rependre des systèmes informatiques dans la vie courante par l'intermédiaire des objets *banals* [4].

Ces deux développements parallèles sont en train de se rejoindre. À travers ces objets communicants, Internet s'infuse dans nos vies quotidiennes. Il est désormais possible d'accéder à Internet en utilisant nos téléphones portables ou nos PDA. Il est également possible que des appareils diffusent des informations provenant d'Internet ou à l'inverse de diffuser des informations du monde physique sur Internet. Cette intégration ouvre la voie à de nouvelles applications profitant de cet accès au monde réel pour fournir de nouveaux services via Internet. Cependant, le développement de telles applications reste très complexe. En effet, elles ont des besoins particuliers qu'il est difficile de maitriser. Ce chapitre présente un bref historique d'Internet et de l'informatique ubiquitaire. Ensuite deux exemples d'applications seront étudiés afin de déterminer les propriétés cruciales requises par ces applications. Ces propriétés seront détaillées à la fin de ce chapitre.

1. Les déclencheurs

1.1. Vers un Web 3.0

Internet est un réseau informatique mondial qui prend ses origines à la création du réseau ARPANET en 1972 [5]. Le réseau ARPANET visait, entre autre, à uniformiser les différents protocoles de communication et a abouti au standard TCP/IP. Les réseaux construits autour d'ARPANET, financés et contrôlés par le gouvernement américain, étaient restreints à un but non commercial et réservés à des connexions vers des sites militaires ou universitaires. À partir des années 80, les connexions se sont étendues vers différentes institutions éducatives et à un nombre croissant de sociétés. Ce réseau était utilisé principalement pour le partage et l'échange de documents. En 1986, NSFNet, réseau spécialement conçu pour TCP/IP et successeur d'ARPANET, est mis en place. À l'occasion de la fusion de ces deux réseaux, le terme Internet est apparu, désignant un réseau s'appuyant sur TCP/IP.

À la fin des années 80, les premières entreprises fournissant un accès Internet furent fondées afin d'offrir une assistance aux réseaux de recherche régionaux, mais également de proposer des accès aux particuliers. À partir de 1994, les institutions gouvernementales et les fournisseurs créèrent leurs propres épines dorsales et liaisons. Les points d'accès régionaux devinrent les liens principaux entre les réseaux et la dernière restriction commerciale tomba. Cette ouverture accéléra le développement et le déploiement de services sur les réseaux, hébergés par des fournisseurs. Dans un premier temps, ces services consistèrent en de l'hébergement de contenu statique ou de données (bases de données).

Cette première version du web (nommé à postériori Web 1.0) se compose de documents statiques et de lien hypertexte entre ces documents. Ce n'est qu'à partir des années 2000 que se popularisent réellement les contenus dynamiques. Les sites web peuvent alors proposer du contenu pouvant évoluer avec le temps. Cependant, tout le contenu et les services du site étaient fournis par le site. Le web était alors une sorte de multitude de sites pratiquement indépendants les uns des autres.

À partir de 2004 est apparu le Web 2.0 [6]. Cette révolution n'est pas un changement technique, mais un bouleversement du mode d'utilisation de l'Internet. Les sites Web 2.0 permettent aux utilisateurs de non seulement récupérer de l'information, mais également d'y participer. Avec le développement des wikis, des blogs et des sites communautaires, l'utilisateur peut devenir un acteur complet et peut participer au contenu du site web. L'utilisateur peut également utiliser des applications web tel qu'il le ferait avec une application standard. La frontière Internet / Bureau devient de plus en plus floue. L'autre changement apporté par le Web 2.0 est de permettre à un site de publier du contenu ou d'utiliser des services provenant d'un autre site. Il n'est plus rare de naviguer sur un site utilisant un système de cartographie provenant d'un autre site et s'appuyant sur un système de paiement d'un troisième fournisseur.

Alors que le Web 2.0 se développe encore, le Web 3.0 est en train d'apparaître. Bien que rien n'est clair ce que sera réellement ce *web*, de nombreux acteurs s'accordent sur les faits qu'il deviendrait à la fois plus « intelligent » et ne se limiterait pas à Internet [7]. Le Web 3.0 serait plutôt une énorme base de données sur laquelle il serait possible de faire des requêtes en langage naturel. L'intégration d'Internet dans le monde physique par l'intermédiaire des objets communicants semble être également une des caractéristiques que le Web 3.0 devrait posséder. Il serait alors envisageable d'interagir avec des applications web via d'objet de la vie courante.

1.2. Vers une intégration transparente des systèmes informatiques dans le monde réel

En repoussant les limites technologiques toujours plus loin, le concept même de système d'information ou d'ordinateur change : d'une activité de traitement exclusivement centrée sur l'utilisateur, l'informatique tend à devenir une « intelligence ambiante » (Figure 1). Grâce aux technologies permettant de fabriquer des ordinateurs minuscules et omniprésents (processeur, mémoire vive et mémoire de stockage) et à la diminution de l'énergie consommée par ces composants, il est permis à presque tous les objets de la vie courante d'effectuer des actions sans interactions avec leur utilisateur. Tout ceci ne sera possible sans les récents progrès en réseau et en interfaces hommes-machines.

C'est un modèle fondamentalement différent des systèmes informatiques du 20e siècle et de la notion d'ordinateur qui lui était couramment rattachée. Ce nouveau concept introduit de profonds changements dans le monde de l'entreprise et celui de la vie quotidienne et qui impose inéluctablement une évolution capitale des activités et métiers informatiques.

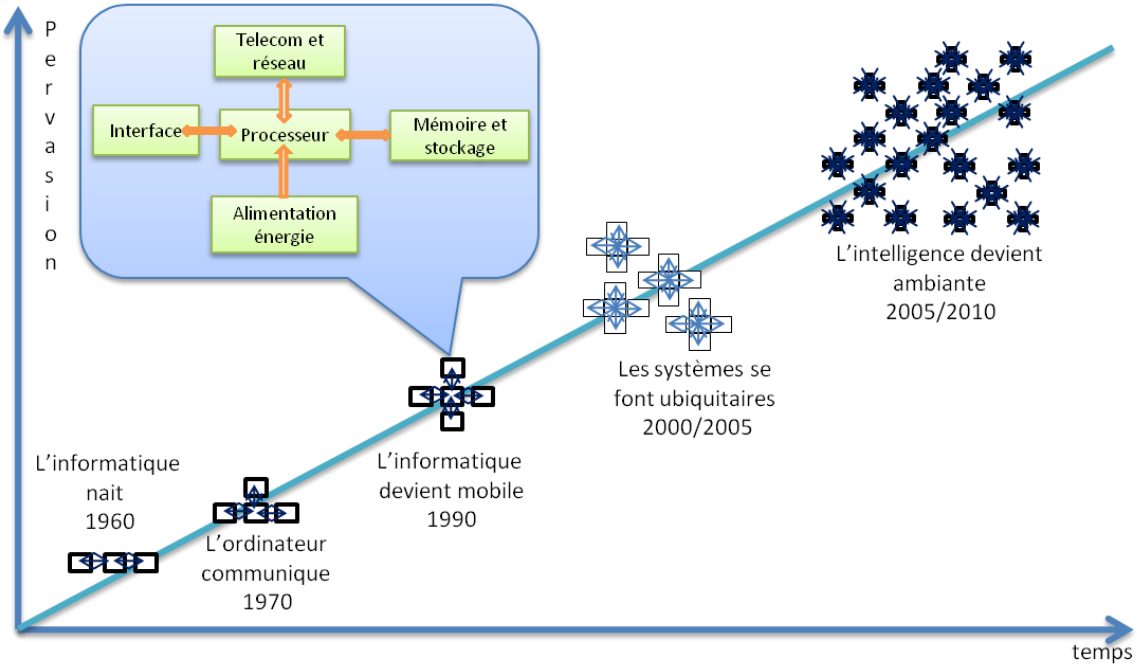


Figure 1. Évolution des ordinateurs vers la miniaturisation et à la diffusion dans le milieu ambiant [8]

Le terme « informatique ubiquitaire » a été inventé par Mark Weiser, lors de ses fonctions en tant que *technologue* en chef du centre de recherches de Xerox Palo Alto (PARC) [9]. Mark Weiser s'est inspiré du mouvement post-modernisme comme l'illustre de nombreuses références à *Ubik* de P. K. Dick [10], à Asimov (la série *Le Cycle des Robots* et plus particulièrement le *Défilé des Robots* [11]) ainsi qu'au film « *Mon Oncle* » de J. Tati [12]. Il s'est rendu compte que la mise en place d'une intelligence ambiante dépassait de beaucoup le seul domaine de l'informatique, mais nécessitait une compréhension des phénomènes sociaux, culturels et psychologiques.

Le Massachusetts Institute of Technology (MIT) a également contribué à la recherche de manière significative dans ce domaine, notamment avec *les choses qui pensent* (*Things That Think*)[13] et au projet « Oxygen »[14].

Aujourd'hui, les premiers objets communicants rentrent dans les maisons. Il s'agit principalement d'appareils de télécommunication, d'électroménager et de divers gadgets. Cependant, la popularisation du téléphone portable évolué (PDA communicant, Smartphone …) laisse à penser que les objets communicants vont se développer de plus en plus.

Figure 2. Exemple d'objets communicants
Une télévision « ambilight » de Philips, Un Nabaztag,
Les robots-jouets de Sony et un Smartphone

Si Internet a consacré l'avènement des réseaux planétaires conventionnels, la prochaine mutation se prépare sur un autre terrain : celui des réseaux d'objets sans fil et à très grande échelle [3]. En effet, le développement des réseaux sans fil (Wifi, Bluetooth) favorise le développement de l'intelligence ambiante et d'une informatique diffuse. De plus, un lien de plus en plus fort se créé entre ces réseaux et Internet. À la fois, Internet devient accessible à travers ces objets, mais ces objets communiquent également sur Internet.

2. De nouveaux types d'applications

La jonction de l'informatique ubiquitaire et d'Internet a offert un nouveau contexte très prometteur. Ce nouvel environnement a fait apparaître de nouveaux types d'applications. En profitant des récents progrès, des applications s'insérant de plus en plus dans la vie quotidienne voient le jour. Leur but est de développer de nouveaux services pour les utilisateurs, mais également d'avoir une emprise réelle sur le monde physique. Cette situation est en train de profondément changer le monde de l'informatique. Cette section décrit deux types d'applications innovantes s'illustrant dans ce contexte. Elle met en avant les problèmes rencontrés dans la mise en place de telles applications.

2.1. Les applications Machine to Machine (M2M)

Les applications M2M reposent sur la communication sans intervention humaine entre des machines. Dans une application M2M, des machines communiquent entre elles afin de s'échanger des données. Aujourd'hui le M2M se répand très rapidement dans de nombreux secteurs tels que la logistique.

Les machines interagissant dans une application M2M peuvent correspondre à des serveurs puissants, par exemple, pour faciliter les interactions entre entreprises. Ces machines peuvent également être des équipements communicants de faible puissance, éventuellement mobiles, permettant de faire le lien avec des processus physiques ou avec un utilisateur final. Le M2M impacte dès à présent les processus métiers des entreprises, soit en les optimisant, soit en permettant de nouveaux usages innovants et intégrés dans les cadres décisionnels des organisations. La construction d'applications de type M2M demande cependant la mise en place d'infrastructures de communication et de calculs spécifiques et complexes et remet en cause les solutions actuelles.

Il existe aujourd'hui de nombreuses applications de type M2M, largement utilisées ou en devenir. Pour illustration, nous détaillons ici un exemple typique d'utilisation de la RFID («Radio Frequency Identification »).

L'identification par radiofréquence est une méthode pour stocker et récupérer des données à distance en utilisant des marqueurs appelés « Tag RFID ». Les Tag RFID sont de petits objets tels que des étiquettes autoadhésives (Figure 3), qui peuvent être collés ou incorporés dans des produits. Ces Tags sont constitués d'une antenne qui leur permet de recevoir et de répondre à des requêtes radio émises depuis un lecteur. Un système RFID est généralement constitué de plusieurs composants qui communiquent de façon automatisée via des réseaux de type Internet (Figure 3). La technologie RFID peut être employée pour suivre du matériel, dans le domaine de l'automobile par exemple, pour du paiement (tramway de Porto par exemple) voire même dans le contexte d'implants humains. Le marché du RFID devrait passer de 2,7 milliards de dollars en 2006 à 26,2 milliards en 2016[1]. De plus le nombre de Tags vendus en 2016 devrait être 450 fois plus important que cette année.

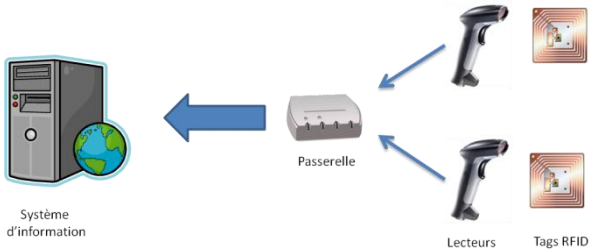


Figure 3. Architecture de collecte d'information RFID

Les composants majeurs sont

- les étiquettes électroniques,
- des passerelles permettant d'effectuer des calculs intermédiaires sur les données collectées,
- des serveurs contenant des logiciels applicatifs utilisant les informations collectées et transformées.

Les étiquettes électroniques sont composées d'une puce électronique et d'une antenne. Les étiquettes sont généralement passives et sont réveillées lorsqu'un lecteur émet un signal d'éveil. Une fois éveillée par le lecteur, un dialogue s'établit selon un protocole prédéfini et les données sont échangées. L'énergie nécessaire pour cet échange est fournie par le signal d'éveil. Les étiquettes actives sont munies d'une pile leur fournissant assez d'énergie pour alimenter une petite mémoire de stockage (environ 10 Kbits). Il est donc possible d'écrire sur une étiquette active alors que les étiquettes passives ne peuvent être que lues.

Les passerelles permettent de rassembler les données récoltées par des lecteurs. Ces passerelles doivent donc gérer à la fois les lecteurs évoluant sur un réseau local, mais aussi le stockage et le transfert des données vers le système d'information central. Ce transfert s'effectue généralement via Internet. Afin de soulager les systèmes d'information, ces passerelles permettent également d'amorcer le traitement de ces données (agrégation, filtrage …). La mise en place de ces passerelles reste complexe, car elles évoluent dans un contexte où des données sensibles peuvent être échangées. De plus, les algorithmes de traitements utilisés peuvent évoluer dans le temps en fonction des données lues.

Les serveurs du système d'information composent la dernière entité des applications RFID. Ces serveurs ont en charge la réception des données récoltées sur les passerelles, mais également leur traitement et généralement leur stockage. La principale difficulté dans la mise en place de ces serveurs est la quantité de données à traiter. Une fois stockées, ces données doivent également pouvoir être interrogées. Vu la quantité de données pouvant être récoltées, des algorithmes d'extraction de connaissances (data mining) sont souvent nécessaires.

[1] Source : Independent research and analysis on RFID (http://www.idtechex.com/)

Cette infrastructure est complétée par un intergiciel permettant de transmettre aussi bien les valeurs que les actions à effectuer. Cette infrastructure a en charge le déploiement et l'administration de toute la chaine de récolte de données. Par ailleurs, il existe un grand nombre de lecteurs d'étiquettes électroniques utilisant des protocoles différents qui, de plus, évoluent au cours du temps et dont la standardisation n'évolue pas très vite. Les passerelles doivent communiquer avec les différents lecteurs et être capables d'évoluer elles-mêmes. En raison des problèmes de sécurité, d'hétérogénéité des données échangées et des pannes potentielles, la mise en place de cette infrastructure s'avère extrêmement complexe. En plus des difficultés techniques, cette infrastructure doit également répondre à des contraintes de coûts importantes : les passerelles sont nombreuses et largement distribuées et doivent se contenter de capacités mémoire et de calculs limitées.

2.2. Les applications résidentielles

Les applications résidentielles visent à apporter de nouveaux services aux habitants d'une maison. Ces applications ont plusieurs objectifs tels que l'amélioration du confort de la maison ou la médicalisation de l'habitat en vue d'une hospitalisation à domicile.

L'architecture généralement utilisée pour développer des applications résidentielles consiste à déployer une passerelle résidentielle (Figure 4) dans la maison. La mise en place d'une telle plate-forme reste un défi très complexe aujourd'hui. Celle-ci doit pouvoir se connecter à Internet et centraliser l'exécution des applications résidentielles. Pour illustration, nous détaillons ici deux exemples typiques d'application résidentielle :

- Une application « réveil matin »
- Une application de maintien à domicile de personnes dépendantes ou âgées

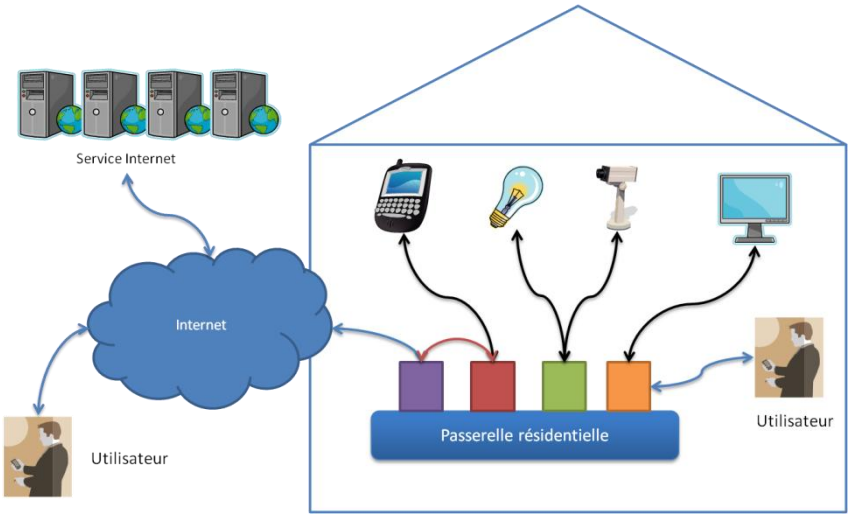


Figure 4. Passerelle résidentielle

L'application « réveil matin » automatise des actions à effectuer programmées à une heure précise. Par exemple, en semaine, 10 minutes avant l'heure du réveil, l'application peut allumer le chauffage de la salle de bain, mettre en route la cafetière … À l'heure du réveil, l'application ouvre les volets de la chambre, allume la télévision, allume l'agenda électronique et affiche l'agenda de la journée. Ce type d'application coordonne différents équipements de la maison afin d'offrir un nouveau confort à l'habitant.

Le maintien à domicile des personnes âgées et dépendantes est devenu un sujet très important aujourd'hui. Avec le vieillissement de la population, il devient crucial de pouvoir laisser des personnes âgées

chez elles plus longtemps (avant le potentiel placement en maison spécialisée). Le patient est alors équipé de capteurs (électrocardiogramme, détecteur de chute, tensiomètre …). Ces capteurs envoient leurs valeurs à la passerelle. L'application déployée sur la passerelle réceptionne et traite les données issues de capteurs disséminés dans la maison et sur le patient. Régulièrement (par exemple, chaque nuit), les données traitées sont envoyées à l'hôpital ayant en charge le patient. Si l'application détecte une situation urgente (chute, malaise), elle envoie immédiatement un message afin d'alerter les secours. Ce type d'application coordonne également des équipements de la maison, mais communique aussi avec le monde extérieur. De plus, cette application est critique vu qu'elle a en charge la vie du patient ce qui impose qu'elle ne doit pas être arrêtée et qu'elle ne doit pas tomber en panne.

Ces applications sont de plus en plus demandées à ce jour, mais ne sont que très rarement mises en place. En effet de nombreux problèmes subsistent avant le développement et la mise en place de telles applications. Tout d'abord, le modèle économique n'est pas clair. Par exemple, les fournisseurs d'accès internet par ADSL proposent un modèle « *Triple-Play* » (Internet, téléphone, vidéo) contrôlé exclusivement par le fournisseur d'accès. Les constructeurs et opérateurs proposent un autre modèle ouvrant le marché à de multiples fournisseurs de services spécialisés.

De nombreux défis techniques sont également à relever. La mise en place de services Internet, dans un contexte industriel ou résidentiel, demande la construction et la gestion de réseaux à large échelle hétérogènes. La plate-forme doit gérer les interactions avec des objets communicants qui utilisent des protocoles de communication différents et qui peuvent être dynamiques. De plus, la plupart des services proposés auront des exigences en termes de sécurité, de sureté et d'évolution.

2.3. Une mise en place complexe

Les deux précédentes sections ont décrit deux nouveaux types d'applications intégrant des objets communicants et Internet. Cependant, bien que les enjeux soient très importants, ce type d'application peine à se rependre. En effet, il est très complexe de mettre en place l'infrastructure nécessaire pour faire fonctionner correctement ces applications. De nombreux travaux restent à entreprendre afin de faciliter le développement, l'installation, l'exécution et l'administration de ces applications. Ces applications ont des contraintes importantes telles que la sécurité et l'autonomie. Le reste de ce chapitre détaillera quelles sont ces propriétés requises par ces applications et quelles sont les tendances actuelles.

3. Des propriétés indispensables

Les nouvelles familles d'applications citées précédemment sont aujourd'hui très difficiles à développer. Les besoins engendrés par Internet et la prolifération d'objets communicants entrainent de nouveaux défis. Des contraintes telles que la sécurité ou le support de l'évolutivité deviennent primordiales. Au cours des descriptions des applications précédentes cinq propriétés importantes sont apparues :

- Le passage à l'échelle, afin d'être capable de s'adapter en fonction de l'augmentation de la charge de travail.
- La sécurité, de manière à garantir l'intégrité de données sensibles et privées.
- L'hétérogénéité, afin de pouvoir intégrer des données de différents systèmes.
- L'autonomie, pour diminuer les coûts d'administration et également augmenter la sureté des applications.
- L'évolutivité, de manière à supporter les changements de l'environnement, mais également la mise à jour des applications

Toutes ces propriétés sont devenues incontournables pour les applications afin de pouvoir fonctionner correctement. Cette section décrit brièvement chacune de ces propriétés. Nous nous attarderons plus amplement sur le dynamisme, thème principal de cette thèse.

3.1. Le passage à l'échelle

En informatique et en télécommunication, le passage à l'échelle est une propriété très importante pour un système. Il se réfère à sa capacité à faire face à une augmentation du « travail » à effectuer [15]. Il n'existe toutefois pas de consensus concernant la mesure de cette capacité. Par exemple, le passage à l'échelle peut se référer à l'aptitude d'un système à toujours répondre de manière satisfaisante malgré l'augmentation du nombre de requêtes (utilisateurs), mais également à la capacité pour un réseau à supporter l'ajout de nouveaux nœuds.

Le passage à l'échelle n'est pas une nouvelle propriété [16]. Cependant, les nouvelles applications ont de plus en plus besoin d'avoir cette propriété. Par exemple, dans les systèmes M2M, les applications doivent pouvoir faire face à un grand nombre de capteurs et donc de données à traiter, les serveurs d'applications doivent garantir un « bon » temps de réponse en permanence, malgré les pics d'affluence … Afin de satisfaire ces requis, l'approche habituelle consiste à étendre le système en y rajoutant des ressources de calculs (généralement des machines physiques). Dans le cadre d'une grappe de machines, afin d'améliorer le temps de réponse d'un système d'information, la technique la plus usitée consiste à rajouter des serveurs. Quant à faire face au flux de données des capteurs dans un réseau M2M, des plates-formes intermédiaires sont souvent utilisées pour effectuer une partie du traitement (agrégation, médiation…).

Dernièrement, le positionnement géographique des nouvelles machines a évolué. Il était habituel de rajouter des machines dans des grappes ou des grilles de calculs. Aujourd'hui, la tendance est de placer ces ressources de calculs plus proches de l'utilisateur. Cette tendance est parfaitement illustrée par l' « Edge Computing », où des machines sont louées à la demande chez les fournisseurs d'accès Internet. Nous pouvons également remarquer que de plus en plus une partie du calcul est déplacée vers le client, voir chez le client. De nombreux sites web utilisent des scripts JavaScript (s'exécutant sur la machine du client) afin d'amorcer le traitement de données (vérification, agrégation, formatage …).

Bien que ces solutions se généralisent, elles posent de gros problèmes d'implantation et d'administration. Par exemple le rajout de machine dans un réseau entraine des problématiques de déploiement et de reconfiguration du réseau.

3.2. La sécurité

La sécurité des systèmes informatiques a toujours été un sujet très étudié comme l'illustre [17]. Ce besoin est d'autant plus présent dans les nouvelles applications que des données sensibles sont manipulées et peuvent transiter entre des machines. Par exemple, dans une application d'hospitalisation à domicile des données sur l'utilisateur sont gérées par l'application.

Un système est sécurisé lorsqu'il est capable de résister à des utilisations non autorisées tout en fonctionnant nominalement. La sécurité peut s'évaluer suivant six critères différents :

- La confidentialité
- L'intégrité
- La disponibilité
- L'authentification
- La non-répudiation
- La gestion des preuves

Assurer la confidentialité est le fait de garantir que des informations confidentielles ou privées ne seront par révéler à des tiers qui ne doivent pas en avoir connaissance. Il est évident que ce critère est extrêmement important dans les applications de maintien à domicile où des données sur la santé du patient peuvent être échangées avec un organisme de santé.

L'intégrité a pour but de garantir qu'un système est protégé contre les dysfonctionnements, les agressions et les attaques. Plus particulièrement, l'intégrité des données doit assurer que les données ne sont pas corrompues de manière volontaire ou accidentelle.

La disponibilité vise à garantir qu'un système doit rester disponible malgré les potentielles attaques ou utilisations non autorisées. Ce critère est devenu primordial. En effet de nombreuses applications sont distribuées et la non-disponibilité d'un seul des nœuds peut mettre en péril toute l'application.

Les trois autres critères ont été rajoutés dans la spécification ISO 13335 qui vise à définir la sécurité des systèmes informatiques et à donner des lignes de conduites pour obtenir un système sécurisé. L'authentification tend à assurer à un utilisateur utilisant des données de l'identité de l'émetteur des données qu'il consulte. Par exemple, une application traitant des données issues d'un réseau de capteurs RFID doit pouvoir vérifier que les données sont bien issues de ces capteurs et n'ont pas été rajoutées. La non-répudiation vise à empêcher que l'auteur d'une donnée puisse prétendre ensuite qu'il n'en est pas l'auteur ; elle implique l'intégrité, mais s'étend au-delà, car doit apporter la preuve que des informations ont bien été émises par l'auteur. Enfin, la gestion des preuves doit fournir un système archivant toutes les transactions du système afin de pouvoir vérifier l'authenticité des données échangées.

La sécurité est donc primordiale dans les nouvelles applications. En effet, la proximité des utilisateurs réels et les données échangées font que ces applications doivent être sécurisées. Cependant, ce besoin influence beaucoup les applications à la fois sur leurs architectures, mais également sur leur développement et le choix des composants et technologies mises en œuvre (choix des protocoles d'échanges, système de stockage de données …).

3.3. L'autonomie

L'autonomie a toujours était une propriété recherchée. En effet, l'ordinateur autonome se réparant, se configurant et s'adaptant tout seul a toujours été un rêve. De nombreuses entreprises ont lancé des projets pour créer des logiciels de plus en plus autonomes tels que Hitachi Harmonious Computing, Intel N1, Microsoft Dynamic Systems, ou encore le projet Organic Systems lancé par Fujitsu-Siemens. Aujourd'hui, l'autonomie est devenue une propriété cruciale dans les nouvelles applications. En effet, avec l'intégration de dispositifs communicants (+38% par an[2]) et l'augmentation de la complexité des applications, l'administration est devenue un problème très important et coûteux. Plus particulièrement, le fait qu'un système puisse s'administrer tout seul (zéro-configuration) est devenu une propriété de plus en plus désirée. Récemment, Kephart et Chess ont présenté dans « The Vision of Autonomic Computing » [18] que l'intégration d'appareils communicants dans des applications est en train de devenir un cauchemar. Ils annoncent que les architectures actuelles ne sont pas capables d'anticiper, de modéliser et de maintenir la complexité de ces interactions. Ils proposent l'informatique *autonomique* comme un moyen d'ajouter un certain degré d'autonomie au sein des applications. Ceci a pour objectif de libérer les administrateurs d'un certain nombre de tâches, le système s'optimisant automatiquement.

Bien que le terme informatique autonomique soit apparu en 2001 dans le document « Autonomic Computing: IBM's Perspective on the State of Information Technology »[19], l'informatique autonomique a un énorme succès dans de nombreux domaines tels que la finance, le transport, la logistique, les systèmes de distributions d'eau et d'énergie. Plus particulièrement, l'informatique autonomique est utilisée pour faciliter

[2] Source : http://www.research.ibm.com/autonomic/

l'administration de systèmes complexes tels que des applications distribuées ou l'administration de grappe de calcul [20-22] …

Avec la popularisation de l'informatique mobile, la complexité des applications a encore augmenté. En effet, le système doit réagir en fonction de la disponibilité du réseau, de l'arrivée et du départ d'autres appareils communicants proches. Les principes de l'informatique autonomique sont donc également très utilisés dans ce domaine [23].

Toutefois, bien que l'informatique autonomique ouvre des voies très intéressantes, nous sommes encore loin des systèmes entièrement autonomes [24]. De plus, la création d'applications autonomiques reste très complexe. En effet, en plus de l'application, il est nécessaire de mettre en place un mécanisme de supervision (permettant d'analyser l'état du système), une politique d'administration qui à partir de l'analyse du système est capable de réagir sur le système. Aujourd'hui, des outils visent à faciliter ce développement en fournissant des boites à outils et des plateformes de supervision tels que « l'Autonomic Toolkit » [25] (Figure 5), ou Autonomia [26].

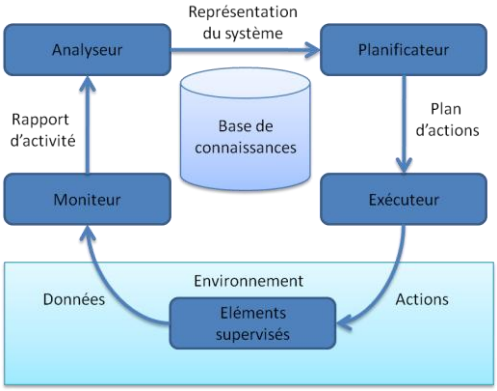


Figure 5. Architecture d'un gestionnaire "autonomique" proposée par IBM

3.4. Gestion de l'hétérogénéité

Avec l'essor d'Internet et des appareils communicants, l'intégration verticale (entre des dispositifs de terrain et un système d'information), mais également l'intégration horizontale (entre plusieurs entreprises) sont désormais possible. D'ailleurs, de nombreuses entreprises, dont des industries et des services doivent aujourd'hui être capables d'intégrer leurs systèmes d'informations avec leurs procédés opérationnels. En particulier, leur réussite est souvent liée à leur capacité à déployer des applications qui communiquent avec des dispositifs réels (gestion des protocoles de communications, gestion du dynamisme …), mais également avec d'autres applications, d'autres entreprises de manière transparente (transformation du format des données…). Les raisons de ces intégrations sont multiples. Tout d'abord, celles-ci permettent une prise de décision plus rapide en offrant en permanence des données à jour, issues à la fois du terrain, mais également d'autres entreprises partenaires. Deuxièmement, le fait de pouvoir agir directement sur des dispositifs réels permet de répondre rapidement aux demandes du marché. Cette flexibilité est devenue indispensable au vu des rapides fluctuations des demandes. Ceci permet également une personnalisation des services offerts en fonction des clients. La troisième raison de cette intégration concerne les coûts. Les coûts de maintenance, mais également les couts de développement en souscrivant à des services fournis par d'autres entreprises, sont énormément réduits.

Les intergiciels d'intégration d'application d'entreprise (Enterprise Application Integration ou EAI) permettent l'intégration de plusieurs applications hétérogènes [27]. L'intergiciel gère les échanges de messages entre ces applications. Le principal problème des EAI est le cout d'installation qui est très élevé. De plus, une fois le système mis en place, il peut être difficile à reconfigurer.

Avec l'avènement des services web, les EAI tendent à être remplacés par des bus à service d'entreprise (Enterprise Service Bus ou ESB) généralement basé sur des services Web [28]. Les Entreprises Service Bus (ESB) permettent à des applications hétérogènes de communiquer via l'envoi et la réception asynchrone de messages. Ces bus sont composés principalement d'un courtier de message réceptionnant et transmettant les messages des différents systèmes connectés au bus. L'intérêt de ces bus par rapport aux EAI est qu'ils offrent des services techniques plus élaborés et plus flexibles. Ils permettent également la mise en place de *médiateurs* spécifiques. Ceux-ci ont pour but de traduire les messages du bus afin de les rendre compréhensibles par une application spécifique. Les Enterprise Service Bus actuels permettent l'intégration horizontale, mais rares sont ceux adressant l'intégration verticale. Aujourd'hui, des chaines de médiation [29] tendent à être intégrées dans les ESB afin de faciliter la mise en place de ces bus, [30, 31] mais également afin de permettre l'intégration verticale grandement facilitée par les chaines de médiations [32].

Malheureusement, l'intégration reste une tâche très délicate. Il est difficile de faire collaborer des systèmes distribués complexes et hétérogènes. Des propriétés telles que la sécurité ou l'évolution doivent être prises en compte dans l'intégration.

3.5. L'évolutivité

Le contexte dans lequel s'exécutent ces nouvelles familles d'application pousse ces applications à devoir évoluer dynamiquement [33]. En effet, les applications doivent aujourd'hui s'adapter durant leur exécution en fonction de leur contexte d'exécution ainsi que pouvoir intégrer de nouvelles versions des composants qu'elles utilisent.

Dans les applications ubiquitaires, supporter le dynamisme est primordial. Tout d'abord, la sensibilité au contexte est très importante pour ces applications où l'environnement change constamment. Ces changements sont de plusieurs sortes [34] telles que la mobilité de l'utilisateur, l'apparition et la disparation d'appareils communicants et de la connectivité au réseau. Les applications sensibles au contexte (« *context-aware applications* ») sont étudiées depuis une quinzaine d'années [35, 36] et restent encore très étudiées aujourd'hui [37]. Malgré ces études, développer des applications sensibles au contexte reste une tâche très complexe. L'adaptation est généralement programmée à l'intérieur des applications. Ce type de mécanismes est souvent limité, car il n'est pas possible de réagir à des évènements non prévus à la conception de l'application. Les applications ubiquitaires doivent également supporter leur propre évolution. Il est de plus en plus fréquent de voir des composants des systèmes d'exploitation de téléphones mobiles subirent des mises à jour pour corriger un bug ou fournir un nouveau service. Les nouveaux systèmes d'exploitation pour téléphones mobiles fournissent aujourd'hui ce genre de fonctionnalités [38, 39]. Malheureusement, ces systèmes ne supportent pas la gestion des interactions entre les différents composants de l'application et plus particulièrement la gestion de ces interactions lorsqu'un composant est mis à jour.

L'évolutivité touche également les serveurs d'applications. Avec l'apparition de serveur dynamique [40, 41], les services techniques de ces serveurs peuvent évoluer dynamiquement. Le serveur doit supporter ces changements sans pour autant redémarrer ce qui aurait pour conséquence de rendre inaccessibles les applications hébergées. Toutefois, bien que beaucoup de serveurs d'applications ont choisi une infrastructure permettant ce dynamisme (OSGi™[42] ou JMX [43]), peu d'entre eux supportent réellement ce dynamisme sans un redémarrage complet du serveur. En effet, ce dynamisme doit généralement être géré manuellement par les développeurs ce qui rend la réutilisation de la base de code existante très limitée.

Nous pouvons illustrer ce phénomène sur un logiciel célèbre : Eclipse. Eclipse est un environnement de développement intégré (IDE) fondé une plate-forme à plug-in. Chaque possibilité d'Eclipse est livrée dans un plug-in. Bien qu'Eclipse ne soit pas un serveur d'applications à proprement dit, on y retrouve des concepts communs. Par exemple, Eclipse fournit des services techniques pour faciliter le développement de plug-ins (persistance, accès aux fichiers …). Eclipse permet également, l'installation dynamique de nouveaux plug-ins.

Eclipse tourne au dessus d'une passerelle OSGi™ qui gère le déploiement dynamique des plug-ins. Cependant, lorsqu'un plug-in est installé Eclipse conseille un redémarrage (Figure 6). La transformation de la base de code existante (2,5 millions de lignes de code) serait trop coûteuse pour supporter le dynamisme. C'est pourquoi, bien qu'Eclipse possède toute l'infrastructure permettant la mise à jour dynamique de l'environnement de développement, celle-ci n'est gérée que partiellement. Dans le contexte d'Eclipse, le redémarrage est possible. Il est beaucoup plus difficile à négocier lorsque nous devons redémarrer des serveurs d'applications ou des passerelles résidentielles.

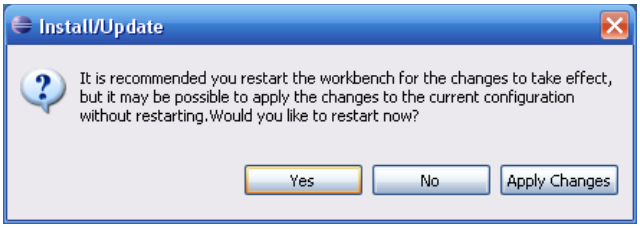


Figure 6. Eclipse conseille de redémarrer lorsqu'un nouveau plug-in est installé

L'évolutivité est donc devenue un challenge majeur aujourd'hui pour les fabricants de systèmes d'exécution. Ils doivent dorénavant offrir la possibilité pour une application de pouvoir s'adapter à son environnement et également de pouvoir supporter l'évolution de cette application. Cette capacité est devenue cruciale dans de nombreux domaines d'applications. Bien que cette propriété soit étudiée depuis longtemps [44], il reste aujourd'hui très délicat de développer des applications dynamiques et encore plus délicat de transformer des applications existantes en applications dynamiques. Peu d'intergiciels offrent cette possibilité. En outre, bien que supporté par l'intergiciel, il est extrêmement complexe de gérer correctement ce dynamisme. En particulier, des problèmes de concurrence, de gestion de l'état et de disponibilité de service interviennent dans le processus de gestion du dynamisme.

4. Synthèse

La prolifération d'objets communicants accompagnés par l'essor d'Internet pousse de nombreuses industries à proposer des applications intégrant monde réel et monde informatique [9]. En effet, la jonction entre le monde physique et les processus opérationnels permet de développer de nouveaux services, mais également d'améliorer des systèmes existants. Cependant, ce nouveau contexte change profondément le développement de nouvelles applications. En effet, afin de supporter ce nouvel environnement, ces applications ont des besoins cruciaux. Bien que ces besoins existaient tous auparavant, cet environnement les rend incontournable et les complexifie encore plus. Ce chapitre a présenté deux exemples d'applications rencontrant cette problématique : les applications M2M et les passerelles résidentielles. Ce chapitre a également présenté les besoins rencontrés dans le développement de ces applications tels que le passage à l'échelle, la sécurité, l'autonomie, la gestion de l'hétérogénéité et le dynamisme. Ces changements ouvrent de nouveaux pans de recherche dans de nombreuses disciplines du génie logiciel et du middleware telles que la gestion de configuration [45], les environnements de développement [46], les plates-formes d'exécution (serveur d'applications [47], système d'exploitation) … De plus, les relations entre ces propriétés et la séparation des préoccupations restent difficiles à mettre clairement en place[48].

Cette thèse s'intéresse principalement au dynamisme. Ce manuscrit propose de fusionner des concepts d'architectures logicielles, des modèles à composants et de l'approche à service afin de concevoir à la fois une machine d'exécution masquant le dynamisme pour le développeur et un moyen de décrire des applications qui supporteront le dynamisme.

Chapitre 3
Applications dynamiques

Le chapitre précédent a montré que le rapprochement d'Internet et de l'informatique ubiquitaire ouvrait la voie à de nouvelles applications très prometteuses. Cependant, ce nouveau contexte demande des applications capables de s'adapter très rapidement. La prise en compte de ces évolutions accroit le coût de développement et de maintenance des logiciels qui deviennent également plus gros et plus complexes, qui intègrent des briques hétérogènes et qui s'exécutent sur de multiples ordinateurs de manière ininterrompue.

La capacité à faire face à ces besoins dépend de nombreux facteurs. Un de ces facteurs est la capacité à pouvoir faire évoluer des systèmes existants afin d'adapter ces systèmes à de nouveaux requis. Le coût de ces évolutions est en constante augmentation. En 1991, ce coût était estimé à 60% du coût global du logiciel [49], alors qu'il est estimé à plus de 90% en 2000 [50]. Ce cout résulte fréquemment d'une mauvaise compréhension de l'architecture globale du système et des dépendances entre ses composants. Ce problème est encore plus flagrant sur de grands systèmes, complexes et fonctionnant de manière ininterrompue.

De plus, la traditionnelle évolution du logiciel n'est pas le seul changement pouvant apparaître dans un logiciel. Les applications innovantes détaillées dans le chapitre 2, font également face à un problème de disponibilité des services qu'elles utilisent. Ceci se remarque particulièrement dans le contexte des applications ubiquitaires qui dépendent d'objets physiques. Ces dispositifs peuvent être mobiles, s'éteindre, réapparaître … La prise en compte de ces événements complexifie encore plus le développement de ces applications.

De nombreux domaines de recherche étudient ces problématiques telles que les systèmes auto-adaptables [51-53], l'informatique sensible au contexte[36, 54], le déploiement de logiciels [55, 56] et les lignes de produits [57-59]. Ce chapitre a pour but de définir le terme « application dynamique » et les concepts relatifs au dynamisme et à sa gestion. Ce chapitre explore également les différents moyens de traiter le dynamisme afin de positionner différents travaux permettant de développer ou/et d'exécuter des applications dynamiques.

1. Des besoins importants d'adaptation

Les nouveaux types d'applications présentés dans le chapitre précédent exigent une propriété remarquable : il leur est nécessaire de pouvoir évoluer à l'exécution. L'évolution est nécessaire pour ajouter de nouvelles fonctionnalités à une application, mais également pour en améliorer la qualité ou pour l'adapter à un nouvel environnement d'exécution ou à un nouveau contexte[3].

De nombreux travaux se sont intéressés ou s'intéressent toujours à l'adaptation dynamique d'applications. Ces travaux sont très hétérogènes et adressent la modélisation de l'adaptation, l'exécution, ou la conception et le développement d'applications… Nous retrouverons donc des travaux provenant de différents domaines de l'informatique telle que l'architecture logicielle, les intergiciels, l'informatique autonomique… Cependant, les solutions proposées sont hétérogènes et supportent très rarement les différents besoins exigés par les adaptations dynamiques.

Les deux premiers besoins sont souvent traités dans la littérature [60, 61]. L'évolution peut entrainer, par exemple, le remplacement d'un composant[4] de l'application par un ou plusieurs autres (Figure 7). Ce remplacement peut être iso-fonctionnel ou non. Cette évolution sert généralement à ajouter des fonctionnalités à une application ou à améliorer un composant existant. Ce type d'adaptation est le plus couramment supporté, mais il suppose un contrôle complet de l'application. Ce dynamisme est généralement provoqué par un administrateur qui met en place la nouvelle configuration de l'application.

Figure 7. Exemple d'évolution d'une application - Le composant B est remplacé par B' et C

Un autre besoin d'adaptation provient de l'évolution de l'environnement d'exécution de l'application. Il concerne l'évolution de service tiers ou de ressources n'appartenant pas directement à l'application, mais que l'application utilise. Il s'agit généralement d'un problème de disponibilité. Ces services n'étant pas directement contrôlés par l'application, leur disponibilité n'est pas garantie et leur accessibilité peut évoluer au cours de l'exécution (Figure 8). Ce type d'adaptation est tout particulièrement requis lorsque les applications dépendent d'objets physiques qui peuvent apparaître et disparaître dynamiquement, mais également dans les applications réparties où un service distant peut ne plus être accessible à n'importe quel moment.

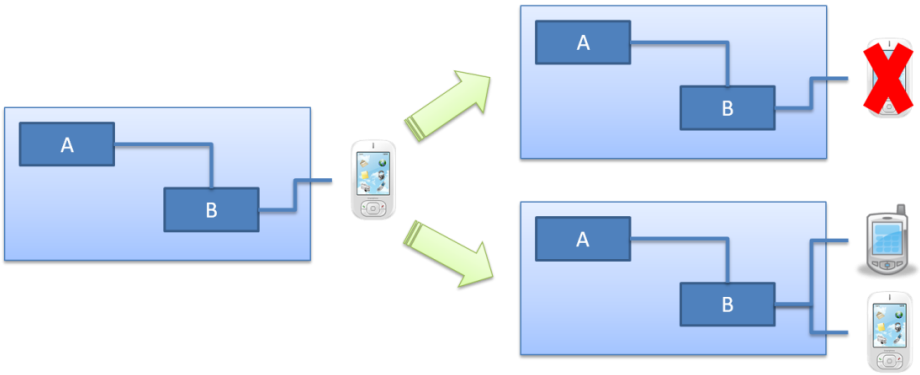


Figure 8. Adaptation provenant d'un changement environnemental :
Disparition (en haut) et apparition (en bas) d'un objet communiquant

[3] Dans cette thèse, nous différencierons l'environnement d'exécution d'une application (l'ensemble des ressources et des services disponibles) de son contexte ambiant d'exécution (utilisateur, environnement physique)

[4] Le terme « composant » est utilisé au sens large et ne cible pas exclusivement les approches à composants. Celles-ci seront traitées ultérieurement dans ce chapitre.

L'adaptation peut également venir d'un changement du contexte de l'application. Souvent l'adaptation est nécessaire, non pas parce que quelque chose a changé, mais par ce que quelque chose doit changer. En effet, une application peut être sensible à son contexte d'exécution (défini par l'application) et ce contexte peut être amené à évoluer durant l'exécution [62]. Ce genre d'adaptation s'illustre particulièrement dans les applications dont le contexte prend en compte l'utilisateur de l'application. Par exemple, une application requérant un service présent dans la même pièce que l'utilisateur devra remplacer ce service lorsque l'utilisateur changera de pièce (Figure 9). Pour cela, l'application doit surveiller en permanence la location de l'utilisateur afin de s'adapter dès que celui-ci se déplace.

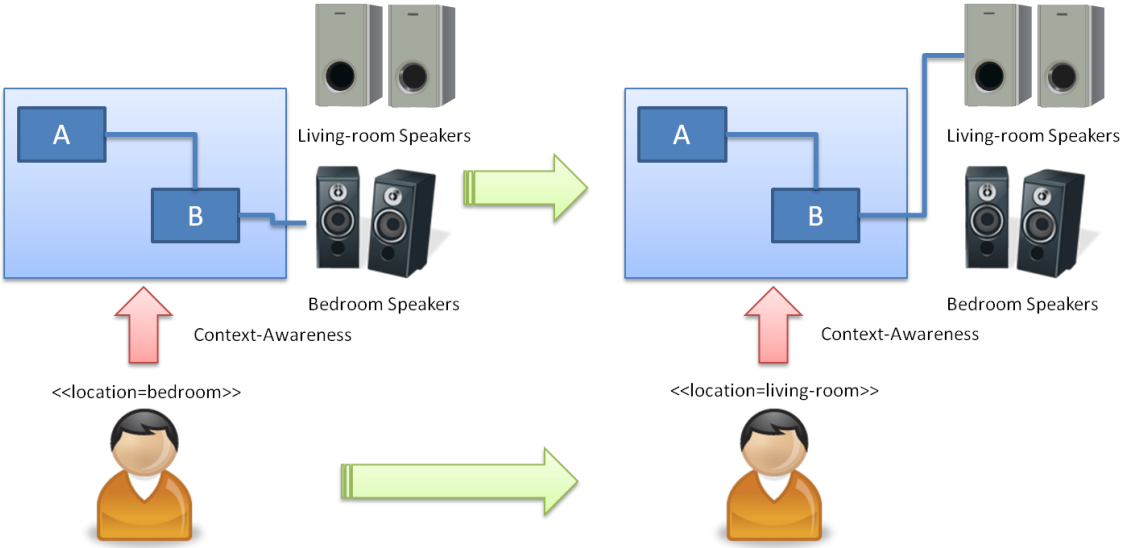


Figure 9. Dynamisme provenant d'un changement de contexte

2. Applications dynamiques : définition & généralités

Une application dynamique est une application qui est adaptée durant son exécution. Les applications dynamiques font donc parties des applications adaptables. Cependant, de nombreux travaux ont montré que la prise en charge de ces adaptations impacte directement l'architecture de l'application [63-66], ce qui en fait une catégorie très spéciale des applications adaptables. L'architecture d'une application est la structure des composants constituant l'application et les interactions entre ces composants [67, 68]. Une architecture peut être décrite à l'aide d'un langage de description d'architecture (ADL) [69].

Il apparaît que la gestion du dynamisme influence à la fois les composants utilisés par une application, mais également leurs interconnexions. Afin de pouvoir supporter le dynamisme, une application doit alors supporter la modification de son architecture. Pour s'adapter à un nouvel environnement d'exécution ou à un changement de contexte, il est en effet nécessaire d'ajouter, de supprimer, de remplacer ou de configurer des composants et/ou des interactions. Nous pensons dès lorsque, de façon à supporter le dynamisme, une application doit être implémentée de manière à supporter l'adaptation de son architecture, c'est-à-dire fournir des mécanismes permettant de faire évoluer les composants et les connecteurs utilisés.

De nombreux systèmes permettent de créer des applications dynamiques. Cependant, ce dynamisme est rarement une préoccupation fondamentale et se retrouve géré de manière limitée. De plus, la politique d'adaptation est exprimée de manière très versatile. Ceci peut aller de l'absence de politique, à une description dans l'architecture de l'application en passant par du codage dans l'application.

Cette section positionne les applications dynamiques par rapport aux applications adaptables et définit quelques concepts relatifs aux applications dynamiques telles que la reconfiguration dynamique, le gestionnaire d'adaptation et la variabilité.

2.1. Applications adaptables et dynamisme

Les applications adaptables sont des applications qui sont transformées en fonction de leur contexte d'exécution. Il existe plusieurs moments pour adapter une application. Plus l'adaptation est faite tardivement plus elle peut être efficace, mais, également, plus elle est complexe à mettre en place et à garantir. Lorsque ces adaptations sont effectuées à l'exécution, alors l'application est dite dynamique (Figure 10). Cette section reprendra la classification des applications adaptables de McKinley [48] et distinguera les applications dynamiques des applications adaptables.

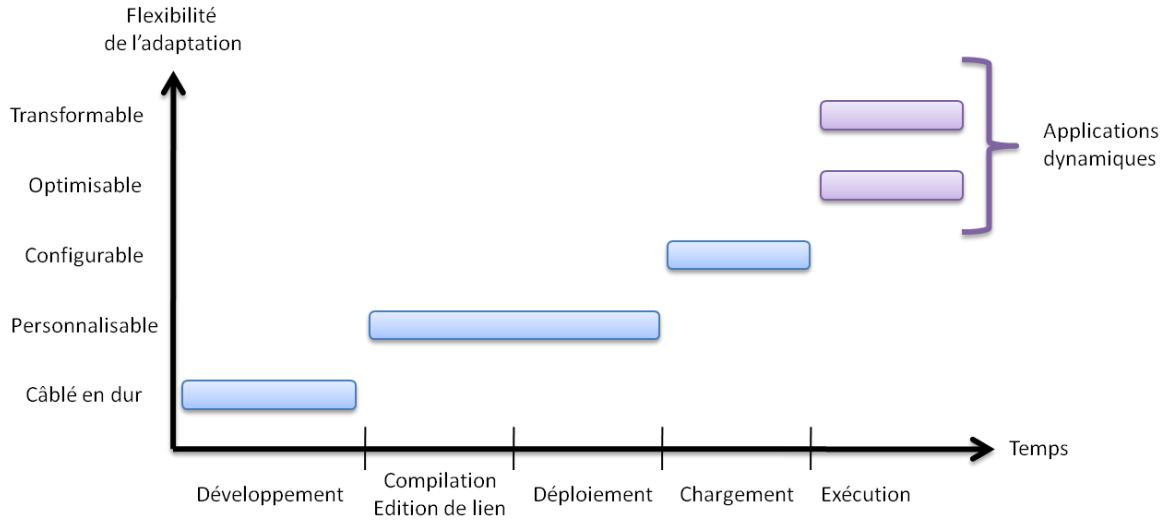


Figure 10. Positionnement des applications dynamiques par rapport aux applications adaptables

Le premier niveau d'adaptation est lorsque le code de l'application est modifié afin de satisfaire un nouvel environnement d'exécution. Cette adaptation ne peut être modifiée qu'en re-développant le logiciel.

Les applications dites « personnalisables » sont ajustées entre la compilation et le déploiement à leur environnement d'exécution. Il est possible d'adapter une application durant sa compilation (ou à l'édition de lien) en choisissant les bibliothèques à utiliser. Des pré-processeurs peuvent être employés afin d'effectuer ce type d'adaptation (Figure 11). Avec le développement de la programmation orientée aspect [70], il est possible d'adapter une application en lui ajoutant certains aspects au moment de la compilation de l'application. Le développeur peut choisir les aspects à intégrer. Afin de changer la logique d'adaptation, il suffit de changer les aspects combinés avec l'application. Au déploiement, il est envisageable de sélectionner les composants à utiliser en fonction de la cible sur laquelle le logiciel sera déployé. Cette approche est souvent utilisée dans les lignes de produits.

```
#if SYSTEM == MSDOS
    #include <msdos.h>
#else
    #include 'default.h'
#endif
```

Figure 11. Adaptation d'une application en fonction de l'OS utilisé à la compilation

Les applications « configurables » sont adaptées durant leur lancement ou leur chargement. L'application est alors configurée afin de satisfaire certains besoins. Par exemple, si un utilisateur démarre une application sur son PDA, le système d'exécution peut choisir les composants à charger en fonction de la résolution de l'écran du PDA.

Les différents types d'applications adaptables cités auparavant dans cette section ne sont pas des applications dynamiques. En effet, leurs adaptations précèdent leur exécution. Lorsque cette adaptation a lieu à l'exécution, nous pouvons distinguer deux classes d'applications :

- Les applications optimisables
- Les applications transformables

Les logiciels optimisables ne permettent pas la modification du code métier de l'application. Cependant, ils supportent leur adaptation lorsque le contexte d'exécution change en leur réassignant de nouveaux paramètres. Par exemple, lorsque la résolution de l'écran change, l'application peut choisir d'utiliser un autre composant d'affichage (plus adapté à la nouvelle résolution) où reconfigurer le composant utilisé. Le code de l'application (c'est-à-dire les composants formant le logiciel) n'est pas impacté. Ce changement a un impact direct sur l'architecture de l'application.

Les logiciels transformables permettent également la modification du code métier de l'application. Les logiciels supportant le remplacement des composants internes à l'application de manière dynamique rentrent dans cette catégorie. Par exemple, Open ORB [71] permet la mise à jour et la reconfiguration d'une application CORBA de manière dynamique. Cet intergiciel permet le remplacement des composants internes à une application sans redémarrer cette application. Bien que les possibilités soient très puissantes, dans la plupart des cas, le développeur doit restreindre ces possibilités afin d'assurer l'intégrité du système [66].

2.2. Reconfiguration dynamique

Dans les applications transformables, le procédé de modification de leur architecture est appelé ***reconfiguration dynamique***. La reconfiguration dynamique permet de modifier l'architecture d'une application durant l'exécution [60]. L'objectif celle-ci est de faire évoluer le système tout en garantissant l'intégrité du système. Une reconfiguration dynamique est généralement faite de manière déclarative (script) et manipule la structure de l'application, souvent appelée ***configuration*** [72]. Les changements sont effectués par un gestionnaire d'adaptation (Figure 12) et doivent laisser le système dans un état cohérent tout en évitant d'interrompre trop longtemps le système.

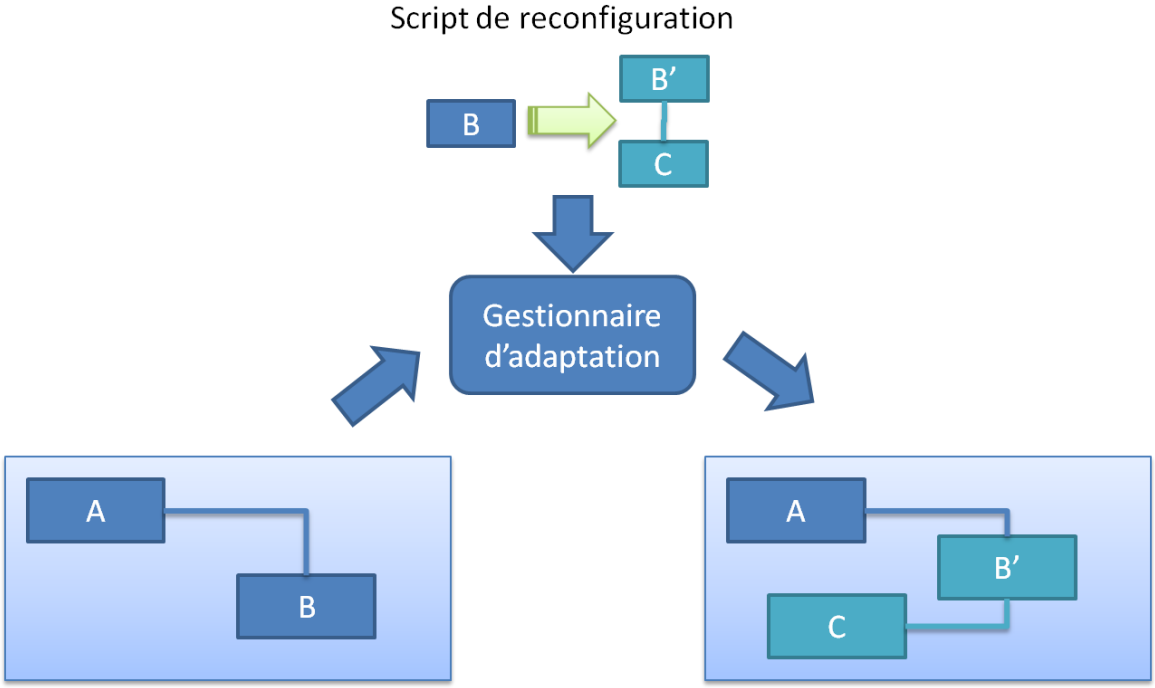


Figure 12. Exemple de reconfiguration

Les actions communément acceptées lors d'une reconfiguration sont [60, 72] :

- L'ajout d'un composant
- Le retrait d'un composant
- La création d'une liaison entre deux composants
- La destruction d'une liaison entre deux composants

De plus, deux autres actions sont généralement nécessaires afin de garantir l'intégrité du système :

- La passivation d'un composant
- L'activation d'un composant

```
Passiver B
Délier A et B
Supprimer B
Ajouter B'
Ajouter C
Lier A à B'
Lier B' à C
Activer C
Activer B'
```

Figure 13. Exemple de script de reconfiguration

La passivation d'un composant doit bloquer toutes les transactions entrantes afin de permettre la manipulation de ce composant. En effet, tout contact avec l'extérieur doit être empêché durant la reconfiguration du système afin d'éviter les états incohérents. Lorsque plus aucune transaction ne s'exécute sur un composant, il est dit ***quiescent*** et peut alors être manipulé par la reconfiguration en cours. Lorsque celle-ci est terminée, les composants passivés ou ajoutés sont (ré)-activés afin de (ré)-accepter des transactions entrantes (Figure 13).

L'obtention de la ***quiescence*** est coûteuse en temps d'interruption de service et demande un contrôle total du système afin de détecter les transactions. Récemment, des travaux tendent à limiter ce coût en proposant l'état de « tranquillité ». Cet état peut être obtenu plus rapidement que la quiescence [73, 74] et suffit pour la plupart des reconfigurations. Wermelinger propose également une autre approche en gelant des régions d'applications [75]. Cependant, bien que toutes ces propriétés soient atteignables lors de la gestion de l'évolution de l'application, elles sont très difficiles à garantir lors du traitement des autres types de dynamisme. En effet, afin d'obtenir la quiescence, la tranquillité ou le gel, le gestionnaire d'adaptation doit pouvoir détecter et bloquer toutes les transactions impliquant des composants touchés par la reconfiguration avant de la mettre en œuvre. Or, lors d'un changement de l'environnement d'exécution ou de contexte, le gestionnaire d'adaptation n'est pas à l'origine de la reconfiguration et donc ne peut pas passiver le composant à temps.

Toutes les approches permettant de créer des applications dynamiques gèrent différemment la reconfiguration dynamique. En effet, il n'est pas rare de rencontrer des applications où il n'est pas possible de supprimer des composants. Par exemple, l'IDE Eclipse ne permet pas la désinstallation de plug-in de manière dynamique. Il faut supprimer les plug-ins manuellement lorsque le logiciel est arrêté, puis redémarrer. Dans les applications à plug-ins, les liaisons entre l'application et les plug-ins sont souvent gérées par l'application elle-même et ne sont pas manipulables. L'activation et la désactivation ne sont pas utilisables non plus. Le processus est automatisé dans l'application.

Certains travaux ne se limitent pas à ces opérations et en proposent d'autres. Deux types d'actions sont généralement ajoutés. Tout d'abord, il est possible d'ajouter des opérations *composites* automatisant les opérations de base. L'opération « remplacer un composant » peut, par exemple, être exprimée avec les opérations de base. La séquence d'opération : « passivation, suppression, ajout, activation » est une implémentation possible de « remplacer un composant ». Ensuite, dans certains contextes ces opérations ne sont pas suffisantes. Les applications distribuées rajoutent généralement des opérations de placement afin de reconfigurer la géométrie de l'application [61, 76].

2.3. Gestionnaire d'adaptation

Le gestionnaire d'adaptation est une entité logicielle exécutant les reconfigurations dynamiques du système tout en garantissant son intégrité. Les gestionnaires d'adaptation sont généralement formés de deux parties [77, 78] :

- Une partie supervision
- Une partie contenant la logique de reconfiguration

La partie supervision surveille les composants de l'application, les connecteurs reliant ces composants [78] et peut également avoir en charge la collecte d'évènements publiés par les constituants de l'application[79]. La partie contenant la logique de reconfiguration décrit les adaptations à effectuer sur le système (logique d'adaptation), c'est-à-dire quand adapter et comment adapter.

Ce découpage se retrouve également dans les gestionnaires *autonomiques* (présentés page 25) ou la partie supervision est constituée du moniteur et de l'exécuteur alors que la partie contenant la logique de reconfiguration est constituée de l'analyseur et du planificateur.

Alors que le gestionnaire d'adaptation d'un système adaptable contient seulement la partie reconfiguration et une interface permettant d'invoquer les opérations de reconfiguration dynamique. Dans ce cas, la logique d'adaptation est fournie par une entité extérieure (utilisateur, administrateur …), et est par nature modifiable à l'exécution. Dans le cas des applications auto-adaptables, les adaptations sont effectuées automatiquement. Dans ce cas, le gestionnaire d'adaptation contient à la fois la partie supervision et la partie contenant la logique d'adaptation.

L'infrastructure de supervision peut être centralisée ou distribuée [80]. Différentes variations concernant les gestionnaires d'adaptation et les politiques d'adaptation seront étudiées dans ce chapitre.

2.4. Degré de variation

Un des points importants différenciant les approches pour créer des applications dynamiques concerne le degré de variation autorisé c'est-à-dire l'impact des variations sur le système. Nous distinguons deux degrés différents en fonction de la connaissance du système des éventuelles adaptations (Tableau 1).

Le premier degré de variation est lorsque le système connaît les adaptations potentielles avant son exécution. Il s'agit de l'approche généralement utilisée dans les lignes de produits où l'assembleur peut choisir les options à installer [81, 82]. Cette approche se base sur des points de variabilité sur lesquels l'assembleur peut agir afin d'adapter l'application à des contraintes. On peut obtenir ainsi des applications « sœurs » (famille d'applications ou famille de produits) qui partagent un socle commun, mais qui diffèrent dans les options sélectionnées. Bien que cette adaptation soit généralement utilisée avant l'exécution, il est possible de faire ces choix durant l'exécution. L'application est alors une application dynamique.

Le deuxième degré de variabilité est atteint lorsque le système ne connaît pas les adaptations éventuelles, ni qui l'adapte. Dans cette catégorie, on retrouve les applications à plug-ins. Une application à

plug-ins définit ces points d'extension. Ensuite, il est possible d'étendre l'application en fournissant un plug-in. Celui-ci ajoute une fonctionnalité dans le cadre défini par l'application. De nombreuses applications utilisent ce principe dans de nombreux domaines tels que des environnements de développement (Eclipse, Netbeans), des logiciels de graphisme (The Gimp, Adobe Photoshop), des navigateurs web (Mozilla Firefox, Microsoft Internet Explorer)… Ce degré contient également les systèmes d'exploitation (fournissant les mécanismes pour installer et désinstaller des applications) et les serveurs d'applications. Les applications hébergées ont beaucoup plus de possibilités que les plug-ins habituels. Dans tous ces cas, le logiciel fournit un mécanisme d'extension. Ensuite des plug-ins ou des applications (pas forcément fournis par l'entreprise ayant développé le logiciel) se greffent sur le logiciel de base afin de l'étendre.

Degré de variation	Définition	Exemples caractéristiques
1	L'ensemble des adaptations est connu, mais ces adaptations sont effectuées à l'exécution.	Lignes de produits dynamiques (points de variation prédéfinis, mais sélectionnés à l'exécution)
2	Les adaptations ne sont pas connues avant l'exécution. Le système ne sait pas qui l'adapte	Architecture à plug-in (points d'extension définis), Systèmes d'exploitation, Serveurs d'applications

Tableau 1. Tableau récapitulatif des degrés de variation

3. Caractérisation des approches

La précédente section a présenté les applications dynamiques et les concepts relatifs à cette classe d'application. De nombreux travaux proposent des classifications abordant le dynamisme [48, 53, 72]. Cependant, nous nous intéresserons plus particulièrement aux variations du gestionnaire d'adaptation c'est-à-dire :

- La description des adaptations
- La gestion des adaptations à l'exécution

Le reste de ce chapitre a pour objectif de définir une caractérisation permettant de classifier les approches pour implémenter et exécuter des applications dynamiques. Les critères définis dans cette caractérisation nous permettrons de positionner différents travaux sur la description et l'exécution d'applications dynamiques.

3.1. Critères

Depuis les années 70, de nombreux travaux se sont intéressés à créer des applications dynamiques. Cependant, les solutions proposées sont très hétérogènes. Afin de classifier ces approches, nous définissons six critères permettant de caractériser les approches permettant de créer des applications dynamiques. Ces critères sont répartis en deux sous catégories : ceux s'intéressant à la logique d'adaptation mise en place pour supporter le dynamisme et ceux se focalisant sur l'infrastructure gérant les adaptations.

Les applications dynamiques subissent des adaptations. Cependant, ces adaptations sont généralement régies par une politique d'adaptation décrivant quand et comment l'application est adaptée. Un point important discriminant les approches permettant d'exécuter des applications dynamiques concerne la description et la gestion de cette politique. Nous distinguerons trois critères concernant les logiques d'adaptations :

- Les déclencheurs des adaptations : Ce critère s'intéresse aux évènements déclenchant les adaptations. En effet, celles-ci peuvent être déclenchées par administrateur ou un utilisateur, par une modification de l'environnement d'exécution ou par un évènement interne à l'application.
- La localisation de la logique d'adaptation : Afin de piloter l'adaptation, plusieurs alternatives sont possibles. Tout d'abord, il peut ne pas y avoir de politique lorsqu'un administrateur (humain) gère l'adaptation. D'un autre coté, lorsqu'elle existe, cette politique peut être noyée dans le code, décrite et externalisée, ou bien infusée dans la description de l'architecture.
- La gestion des politiques d'adaptation : Ce dernier critère s'intéresse aux capacités du système à supporter la modification de la politique d'adaptation.

A l'exécution, le gestionnaire d'adaptation supervise et modifie l'application en fonction d'une politique d'adaptation. La seconde catégorie de critère se focalise sur l'exécution de cette adaptation. Nous distinguons trois critères concernant ce processus :

- Le positionnement du gestionnaire d'adaptation par rapport à l'application : Le gestionnaire d'adaptation peut être dans l'application, ou externalisée.
- Les mécanismes utilisés pour l'adaptation : Aujourd'hui, il existe de nombreux mécanismes permettant l'adaptation. Ce critère liste les différentes approches (protocoles méta-objet, aspects …) utilisées afin d'adapter l'application.
- La gestion des interruptions : L'adaptation d'une application dynamique peut entrainer une interruption de service. Cette interruption doit être minimisée afin de limiter l'impact de l'adaptation sur les utilisateurs. Ce critère distingue les différents niveaux d'interruption.

3.2. Qu'est-ce qui initialise l'adaptation ?

Le but de ce critère est de définir quand est adaptée une application. En fonction des approches, il n'est pas possible d'adapter au même moment. Ce premier critère concerne donc les stimuli (c'est-à-dire les événements) qui vont déclencher des adaptations. Ces adaptations peuvent être initialisées soit de manière externe et consciente (administrateur, configuration de l'utilisateur), soit suite à un événement externe issu de l'environnement d'exécution ou du contexte, soit suite à une décision de l'application elle-même.

La première catégorie de déclencheurs concerne les adaptations initialisées à l'extérieur de l'application de manière consciente (Figure 14). L'adaptation est initiée par un administrateur (humain ou virtuel) ou par un utilisateur voulant reconfigurer l'application. Ce type de déclencheur est très commun. De nombreuses applications permettent d'être reconfigurées. Cette reconfiguration peut être exprimée à divers niveaux comme la liste des options à déployer ou la liste des composants à ajouter … Un élément important de ce déclencheur est que la reconfiguration est effectuée de manière consciente ; c'est-à-dire par un administrateur / utilisateur sachant qu'il est en train de reconfigurer son logiciel. Ceci est important, car cela simplifie beaucoup la vérification de la reconfiguration. Le gestionnaire d'adaptation n'a pas à vérifier l'intégrité de l'application.

La deuxième catégorie d'événement pouvant déclencher une adaptation contient les événements externes issus de l'environnement d'exécution et des changements de contexte (Figure 15). Les événements provenant de l'environnement d'exécution sont utiles pour traquer la disponibilité des services utilisés par l'application. Cependant, quand un service utilisé s'en va ou réapparaît, l'application doit réagir. Cette catégorie contient également les événements provenant du contexte surveillé par l'application. Par exemple, un utilisateur peut se déplacer. De ce changement peut résulter une adaptation afin de s'adapter au nouveau contexte de l'utilisateur. Deux différences majeures distinguent cette catégorie de la précédente. Bien que ces événements proviennent également de l'extérieur de l'application, l'application doit réagir seule sans la

supervision d'un administrateur. Ceci rend beaucoup plus complexe la vérification de l'intégrité de l'application après l'adaptation.

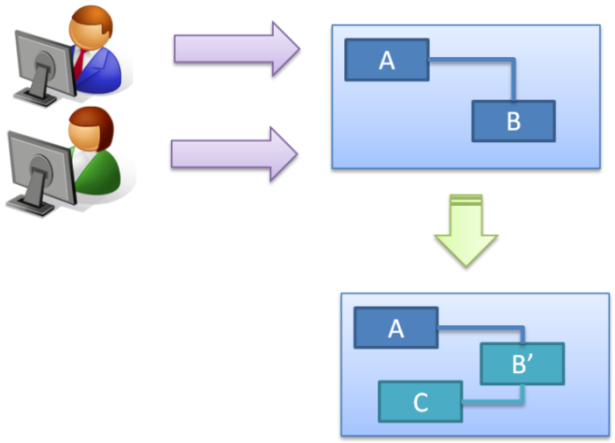


Figure 14. Acteur externe déclenchant l'adaptation de l'application

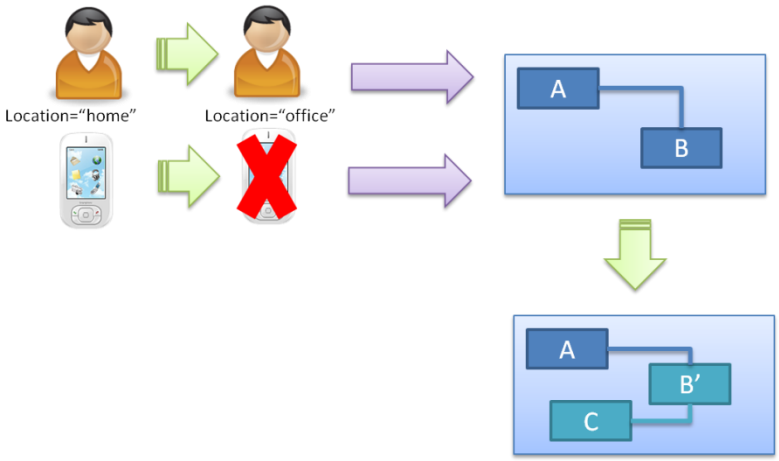


Figure 15. Adaptation déclenchée par un changement de l'environnement d'exécution ou par un changement de contexte

L'application peut elle-même décider de s'adapter (Figure 16). Dans ce cas, l'application se supervise elle-même afin de décider quand s'adapter et comment s'adapter. Ce genre d'adaptation est souvent utilisé dans les applications auto-adaptables où l'application s'optimise en fonction de son état ou d'un changement de contexte interne. Par exemple, une application dotée d'un détecteur de panne pourra se reconfigurer lorsqu'une panne ou un dysfonctionnement est détecté. L'événement à l'origine de la reconfiguration est interne puisque l'application elle-même a détecté sa panne. Ce type d'adaptation est plus rare, car l'application doit avoir la capacité de s'analyser elle-même et d'agir sur son architecture.

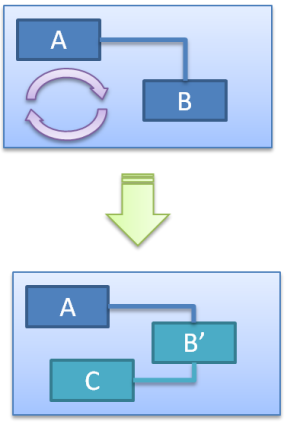


Figure 16. Adaptation d'une application provenant d'un stimulus interne

Les approches pour la création d'application dynamique peuvent supporter une ou plusieurs de ces catégories (Tableau 2). En fonction des catégories supportées, le dynamisme sera plus ou moins difficile à mettre en place.

Source des évènements déclenchant des adaptations	Exemples d'événements	Exemples typiques d'application
Événements externes supervisés par un administrateur ou un utilisateur	Administrateur ou utilisateur reconfigurant l'application	Applications configurables
Événements externes provenant de la modification de l'environnement d'exécution ou d'un changement de contexte	Arrivée ou départ d'un dispositif physique, disponibilité d'un service tiers, changement du contexte de l'utilisateur	Applications sensibles au contexte, Applications ubiquitaires
Événements internes	Panne, Disfonctionnement, Changement de contexte interne	Systèmes autoréparables, systèmes auto-optimisables

Tableau 2. Exemples d'applications en fonction des évènements initialisant l'adaptation

3.3. Où est décrite la logique d'adaptation ?

La logique d'adaptation doit répondre aux deux questions suivantes : quand et comment est adaptée l'application. Ce critère s'intéresse à la deuxième question et plus particulièrement la localisation de l'information décrivant les adaptations, c'est-à-dire à la localisation de la politique d'adaptation. Il existe plusieurs variations. Celle-ci peut ne pas exister (système adaptable) ou être codée, chargée voir exprimée dans l'architecture de l'application (Figure 17).

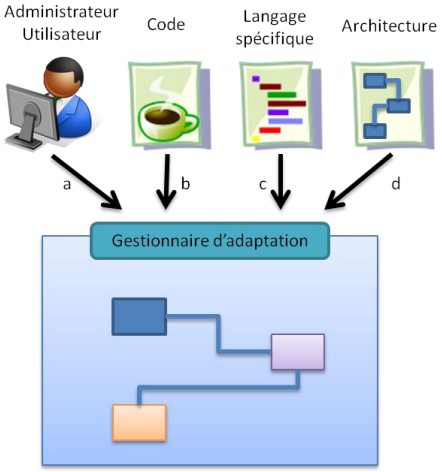


Figure 17. Gestionnaire d'adaptation et logique d'adaptation

L'option la plus souvent utilisée consiste à ne pas supporter de reconfiguration automatique. Toutes les adaptations sont effectuées par un administrateur ou un utilisateur de manière *ad hoc* en utilisant l'interface de contrôle fourni par le gestionnaire d'adaptation (a). En fonction des systèmes, cette adaptation peut se faire de manière interactive ou via l'exécution d'un script. La plupart des applications à plug-in choisissent cette option. L'utilisateur choisit les plug-ins à installer. Il a donc un contrôle complet sur les adaptations et offre une solution flexible. Cependant, elle est inapplicable pour les applications devant réagir rapidement à des événements provenant de l'environnement d'exécution.

Les autres alternatives ont pour but de rendre une application auto-adaptable, c'est-à-dire s'adaptant toute seule sans «*humain dans la boucle»* [83] et donc leur gestionnaire d'adaptation contient une partie reconfiguration. Une technique pour créer des applications auto-adaptables consiste à coder la politique d'adaptation dans le gestionnaire d'adaptation (b). Cette politique est donc fixée et difficilement changeable dynamiquement. De plus, l'utilisateur ou l'administrateur n'a plus aucun contrôle sur les adaptations. Cependant, cette option permet de réagir à des évènements de l'environnement ambiant. De nombreuses applications sensibles au contexte utilisent cette option. Cependant, elle ne permet pas de prendre en compte d'événements nouveaux non prévus initialement et peut devenir obsolète lorsque le contexte dépasse le cadre défini à la conception. De plus, la modification de la politique demande de redévelopper le code contenant celle-ci.

Afin d'éviter les problèmes de cette seconde alternative, il est possible d'externaliser la politique du code en effectuant l'adaptation en fonction de cette politique. Le gestionnaire d'adaptation chargera cette politique et l'exécutera (c). Bien que généralement plus générique que les politiques codées, cette approche permet une réelle séparation entre le gestionnaire d'adaptation et la politique elle-même. De plus, il est généralement possible de faire évoluer cette politique sans faire évoluer le code l'exécutant. La politique peut prendre plusieurs formes telles qu'un fichier XML ou un fichier dans un langage spécialisé et est généralement exprimé par des règles d'adaptation (type règle Évènement – Condition – Action) comme dans [76]. Cette approche est donc plus flexible que l'approche précédente. Il s'agit de l'approche généralement utilisée dans les applications *autonomiques* où un gestionnaire administre un système en fonction d'une politique d'administration [22, 84, 85]. Un aspect intéressant est que les gestionnaires d'adaptation des applications *autonomiques* transforment une application adaptable en une application auto-adaptable.

Bien que l'externalisation de la politique d'adaptation soit plus avantageuse que le codage « en dur », elle ne fait pas forcément partie de l'application. Cela pourrait permettre de rendre une application « standard » dynamique, mais il est très difficile de rendre dynamique une application qui n'a pas été développée pour. De plus, les politiques doivent être développées en fonction des architectures des applications gérées. Pour combler ces lacunes, la dernière alternative propose d'exprimer la politique de gestion du dynamisme dans l'architecture de l'application elle-même. En effet, il parait sensé d'exprimer cette politique dans la description de l'architecture puisque le dynamisme modifie cette architecture. Cette politique peut être exprimée de différente manière soit en considérant l'architecture de l'application comme un *style* qui doit être respecté en permanence [86] comme par exemple en utilisant des grammaires de graphes, soit en décrivant les opérations à effectuer en fonction des événements, soit en annotant l'architecture avec des contraintes à respecter [66]. L'avantage d'exprimer le dynamisme dans une architecture offre également la possibilité de vérifier que l'adaptation laisse l'application dans un état cohérent soit en vérifiant les contraintes de style, soit en effectuant diverses vérifications ou simulations. De nombreux travaux se sont intéressés à supporter le dynamisme dans l'architecture de l'application en utilisant différents moyens tels que les grammaires de graphes [87-89], l'algèbre de processus [90], des langages spécialisés [64, 83] ou bien un système d'évènements[91].

En fonction de l'endroit où est décrite la politique d'adaptation, la gestion du dynamisme peut être plus ou moins flexible (Tableau 3). Bien que l'absence de politique offre une gestion très flexible, elle n'est pas

applicable à tous les cas. L'expression de la politique dans le code du gestionnaire d'adaptation est très utilisée, mais n'est pas assez flexible dans de nombreux cas. La solution la plus utilisée est la description de la politique de manière séparée. Cependant, cette solution peut être difficile à mettre en place sur des applications dont l'architecture ne permet pas le dynamisme. De plus, la politique doit être exprimée en fonction de l'architecture de l'application. La dernière alternative est l'expression du dynamisme dans l'architecture de l'application. Cette solution apporte l'avantage de prendre en compte le dynamisme comme une propriété fondamentale au moment du design de l'application.

Localisation de la politique d'adaptation
Pas de politique
Politique code
Politique externalisée
Politique exprimée dans l'architecture de l'application

Tableau 3. Récapitulatif des critères concernant la localisation de la politique

3.4. Gestion des politiques d'adaptation

Un troisième point important concernant la logique d'adaptation est sa gestion durant l'exécution. En effet, dans le cadre d'applications auto-adaptables (donc régies par une politique d'adaptation) une question importante est : est-ce que le comportement du système et la politique d'adaptation peuvent évoluer dynamiquement? Pour répondre à cette question nous distinguerons deux types de systèmes [83]:

- Les systèmes fermés où il n'est pas possible de changer la politique d'adaptation
- Les systèmes ouverts où il est possible de changer de politique d'adaptation à l'exécution

Dans les systèmes fermés, le comportement du système et les adaptations sont fixés au déploiement de l'application. Dans les systèmes ouverts, le comportement du système peut évoluer lors de l'exécution ainsi que la logique d'adaptation. Changer la politique d'adaptation sert généralement à pouvoir prendre en compte des événements non prévus lors de la conception de l'application et de la politique d'adaptation. La mise en place d'un système ouvert est généralement délicate. Elle fait souvent appel à des technologies complexes comme le chargement dynamique de code. Plus particulièrement, le changement de la politique d'adaptation est extrêmement complexe comme le souligne [65]. Cette difficulté vient à la fois du fait que le changement de politique peut perturber le système (voir le laisser dans un état incohérent), mais également du fait qu'il faut généralement compléter le système avec des événements supplémentaires ou de nouveaux actionneurs utilisés par la nouvelle politique.

Une troisième alternative est envisageable. Il est possible de créer des applications mi-ouvertes : la politique d'adaptation ne peut pas être changée complètement, mais propose néanmoins des points de variations[5]. Un exemple simple serait un système où la politique d'adaptation s'exprime en fonction de variables dont les valeurs sont modifiables dynamiquement.

Gestion des politiques d'adaptations
Fixe (système fermé)
Variable (système semi-ouvert)
Changeable (système ouvert)

Tableau 4. Récapitulatif des options concernant la gestion des politiques d'adaptations

[5] Cette approche découle des principes des lignes de produits. Dans les lignes de produits les valeurs de ces variables seraient généralement choisies avant l'exécution.

3.5. Positionnement du gestionnaire d'adaptation

Le gestionnaire d'adaptation a pour but d'adapter l'application dynamiquement. Cependant, le gestionnaire d'adaptation peut être placé différemment par rapport à l'application. Quatre positions sont envisageables :

- Dans le code de l'application
- Dans un code fusionné avec celui de l'application
- Dans le support d'exécution de l'application
- De manière externe par rapport à l'application

3.5.1. Implantation du gestionnaire d'adaptation dans le code de l'application

Lorsque le gestionnaire d'adaptation est implanté directement dans l'application, le dynamisme est géré directement dans le code de l'application. Celui-ci contient alors toute la gestion de l'architecture de l'application et donc effectue sa propre reconfiguration. Cette approche est l'une des plus utilisées aujourd'hui, car elle permet une gestion très spécialisée du dynamisme. Par exemple, lorsque le contexte n'est plus favorable, une application peut décider de fonctionner en mode dégradé, c'est-à-dire d'offrir moins de fonctionnalités ou peut décider de s'arrêter.

Afin de gérer le dynamisme dans le code, il faut que l'application puisse être notifiée des changements de son contexte ou de son environnement d'exécution. Suite à cette notification, l'application peut réagir et adapter son comportement. Nous citerons ici deux approches généralement suivies lors de la mise en place de gestionnaires d'adaptation dans l'application.

La première approche consiste simplement à reconfigurer des paramètres pour modifier le comportement en fonction des événements du contexte. Par exemple, l'algorithme de TCP (protocole de contrôle de transmission utilisé sur Internet) ajuste son comportement en modifiant la taille de son historique et son délai de retransmission lorsqu'il suspecte une congestion du réseau. Dans ce cas, le gestionnaire d'adaptation détecte le problème et ajuste les paramètres.

Une deuxième approche pour gérer le dynamisme dans le code consiste à utiliser un patron de conception appelé « stratégie » [92]. Ce patron de conception est généralement utilisé pour changer d'algorithme, nommé *stratégie*, en fonction du contexte. Lorsque le contexte change, le gestionnaire d'adaptation détecte ce changement et choisit l'algorithme le plus adéquat en fonction du nouveau contexte (Figure 18). Cette approche est utilisée dans [93].

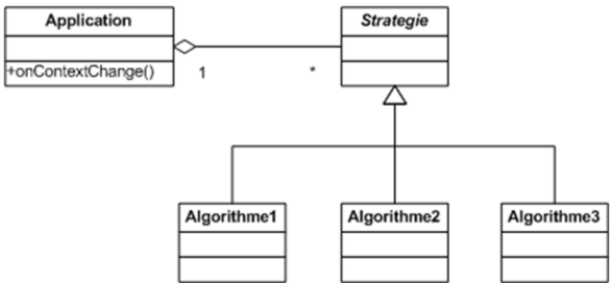


Figure 18. Schéma de principe du patron de conception "Stratégie"

Bien que gérer le dynamisme dans le code soit la méthode la plus employée, elle possède de nombreux défauts. Tout d'abord, elle est difficile à mettre correctement en place lorsque l'application s'exécute dans un environnement très dynamique. Ensuite, le code du gestionnaire d'adaptation est souvent dispersé dans le

code de l'application, ce qui ne facilite pas la maintenance de l'application. Et enfin, toutes les sources de dynamisme ne peuvent pas être gérées ainsi. En effet, l'évolution de l'application est difficilement gérable à partir de l'intérieur même du code de cette application.

3.5.2. Fusion du gestionnaire d'adaptation et du code de l'application

Afin de séparer le code gérant les adaptations du code de l'application, il est possible d'intégrer le code du gestionnaire d'adaptation et le code de l'application en utilisant la composition de filtre ou la programmation par aspect. Comme cette jonction n'est effectuée qu'à la compilation, le code de l'application n'est pas « pollué » par le code du gestionnaire d'adaptation.

La composition de filtre[6] est apparue au début des années 90 et avait pour objectif d'intercepter les interactions entre objets. Chaque filtre intercepte les messages reçus et envoyés entre les objets de l'application [94, 95]. Le filtre peut sélectionner les messages à intercepter et effectuer un traitement lorsqu'un message est capturé. Le filtre peut alors choisir l'objet à appeler en fonction du contexte, bloquer l'appel si l'objet visé n'est plus accessible… Des séquences de filtres peuvent également être mises en place, mais requièrent que l'ordre soit spécifié lorsqu'un filtre transforme les données du message. Les filtres sont développés séparément du code de l'application et sont ensuite compilés avec l'application. La composition de filtre est notamment utilisée dans les *servlets* Java [96].

L'approche la plus couramment utilisée pour ajouter du code non-fonctionnel au code d'une application consiste à utiliser la programmation par aspect (AOP) [97]. Cette approche est plus générique que les filtres qui se focalisent sur l'échange de messages entre deux composants. Les aspects permettent également d'intercepter des appels de méthodes à l'intérieur d'un même objet ou bien l'accès aux membres d'une classe (combinés avec le mécanisme de *superimposition* [98], les filtres permettent d'exprimer des aspects[95]). Aujourd'hui la programmation par aspect est devenue très populaire comme l'illustre le succès d'AspectJ, permettant d'appliquer les principes de l'AOP sur Java. Le principal intérêt de l'AOP est qu'il permet de séparer le code des aspects et le code métier de l'application. Ce n'est qu'à la compilation que ceux-ci seront réunis (tissage). Tout comme les filtres, les aspects peuvent contenir la gestion du dynamisme [99].

Les deux approches présentées ici peuvent être utilisées pour implémenter des gestionnaires d'adaptation [100, 101]. Plus particulièrement, elles permettent de contrôler les interactions ayant lieu dans l'application ce qui permet d'administrer le dynamisme provenant de l'environnement ou le changement de contexte. Cependant, tout comme l'implantation du gestionnaire d'adaptation dans le code, il est très difficile d'appliquer ces techniques pour contrôler l'évolution de l'application.

3.5.3. Gestionnaire d'adaptation au sein des modèles à composant

Bien que l'approche précédente permette de séparer le code de l'application du code du gestionnaire d'adaptation, ceux-ci sont réunis à la compilation et sont donc liés à l'exécution. Une autre approche consiste à séparer réellement le gestionnaire d'adaptation et l'application. Dans cette approche, l'application peut être supervisée par le canevas d'exécution. Nous allons voir comment les modèles à composant permettent de mettre en place cette approche.

La programmation par composants (Component-Based Software Engineering (CBSE)) propose la décomposition d'un système en entités fonctionnelles nommées *composants* [102]. Un composant possède des interfaces utilisées pour les interactions avec les autres composants. Une application est ensuite assemblée en sélectionnant les composants et en les liant [103]. Aujourd'hui de nombreux modèles à composant existent et sont utilisés dans de nombreux domaines. Nous pouvons citer les composants COM [104], les Enterprise

[6] La composition de filtre est une technique souvent mise en place pour gérer les problèmes de médiation.

JavaBeans [105], le canevas Spring [106] ou bien le modèle Koala [107] utilisé dans les lignes de produits chez Philips.

À l'instar de la programmation par objet, la programmation par composant manipule deux types d'entité : les types de composant et les instances de composant. Les types de composant sont similaires aux classes de la programmation par objet. Elles définissent un type composé généralement par des interfaces requises et fournies, l'implémentation, et la configuration des services techniques. Les types de composants permettent de créer des instances de composants qui seront les vraies entités « vivantes ».

Une approche couramment utilisée dans les approches à composant est l'utilisation de *conteneur*. Le conteneur peut être schématiquement représenté comme une coquille entourant le contenu de l'instance (Figure 19) afin de gérer les interactions avec les autres instances (via les interfaces fournies et requises), le cycle de vie et les différents services techniques requis de l'instance (via les interfaces de contrôle). Le contenu peut être un objet de l'implémentation défini dans la déclaration type de composant ou d'autres instances de composants (composition hiérarchique).

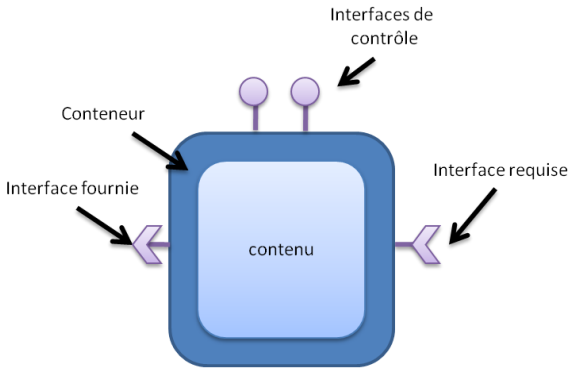


Figure 19. Schéma d'une instance de composant

L'un des avantages des conteneurs est qu'ils permettent de séparer le code métier (contenu de l'instance) des préoccupations non-fonctionnelles gérées par le conteneur et donc plus particulièrement de la gestion du dynamisme. La relation entre le conteneur et le contenu s'effectue en utilisant les motifs de conception d'inversion de contrôle et d'injection de dépendance [108]. Le conteneur gère totalement l'exécution du contenu. Il injecte dans le contenu les objets requis. Il gère également l'accès depuis l'extérieur au contenu. Ceci a été permis par le développement de la programmation « par interface » permettant d'abstraire un « service » de son implémentation [109-111]. Ainsi le contenu utilise « l'interface », alors que le conteneur injecte un objet implémentant cette interface.

Lorsque le modèle à composant supporte la composition hiérarchique, les applications sont décomposées en plusieurs (sous-) composants reliés par des connecteurs. Ces composants s'exécutent à l'intérieur du composant-parent englobant l'application. Le gestionnaire d'adaptation peut alors superviser toute l'application en se situant dans le conteneur englobant l'application (Figure 20).

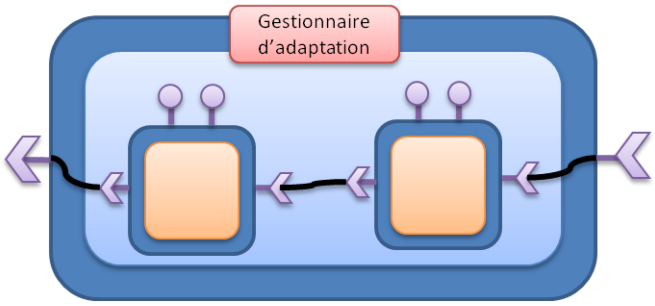


Figure 20. Gestionnaire d'adaptation dans une composition hiérarchique

De ce fait, il est possible de gérer toutes les sources de dynamisme. En effet, l'évolution peut être prise en compte en remplaçant les composants, l'évolution de l'environnement d'exécution peut être gérée en modifiant les connecteurs, enfin l'évolution de contexte peut également être gérée soit en modifiant les composants contenus soit en modifiant les connecteurs.

Cependant, tous les modèles à composant ne permettent pas l'implémentation d'un gestionnaire d'adaptation. En effet, les modèles à composant sont généralement utilisés pour architecturer une application comme dans [107] ou pour faciliter le développement, [105, 106] mais plus rares sont ceux supportant l'adaptation dynamique.

3.5.4. Gestionnaires d'adaptation externes

Bien que l'insertion du gestionnaire d'adaptation dans un modèle à composant permette de gérer correctement toutes les sources de dynamisme, une dernière localisation du gestionnaire est possible. Celui-ci peut être totalement découplé de l'application. Dans ce cas, le gestionnaire d'adaptation est une entité autonome et supervise l'application de manière externe. Le gestionnaire d'application ne fait alors pas partie du canevas d'exécution de l'application, mais peut cependant se reposer dessus. Cette approche est notamment utilisée dans l'informatique *autonomique* où le gestionnaire autonomique (gérant les adaptations) est une entité externe par rapport à l'application.

En externalisant le gestionnaire d'adaptation, il est possible de rendre dynamique une application adaptable. Le gestionnaire d'adaptation peut alors agir sur l'application via des interfaces de contrôle que l'application doit fournir.

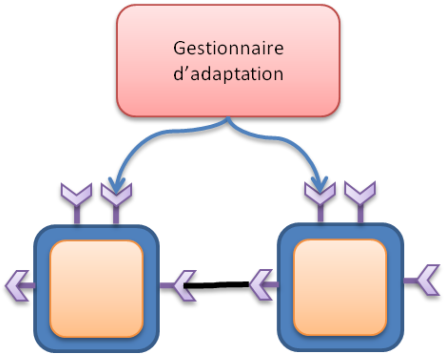


Figure 21. Gestionnaire d'adaptation externe

L'avantage de cette approche est qu'elle ne modifie pas un modèle à composant, mais repose sur des mécanismes offerts par l'application. Cette approche permet de gérer toutes les sources de dynamisme si elle dispose des mécanismes adéquats. Les approches dans lesquelles un humain (administrateur ou utilisateur) gère l'adaptation rentrent dans cette catégorie.

3.5.5. Synthèse sur la localisation du gestionnaire d'adaptation

Dans cette section, nous avons présenté quatre différentes localisations du gestionnaire d'adaptation. Ces localisations se différencient par la *distance* entre l'application et le gestionnaire (Figure 22), c'est-à-dire comment se positionne le gestionnaire d'adaptation par rapport au code de l'application. Cependant, le gestionnaire d'adaptation n'est pas forcément une entité compacte, mais peut être distribué. Une approche peut donc avoir plusieurs positionnements en fonction des différentes parties composant le gestionnaire d'adaptation.

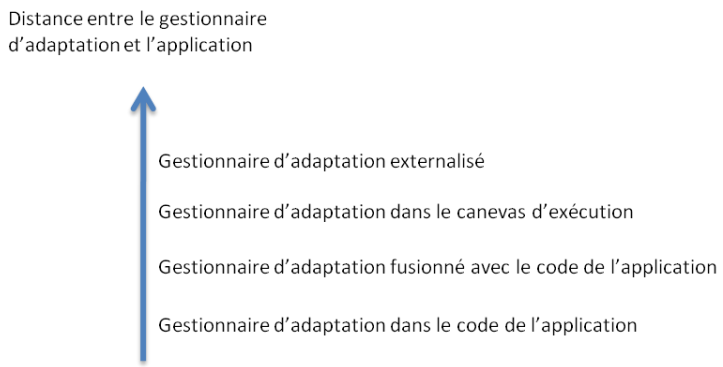


Figure 22. Positionnement du gestionnaire d'adaptation par rapport à l'application

3.6. Mécanismes utilisés pour la reconfiguration dynamique

Cette section explore les différents moyens mis en œuvre afin de permettre la gestion du dynamisme. En effet, le gestionnaire d'adaptation utilise des mécanismes sous-jacents afin de reconfigurer l'application. Tout d'abord, ces mécanismes peuvent être mis en place par l'application elle-même. Ensuite, nous présenterons brièvement les techniques mises en place au sein des systèmes d'exploitation. Nous continuerons par un tour d'horizon des apports des machines virtuelles pour la gestion du dynamisme. Nous terminerons par une étude des moyens proposés par les modèles à composant supportant la reconfiguration dynamique.

3.6.1. Utilisation de mécanismes internes à l'application

L'application peut elle-même proposer des mécanismes internes pour sa reconfiguration. Cependant, ce type de mécanisme demande que le gestionnaire d'adaptation soit spécialement codé pour reposer sur les possibilités fournies. Donc, cette approche peut être utilisée lorsque le gestionnaire d'adaptation se situe dans le code de l'application, mais elle est difficilement utilisable autrement.

3.6.2. Utilisation de mécanismes fournis par le système d'exploitation

Le problème du dynamisme a été adressé depuis très longtemps dans les systèmes d'exploitation. Dans [44], Fabry suggère d'appliquer le principe de redirection dans Multics [112]. En effet, les systèmes d'exploitation peuvent être considérés comme des applications dynamiques. De plus, ils fournissent les bases pour construire des applications dynamiques. En effet, il est possible d'installer de nouvelles applications ou d'en désinstaller et différents moyens de gestion. Généralement, installer une application requiert simplement de la « déposer » dans le système de fichier du système d'exploitation. Cette application peut alors être exécutée et peut utiliser les ressources mises à sa disposition par le système. Par exemple, afin d'installer une nouvelle commande Shell sous Linux, il suffit de copier l'exécutable (un fichier binaire ou un script) dans le répertoire « /bin ». Ensuite, il suffit de taper le nom du fichier copié pour lancer cette nouvelle commande.

Bien que les systèmes d'exploitation fournissent des moyens pour installer et désinstaller des *composants*, construire une application dynamique au dessus d'un système d'application reste complexe et repose souvent sur des mécanismes *ad hoc* mis en place par l'application. Par exemple, le Shell cité précédemment se base sur l'introspection de répertoires indiqués dans une variable d'environnement. De plus, il est difficile de mettre en place une infrastructure permettant l'interception et la redirection de messages échangés entre deux composants, cette infrastructure étant généralement requise dans la gestion du dynamisme.

3.6.3. Machines virtuelles et interceptions

Le sens originel de machine virtuelle est la création de plusieurs environnements d'exécution sur un seul ordinateur dont chacun émule l'ordinateur hôte. Dans son second sens, maintenant devenu plus commun, une

machine virtuelle désigne un logiciel ou interpréteur qui isole l'application utilisée par l'utilisateur des spécificités de l'ordinateur c'est-à-dire de celles de son architecture ou de son système d'exploitation. Cette indirection permet au concepteur d'une application de la rendre disponible sur un grand nombre d'ordinateurs sans les contraintes habituelles au développement d'un logiciel portable tournant directement sur l'ordinateur. Aujourd'hui ce type de machine virtuelle est très répandu. La machine virtuelle Java et la machine virtuelle de .NET (nommée CLR pour Common Language Runtime) sont les plus connues. Cependant, il existe de nombreuses autres machines virtuelles telles que le moteur Zend pour PHP, Parrot pour Perl 6 et le lecteur d'animation Flash (Macromedia Flash Player). Les machines virtuelles sont très intéressantes et utilisées pour construire des applications dynamiques pour deux raisons principales :

- Il est possible de charger et de décharger du code dynamiquement
- Il est facilement possible d'introduire un mécanisme d'interception des communications

Ces deux besoins viennent du fait que pour remplacer un composant au sens d'une application dynamique il faut enlever le composant, ajouter le nouveau composant et rediriger les liaisons de l'ancien composant sur le nouveau[7].

La première caractéristique est fondamentale pour créer des applications dynamiques. En effet, tel que présenté précédemment dans cette section, une application dynamique doit pouvoir remplacer ces composants internes. Pour cela, il faut déployer le code du nouveau composant, le charger et l'instancier dans l'application. Il faut détruire l'ancien composant et décharger son code de la mémoire de la machine virtuelle. Aujourd'hui, le chargement dynamique du code est une fonctionnalité commune des machines virtuelles. Cependant, la capacité à décharger n'est pas aussi bien supportée [113].

La deuxième caractéristique fondamentale pour gérer la reconfiguration dynamique est la mise en place d'un mécanisme d'interception des communications. Celui-ci permettra de manipuler les liaisons entre les composants et de pouvoir bloquer ou rediriger les messages en cas de reconfiguration dynamique. Il existe plusieurs approches pour mettre en place ce mécanisme, mais toutes reposent sur les principes des protocoles à méta-objets (Metaobject Protocol[114] ou MOP).

La première approche pour mettre en place ce mécanisme consiste à utiliser directement la fonctionnalité de la machine virtuelle. En effet, de nombreuses machines virtuelles permettent d'utiliser la réflexion et permettent également de créer des proxys dynamiquement[115, 116]. Grâce à ces deux fonctionnalités, il est possible d'intercepter les messages échangés entre deux composants (en interceptant généralement les appels de méthodes). Au moment de l'interception, le choix peut être fait de changer la cible de l'appel pour utiliser un autre composant voir bloquer l'appel en attendant la fin de la reconfiguration dynamique.

Afin d'automatiser la mise en place de cette infrastructure, la deuxième approche généralise l'utilisation des principes du MOP soit en modifiant la machine virtuelle elle-même, soit en tissant des aspects *génériques* dans l'application permettant l'interception des messages. Par exemple, R-Java [117] modifie la machine virtuelle Java afin de rajouter des instructions en bytecode permettant l'interception de message et la réflexion comportementale. Prose [118] et IguanaJ [119] généralisent le tissage d'aspect à l'exécution afin de permettre l'interception. Enfin Trap/J [120] procède en deux passes. La première tisse des aspects génériques dans l'application avant l'exécution et ensuite propose un mécanisme à l'exécution permettant de manipuler les messages interceptés.

[7] Ces actions ne se feront généralement pas dans cet ordre afin de diminuer l'interruption de service. En effet, on préfèrera souvent installer le nouveau composant avant d'enlever l'ancien afin de n'interrompre le fonctionnement de l'application que durant la reconnexion des liaisons.

Bien que toutes ces techniques permises par les machines virtuelles permettent au gestionnaire d'adaptation de reconfigurer une application, son développement reste très complexe. En effet, que cela soit le chargement / déchargement de code ou l'interception, ils font appel à des connaissances avancées et sont difficiles à mettre en place correctement.

3.6.4. Modèle à composant supportant la reconfiguration dynamique

Une des techniques les plus répandues pour développer des applications consiste à utiliser un modèle à composants. Certains modèles à composant supportent la reconfiguration dynamique. Le gestionnaire d'adaptation peut utiliser des mécanismes pour manipuler les composants d'une application ou les connecteurs dynamiquement. Il s'agit de mécanismes de plus haut niveau que ceux présentés auparavant permettant la manipulation de la structure d'une application.

Aujourd'hui de nombreux modèles à composant supportent cette reconfiguration dynamique. Nous pouvons citer Lua [121], SOFA/DCUP [122], Fractal et son implémentation de référence Julia [123] ou bien DUCS [124].

De nombreux travaux traitants l'adaptation dynamique d'application reposent sur cette approche. En effet, elle permet d'abstraire la complexité technique (gérée dans le modèle à composant) et donc simplifie la mise en place du gestionnaire d'adaptation.

3.6.5. Synthèse des mécanismes utilisés pour la gestion du dynamisme

Cette section a brièvement présenté les mécanismes requis pour la gestion du dynamisme. En fonction des approches, ces mécanismes peuvent être combinés.

Positionnement des mécanismes	Exemple
Application	Mécanisme fourni par l'application
Système d'exploitation	Gestion d'application, système de fichier
Machine virtuelle	Interception, Réflexion
Modèle à composant	Reconfiguration dynamique des composants et des connecteurs

Tableau 5. Récapitulatif des mécanismes requis pour la gestion du dynamisme

3.7. Gestion des interruptions de service

Lorsque le gestionnaire d'adaptation reconfigure une application, il doit tenter de limiter le temps d'interruption de service. Cependant, nous avons vu précédemment que pour reconfigurer une application, il était nécessaire d'attendre l'état de quiescence afin de pouvoir reconfigurer l'application en toute sécurité. Cet état est long à atteindre et demande un contrôle complet sur l'exécution l'application (afin de détecter toutes les transactions). Durant la reconfiguration, l'application n'est pas forcément utilisable ou subit une augmentation de son temps de réponse. En fonction des approches, cette interruption de service est gérée différemment :

- L'application est intégralement gelée
- Seuls les composants et connecteurs impliqués sont gelés
- L'interruption est minimisée au maximum

La première possibilité est de geler l'intégralité de l'application durant la reconfiguration. Cette approche a pour avantage de garantir l'intégrité de l'état durant la reconfiguration, mais d'un autre côté l'interruption peut être longue et contraignante pour l'utilisateur.

Afin d'éviter le gel complet de l'application, il est possible de détecter les composants impactés par la reconfiguration et ne geler que cette partie (région) de l'application. Cependant, ceci oblige également à

bloquer toutes les transactions entrantes vers les composants de la région. La figure ci-dessous montre une reconfiguration suivant cette approche. Cette reconfiguration n'impliquant que l'instance B, seule cette instance sera gelée afin qu'elle devienne quiescente. Les interactions de A vers B seront bloquées le temps de la reconfiguration.

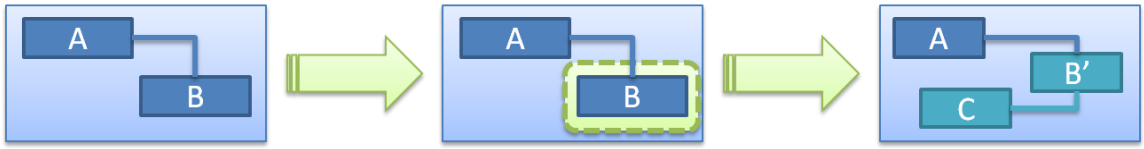


Figure 23. Adaptation par détection de région

Bien que l'alternative précédente permette de réduire considérablement l'impact d'une reconfiguration sur l'application, il est parfois possible de minimiser encore cet impact en ordonnançant la reconfiguration. La dernière alternative consiste à ne geler le fonctionnement d'un composant que si la reconfiguration le requiert réellement. Par exemple, lorsqu'un composant est remplacé par un autre, il n'est pas nécessaire de geler les composants utilisant le composant remplacé (Figure 24). En effet, ceux-ci peuvent directement utiliser le nouveau. Ceci permet de reconfigurer de manière transparente une application. Elle ne subit pas d'interruption de service.

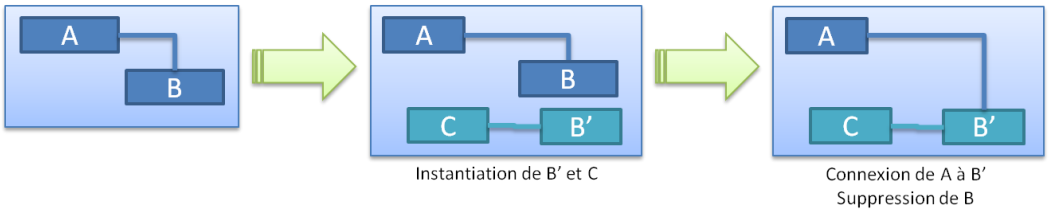


Figure 24. Minimisation de l'impact d'une reconfiguration

En fonction des reconfigurations, il n'est pas possible de minimiser l'impact. Dans d'autres cas, la région impactée par la reconfiguration est l'application entière. La valeur de ce critère est donc la possibilité optimale que supporte une approche.

3.8. Synthèse

Dans les sections précédentes, nous avons distingué six critères permettant de caractériser les approches permettant de créer des applications dynamiques. Ces critères sont répartis en deux sous catégories : ceux s'intéressant à la logique d'adaptation mis en place pour supporter le dynamisme et ceux se focalisant sur l'infrastructure gérant les adaptations. Afin de classer une approche, il faut sélectionner les valeurs pour chacun des critères. Certaines approches peuvent avoir plusieurs valeurs pour un même critère (initialisation des adaptations, positionnement du gestionnaire d'adaptation et mécanismes sur lesquels repose le gestionnaire d'adaptation). Les deux tableaux suivants rappellent les différents critères évoqués et les différentes possibilités distinguées :

Initialisation des adaptations	Localisation de la logique d'adaptation	Gestion des politiques d'adaptation
• Événements externes supervisés par un administrateur • Événements externes (modification de l'environnement d'exécution ou de contexte) • Événements internes	• Pas de politique • Politique codée • Politique externalisée • Politique exprimée dans l'architecture de l'application	• Système ouvert • Système fermé • Système semi-ouvert

Tableau 6. Caractérisation des approches en fonction de la logique d'adaptation

Positionnement du gestionnaire d'adaptation par rapport à l'application	Mécanismes utilisés pour l'adaptation	Gestion des interruptions
• Dans le code de l'application • Dans du code fusionné avec celui de l'application • Dans le canevas d'exécution • Gestionnaire d'application externe	• Mécanismes ad hoc • Mécanismes fournis par le système d'exploitation • Mécanismes fournis par une machine virtuelle • Mécanismes fournis par un modèle à composant	• Gel de l'application • Gel des composants impactés par la reconfiguration • Minimisation de l'interruption de service

Tableau 7. Caractérisation des approches en fonction du gestionnaire d'adaptation

4. Application de la caractérisation

La section précédente a défini des critères permettant de différencier les approches utilisées pour créer des applications dynamiques. Cette section utilise ces critères pour classer différents travaux. Nous analysons deux types de travaux :

- Les modèles à composant supportant la reconfiguration dynamique
- Les architectures dynamiques

4.1. Modèles à composant supportant la reconfiguration dynamique

De nombreux modèles à composant ont pour objectif de supporter le dynamisme. Cependant, tous ces modèles ne supportent pas forcément les mêmes types de dynamisme ou ne l'automatisent pas de la même manière.

4.1.1. Dynamically Updatable Component based System (DUCS)

DUCS est un modèle à composant supportant la mise à jour des composants de manière dynamique [124]. Le but de DUCS est de fournir une infrastructure pour créer des applications réparties où les composants peuvent évoluer sans arrêter l'application.

L'infrastructure de DUCS est composée de quatre entités. Le niveau de plus bas représente les nœuds. Les nœuds sont apparentés à une machine virtuelle (Java). Au dessus de ces nœuds se trouve la base de l'infrastructure fournissant l'environnement d'exécution des composants de l'application. La communication entre les composants utilise cette infrastructure. Au sein de cette infrastructure se trouve le gestionnaire de configuration gérant l'évolution sur le nœud. Ce gestionnaire de configuration fournit une interface afin de faire évoluer les composants du nœud. Cette évolution est commandée par une « requête de mise à jour » (*update request*). La dernière entité formant le canevas de DUCS est le gestionnaire d'architecture. Ce gestionnaire gère l'ajout, la suppression et la mise à jour de composants sur tous les nœuds formant l'application. Un aspect intéressant de ce modèle est le support du transfert d'état lors d'une mise à jour. À l'aide de fonctions de transfert, l'état d'un composant est réinjecté dans le composant le remplaçant.

DUCS permet donc de supporter l'évolution de composant, mais pas de supporter le dynamisme provenant de l'environnement d'exécution ou d'un changement de contexte. Le tableau ci-dessous positionne DUCS par rapport aux critères énoncés précédemment.

Critères	
Initialisation des adaptations	Événements externes supervisés par un administrateur
Localisation de la logique d'adaptation	Pas de politique, les adaptations sont rédigées par l'administrateur sous la forme de « *update request* ».
Gestion des politiques d'adaptation	Système fermé
Positionnement du gestionnaire d'adaptation	Dans le modèle à composant
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant, utilisation de la réflexion et du chargement dynamique de code (Java)
Gestion des interruptions	Minimisation des interruptions (support de différentes versions de composant), quiescence garantie par décompte du nombre d'invocations

Tableau 8. Positionnement de DUCS par rapport aux critères

4.1.2. SOFA/DCUP

SOFA/DCUP (DCUP pour « Dynamic Component Updating ») [122] est une extension au modèle à composant SOFA pour supporter la mise à jour automatique d'application. L'architecture de ce modèle repose sur SOFA, c'est-à-dire des SOFANodes (Figure 25) et un SOFANet (un réseau homogène de SOFANodes). Un SOFANode est un nœud du réseau composé de 5 parties. La partie « in » gère les requêtes entrantes. À l'inverse, la partie « out » gère les transactions sortantes. La partie « Template Repository (tr) » est un entrepôt contenant tous les types de composants disponibles pour ce nœud[8]. Les instances de composants sont créées par la partie « run » qui a en charge l'exécution des instances. De nouveaux types de composants peuvent être ajoutés à un noeud via la partie « made ».

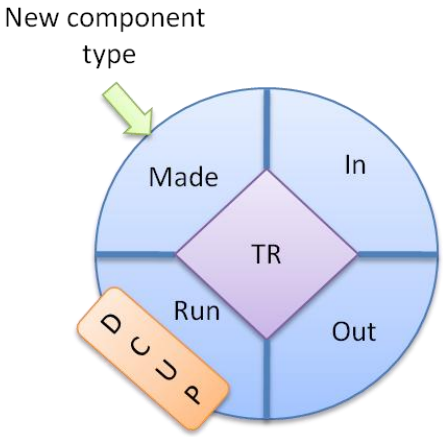


Figure 25. Structure d'un SofaNode

DCUP se greffe à la partie « run » afin d'automatiser la mise à jour dynamique de composant. Plus particulièrement, DCUP introduit un nouveau type de composant hiérarchique pouvant être remplacé dynamiquement, appelé « DCUP component » (Figure 26). Ceux-ci sont composés d'une partie remplaçable (le contenu) et d'une partie fixe (conteneur). Chaque instance est gérée par un « component manager » (CM sur la figure). Son but est de coordonner l'évolution et la restructuration du contenu de l'instance. La partie remplaçable est contrôlée par un « component builder » (CB sur la figure). Cet acteur doit gérer l'instanciation d'un type de composant, la destruction d'instance et le transfert d'état. Les évolutions des composants sont commandées par un gestionnaire de mise à jour (« updater »), qui reçoit les demandes de mises à jour (c'est-à-dire le remplacement de la partie remplaçable d'un composant par une version plus récente) et qui les délègue au « component manager » impliqué. Dès qu'une nouvelle version d'un type de composant utilisé est disponible, le gestionnaire de mise à jour est notifié et demande au « component manager » de mettre à jour les instances utilisées.

[8] En particulier, cet entrepôt gère les différentes versions disponibles d'un même type de composant.

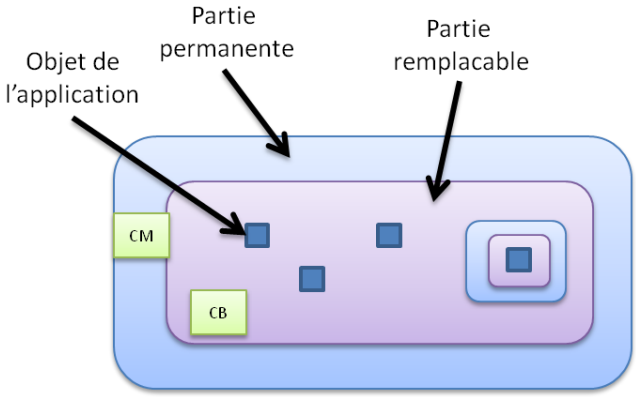


Figure 26. Schéma d'un composant DCUP

Tout comme DUCS, SOFA/DCUP se focalise sur le support à l'évolution des applications. Cependant, il ne traite pas le dynamisme provenant de l'environnement d'exécution ou d'un changement de contexte. Le tableau ci-dessous positionne SOFA/DCUP par rapport aux critères énoncés précédemment.

Critères	
Initialisation des adaptations	Évènements externes supervisés par un administrateur mettant à disposition une nouvelle version d'un type de composant
Localisation de la logique d'adaptation	Dans le modèle à composant
Gestion des politiques d'adaptation	Système fermé
Positionnement du gestionnaire d'adaptation	Dans le modèle à composant
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant, utilisation de la réflexion et du chargement dynamique de code (Java), mécanismes présents dans le code du composant l'activation et la désactivation d'une instance.
Gestion des interruptions	Minimisation des interruptions, les transactions entrantes sur une région en cours de reconfiguration sont bloquées

Tableau 9. Positionnement de SOFA/DCUP par rapport aux critères

4.1.3. K-Component

K-Component [65] est un modèle à composant au dessus de CORBA ayant pour but de créer des applications auto-adaptables. Ce modèle tend à séparer l'implémentation des composants (en CORBA) de la gestion du dynamisme. Une autre particularité de ce modèle à composant est que les adaptations sont directement effectuées sur l'architecture et reproduites sur l'application. Plus particulièrement, l'application est composée d'un ensemble d'instances de composant et de connecteurs. Elle est directement supervisée par un gestionnaire de configuration. Ce gestionnaire maintient en permanence le graphe des instances et des connecteurs composant l'application (Figure 27). La communication entre le gestionnaire de configuration et l'application se fait via des événements d'adaptations émis par l'application afin de remonter des informations, mais également émis par le gestionnaire de configuration afin de « commander » les reconfigurations.

Les reconfigurations sont décrites dans des contrats d'adaptation. Ces contrats sont exprimés avec un ensemble de règles conditionnelles permettant à la fois de reconfigurer l'architecture de l'application en fonction des événements émis par l'application et de définir des contraintes architecturales. La reconfiguration de l'architecture se fait en utilisant des opérations de reconfiguration fournies par le gestionnaire de configuration.

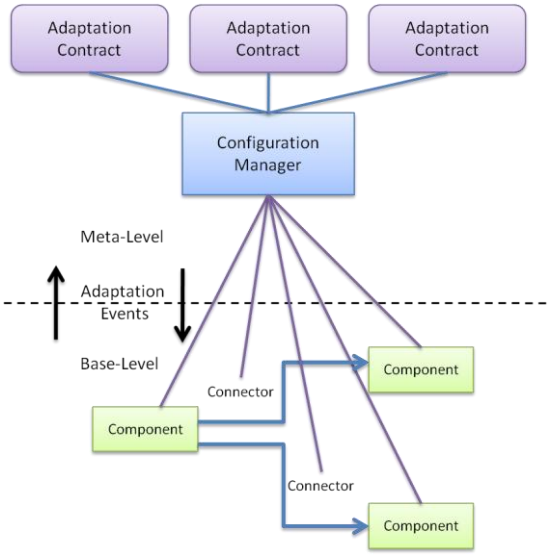


Figure 27. Fonctionnement de K-Component

K-Component propose donc un modèle à composant intéressant pour créer des applications dynamiques. En effet, il est capable de gérer toutes les sources de dynamisme. De plus, le fait de reconfigurer directement l'architecture de l'application permet d'exprimer les reconfigurations à l'aide de primitives de plus haut niveau. Le tableau ci-dessous positionne K-Component par rapport aux critères distingués auparavant.

Critères	
Initialisation des adaptations	Événements d'adaptation provenant de l'application ou de l'environnement
Localisation de la logique d'adaptation	Dans des contrats d'adaptation chargés par le gestionnaire de configuration
Gestion des politiques d'adaptation	Système ouvert
Positionnement du gestionnaire d'adaptation	Externalisé, mais reposant sur un modèle à composant
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant, utilisation de la réflexion comportementale
Gestion des interruptions	Minimisation des interruptions, les transactions entrantes sur une région en cours de reconfiguration sont bloquées

Tableau 10. Positionnement de K-Component par rapport aux critères

4.1.4. OpenRec

OpenRec [125, 126] est un modèle à composant supportant la reconfiguration dynamique. La particularité d'OpenRec est qu'il permet la création d'applications dynamiques ouvertes. De plus, OpenRec ne définit pas un algorithme de reconfiguration, mais permet de définir d'autres algorithmes et de les substituer à l'exécution.

L'infrastructure d'OpenRec est divisée en trois couches qui sont elles-mêmes implémentées en utilisant OpenRec. Tout d'abord le pilote de reconfiguration (« *change driver* ») reçoit les politiques d'adaptation et détermine quand et comment doit être adaptée l'application. Lorsqu'une adaptation est nécessaire, il notifie le gestionnaire de reconfiguration qui en fonction de l'algorithme de reconfiguration utilisé restructure l'application. L'application est donc supervisée par le pilote et le gestionnaire de reconfiguration. La couche application contient des composants et des connecteurs OpenRec, mais également des plug-ins de supervision. La séparation entre le pilote et le gestionnaire de reconfiguration permet d'utiliser OpenRec pour créer des applications adaptées par un administrateur (sans pilote) ou auto-adaptables (le pilote gère alors une politique d'adaptation).

Critères	
Initialisation des adaptations	Tous types d'évènements (en fonction de la présence ou non du pilote et des plug-ins de supervision déployés sur l'application)
Localisation de la logique d'adaptation	Absente, Implémentée dans le pilote de reconfiguration ou externalisée (et gérée par le pilote de reconfiguration)
Gestion des politiques d'adaptation	Système ouvert
Positionnement du gestionnaire d'adaptation	Externalisé, mais reposant sur le modèle à composant sous-jacent
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant, utilisation de la réflexion Java
Gestion des interruptions	Détection des régions impliquées (blocage des transactions)

Tableau 11. Positionnement d'OpenRec par rapport aux critères

Une particularité d'OpenRec est que le pilote et gestionnaire de reconfiguration sont eux même reconfigurables, ce qui permet de changer la politique d'adaptation ou l'algorithme de reconfiguration à l'exécution. Comme il est possible d'ajouter des sondes sur l'application (plug-ins), il est possible de changer la politique d'adaptation et l'infrastructure de supervision de manière coordonnée. OpenRec définit également différents outils permettant de visualiser les reconfigurations pas à pas ainsi que de connaître le temps de reconfiguration de chacun des composants.

OpenRec définit donc un modèle à composant très puissant permettant de prendre en compte tous les types de dynamisme dès lors que l'application possède les plug-ins adéquats. De plus, le fait que l'infrastructure de reconfiguration d'OpenRec soit elle-même reconfigurable permet la création de systèmes ouverts (Tableau 11).

4.2. Architectures logicielles dynamiques

L'architecture logicielle est l'agencement des composants et des connecteurs formant une application. Typiquement, les composants encapsulent l'information et la logique alors que les connecteurs coordonnent la communication entre les composants [127]. Certaines approches tendent à rajouter également de la logique dans les connecteurs [128, 129].

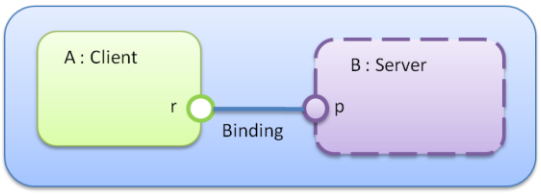
La modification de l'architecture d'un logiciel peut intervenir au développement, au déploiement, mais également à l'exécution [130]. Les architectures logicielles dynamiques sont des architectures logicielles qui peuvent être modifiées à l'exécution. Ces modifications transforment l'application restructurée[131]. Cette section présente et classifie des langages de description d'architecture (ADL) supportant le dynamisme.

4.2.1. Darwin

Darwin est un ADL déclaratif permettant de décrire le dynamisme dans l'architecture. Darwin a d'abord été développé comme un langage de configuration pour Regis, un environnement pour les applications réparties [132]. Cette utilisation a évolué et Darwin est devenu réellement un ADL. Darwin propose une vision textuelle et graphique de l'architecture du système représenté. La sémantique des composants est exprimée en π-calcul [133]. Les systèmes architecturés en utilisant Darwin peuvent être hiérarchiques, c'est-à-dire contenir des composants primitifs et des composants composites (contenant d'autres composants). Les composants primitifs sont généralement définis dans un langage de programmation tel que C++ ou Java [134]. Bien que l'implémentation de ces composants soit effectuée dans un langage de programmation, les interfaces (servant à la communication entre les composants) sont exprimées en Darwin. Les composants composites n'ont pas d'implémentation *réelle* et sont décrits en utilisant la syntaxe de Darwin. Un composant composite

peut contenir des composants primitifs ou d'autres composites. Les connecteurs entre les instances de ces composants sont déclarés dans l'ADL via l'instruction *bind*. Le dynamisme est exprimé dans Darwin de deux différentes manières :

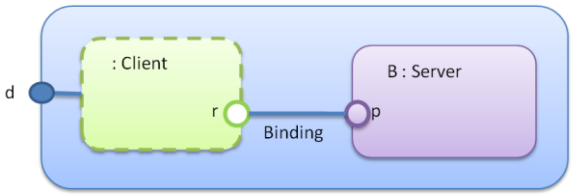
- L'instanciation paresseuse (« *lazzy instantiation* ») : un composant (préalablement sélectionné) est instancié seulement lorsqu'un autre composant a besoin réellement de ce composant (Figure 28).
- L'instanciation dynamique : un composant est arbitrairement choisi et instancié en fonction de ses interfaces fournies (Figure 29).




```
component Server {
    provide p ;
}
component Client {
    require r ;
}
```

```
component ClientServerSystem {
    inst
      A: Client ;
      B: dyn Server ;
    bind
      A.r -- B.p ;
}
```

Figure 28. Exemple d'instanciation paresseuse avec Darwin




```
component Server {
    provide p ;
}
component Client {
    require r ;
}
```

```
component ClientServerSystem {
    provide d <dyn>;
    inst
      B: Server;
    bind
      d -- dyn Client;
      Client.r -- B.p;
}
```

Figure 29. Exemple d'instanciation dynamique avec Darwin

Cependant, Darwin ne supporte pas de façon native les liaisons dynamiques ou les retraits de composant. En effet, l'opérateur *dyn* ne s'applique qu'aux instanciations de composant. Cependant, ces capacités peuvent être ajoutées par un outil externe. Le tableau ci-dessous positionne Darwin en fonction des critères mentionnés dans la section précédente. Les trois derniers critères peuvent être différents en fonction de l'implémentation de Darwin utilisé (Tableau 12).

Critères	
Initialisation des adaptations	Événement interne à l'application
Localisation de la logique d'adaptation	Décrite dans l'architecture de l'application
Gestion des politiques d'adaptation	Système fermé
Positionnement du gestionnaire d'adaptation	Dans le modèle à composant sous-jacent (peut dépendre des implémentations)
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant (peut dépendre des implémentations)
Gestion des interruptions	Détection des régions impliquées (peut dépendre des implémentations)

Tableau 12. Positionnement de Darwin par rapport aux critères

4.2.2. Dynamic Wright

Wright est un ADL permettant de représenter la structure d'une application sous la forme d'un graphe où les composants et les connecteurs sont des nœuds. Dynamic Wright est une extension à Wright intégrant la description de reconfigurations dynamiques dans la description du système [135]. Cet ADL est surtout utilisé pour l'analyse d'une application ou d'une famille d'application (défini par le même style d'architecture) par rapport aux différentes reconfigurations spécifiées. Par exemple, il supporte la vérification automatisée de contraintes architecturales. Un système exprime un style architectural qui est en suite vérifié lors de la simulation du système. Dynamic Wright décrit le comportement du système ainsi que les reconfigurations en utilisant une algèbre de processus proche de CSP. À la fois les composants et les connecteurs sont décrits sous la forme d'un processus.

Style Fault-Tolerant-Client-Server
  Component Client
    Port $p = \overline{request} \rightarrow reply \rightarrow p \sqcap \S$
    Computation $= internalCompute \rightarrow p.request \rightarrow p.reply \rightarrow$ **Computation** $\sqcap \S$

  Component FlakyServer
    Port $p = \S\Box(request \rightarrow \overline{reply} \rightarrow p \sqcap control.down \rightarrow (\S\Box control.up \rightarrow p))$
    Computation $= \S\Box(p.request \rightarrow internalCompute \rightarrow \overline{p.reply} \rightarrow$ **Computation**
      $\sqcap control.down \rightarrow (\S\Box control.up \rightarrow$ **Computation**$))$

  Component SlowServer
    Port $p = \S\Box(control.on \rightarrow \mu Loop.(request \rightarrow \overline{reply} \rightarrow Loop\Box control.off \rightarrow p\Box \S))$
    Computation $= \S\Box control.on \rightarrow \mu Loop.(p.request \rightarrow internalCompute \rightarrow$
      $\overline{p.reply} \rightarrow Loop\Box control.off \rightarrow$ **Computation** $\Box \S)$

  Connector FaultTolerantLink
    Role $c = \overline{request} \rightarrow reply \rightarrow c\Box \S$
    Role $s = (request \rightarrow \overline{reply} \rightarrow s\Box control.ChangeOK \rightarrow s)\Box \S$
    Glue $c.request \rightarrow \overline{s.request} \rightarrow$ **Glue**
      $\Box s.reply \rightarrow \overline{c.reply} \rightarrow$ **Glue**
      $\Box \S$
      $\Box control.ChangeOK \rightarrow$ **Glue**

  Constraints
    $\exists! s \in Component, \forall c \in Component : TypeServer(s) \wedge TypeClient(c) \Rightarrow connected(c,s)$
EndStyle

Configuror DynamicClientServer
  Style Fault-Tolerant-Client-Server
    $new.C : Client$
    $\rightarrow new.Primary : FlakyServer$
    $\rightarrow new.Secondary : SlowServer$
    $\rightarrow new.L : FaultTolerantLink$
    $\rightarrow attach.C.p.to.L.c$
    $\rightarrow attach.Primary.p.to.L.s \rightarrow WaitForDown$
  where
    $WaitForDown = (Primary.control.down \rightarrow Secondary.control.on \rightarrow L.control.changeOk \rightarrow$
      Style Fault-Tolerant-Client-Server
        $detach.Primary.p.from.L.s \rightarrow attach.Secondary.p.to.L.s \rightarrow WaitForUp)$
      $\Box \S$
    $WaitForUp = (Primary.control.up \rightarrow Secondary.control.off \rightarrow L.control.changeOk \rightarrow$
      Style Fault-Tolerant-Client-Server
        $detach.Secondary.p.from.L.s \rightarrow attach.Primary.p.to.L.s \rightarrow WaitForDown)$
      $\Box \S$

Figure 30. Un exemple d'utilisation de Dynamic Wright

Un configurateur (« *configuror* ») décrit comment les types de composants et les types de connecteurs d'une architecture interagissent. Dans la figure ci-dessus, le *style* initial (Style Fault-Tolerant-Client-Server) de l'architecture décrit l'instanciation des composants (Client, FlakyServer et SlowServer) et des connecteurs (FaultTolerantLink) puis les lie entre eux. En plus de ce *style* initial, deux reconfigurations sont décrites dans la Configurator. Les opérateurs *new, del, attach* et *detach* sont utilisés afin de décrire la restructuration de l'architecture. Chaque reconfiguration est attachée à une condition (déclenchant la reconfiguration) telle que *WaitForDown*. Cette condition est un événement déclenché par un des processus constituant l'application. Une différence à noter par rapport à Darwin est que Dynamic Wright supporte le retrait de composant.

Aujourd'hui, il n'existe pas d'implémentation de Dynamic Wright. Les systèmes décrits ne peuvent être que simulés. C'est pourquoi les 3 derniers critères ne peuvent pas être utilisés pour positionner Dynamic Wright (Tableau 13).

Critères	
Initialisation des adaptations	Événement interne spécifié dans un processus
Localisation de la logique d'adaptation	Décrite dans l'architecture de l'application sous forme de reconfiguration
Gestion des politiques d'adaptation	Système fermé

Tableau 13. Positionnement de Dynamic Wright par rapport aux critères

4.2.3. C2ADEL

C2ADEL permet la spécification d'architectures logicielles d'applications développées en utilisant le style architectural C2 [64]. Dans ce style, les composants sont branchés à un connecteur qui gère le transit des messages entre les composants. Les connecteurs peuvent être vus comme des bus à messages asynchrones. Une particularité de C2 est qu'un composant ne connaît que les composants qui sont en aval par rapport à lui, mais ne connait pas les composants de même niveau ou de niveaux sous-jacents (c'est-à-dire en amont). Un composant peut donc utiliser des services fournis par des composants situés au-dessus de lui, mais pas *à côté de lui*, ni en *dessous* (Figure 31).

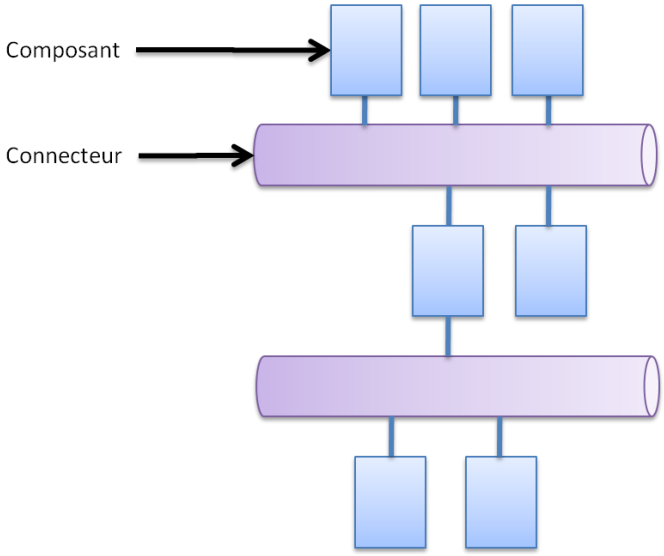


Figure 31. Exemple d'architecture suivant le style C2

Dans C2ADEL, les interfaces de composants sont définies dans un langage de description d'interfaces (IDL). Cette description contient les événements requis et fournis par un composant implémentant cette interface. Ensuite, C2ADL définit un ADL permettant d'architecturer le système en décrivant les composants et leurs liens aux différents connecteurs. Un point important à C2ADL est que le dynamisme est décrit dans un langage de modification d'architecture (AML pour « *Architecture modification language* »). En effet, le système peut être reconfiguré dynamiquement et cette reconfiguration est exprimée dans un langage spécifique. Afin de faciliter la reconfiguration, ArchStudio est un outil graphique permettant le développement d'application C2 en Java et qui permet également leur reconfiguration durant leur exécution [131]. ArchStudio (Figure 32) lie un modèle de l'architecture de l'application (décrit avec C2ADL), avec une implémentation Java de l'application. Avant l'exécution d'une reconfiguration, l'outil vérifie la cohérence de la reconfiguration par rapport à l'architecture actuelle de l'application.

C2ADEL est un langage de description d'architecture très efficace pour décrire des applications adaptables dynamiquement. En effet, grâce à ArchStudio, il est possible de décrire une application puis de la

reconfigurer à l'exécution[9]. Le tableau ci-dessous positionne C2ADEL par rapport aux critères définis précédemment.

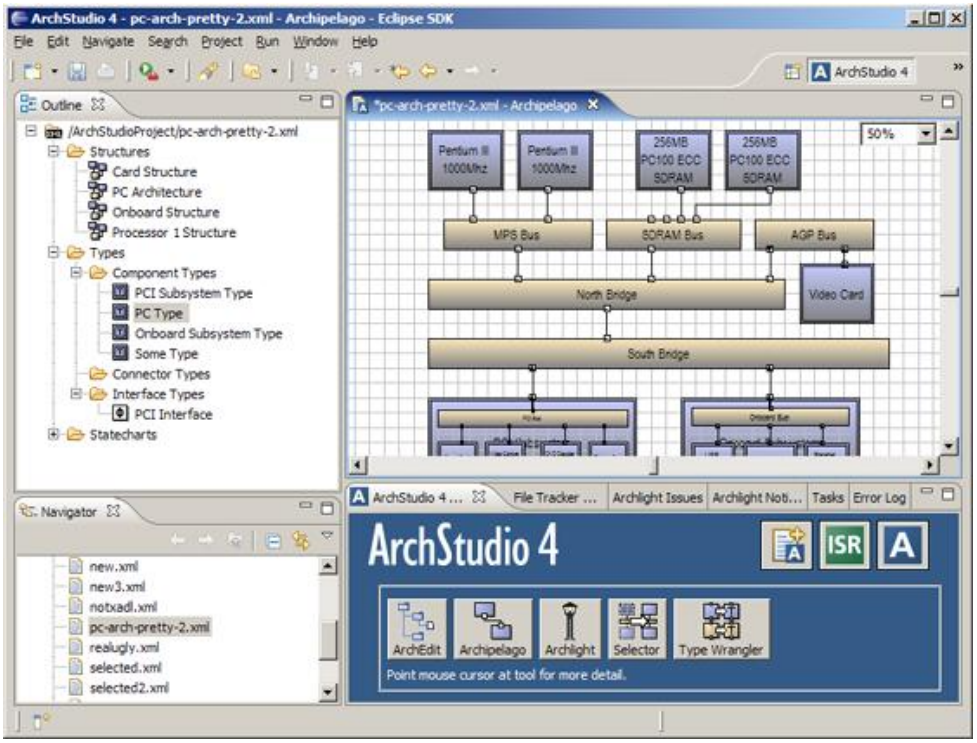


Figure 32. ArchStudio

Critères	
Initialisation des adaptations	Événements externes supervisés par un administrateur (via ArchStudio)
Localisation de la logique d'adaptation	Externalisé dans un script et écrit dans un langage spécifique (AML)
Gestion des politiques d'adaptation	Système ouvert
Positionnement du gestionnaire d'adaptation	Externalisé (gestionnaire d'évolution de l'architecture)
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant, réflexion et chargement de classe de Java
Gestion des interruptions	Minimisée

Tableau 14. Positionnement de C2ADEL par rapport aux critères

5. Conclusion

Construire des applications dynamiques est récemment devenu nécessaire dans de nombreux domaines. Ce besoin provient de trois raisons principales :

- Une application doit pouvoir évoluer durant son exécution : avec la diminution rapide du temps entre deux versions d'un même logiciel, il devient aujourd'hui nécessaire de pouvoir mettre à jour ou reconfigurer des applications déjà déployées.
- Une application doit s'adapter à son environnement d'exécution : les applications mobiles ou ubiquitaires par exemple doivent sans cesse vérifier la disponibilité des ressources (réseau, dispositifs mobiles…) qu'elles utilisent et s'adapter à ces fluctuations.

[9] ArchStudio supporte aujourd'hui d'autres styles architecturaux.

- Finalement, certaines applications requièrent de s'adapter en fonction de leur contexte. Par exemple, une application doit s'adapter en fonction des actions d'un utilisateur afin de toujours fournir un service optimal.

Cependant, la mise en place d'application dynamique reste très complexe. Une application dynamique doit supporter la modification de son architecture durant son exécution. Afin de reconfigurer dynamiquement l'architecture d'une application, trois problèmes majeurs sont à résoudre : quand déclencher les adaptations, comment détecter les régions de l'application impliquées et comment exécuter les reconfigurations en assurant l'intégrité de l'application.

Depuis les années 70, de nombreux travaux se sont intéressés et s'intéressent encore à la mise en place d'applications dynamiques. Les approches proposées sont très hétérogènes et agissent à différents niveaux. Afin de comparer ces approches, ce chapitre définit six critères permettant de positionner les approches permettant la mise en place d'applications dynamiques. Ces critères concernent à la fois comment le dynamisme est exprimé et comment il est géré à l'exécution. Ces critères seront réutilisés dans cette thèse afin de classifier d'autres approches ainsi que la proposition de cette thèse.

Ce chapitre a également utilisé cette caractérisation sur des travaux permettant de mettre en place des applications dynamiques. Le résultat de cette analyse montre que la plupart des solutions proposées se focalisent principalement sur un type de dynamisme. De plus, la plupart des solutions mises en place sont spécifiques à un domaine. Lorsque le dynamisme est géré dans l'architecture, le passage entre cette description architecturale et l'application réelle est délicat à franchir [136]. Ainsi, nous pouvons conclure que bien que nombreuses, la plupart des approches pour mettre en place des applications dynamiques entrainent de nombreuses contraintes et sont donc rarement satisfaisantes.

La suite de cette thèse s'intéressera aux approches à service qui depuis quelques années sont devenues très populaires. En effet, il s'avère que les principes de l'approche à service offrent de nouvelles perspectives pour le support du dynamisme.

Chapitre 4
Architectures à service étendues dynamiques

Dans le chapitre 2, nous avons vu que le développement d'applications innovantes exigeait de nouvelles propriétés difficiles à obtenir. L'intégration et la gestion des infrastructures d'exécution sont les éléments clés pour développer ces applications flexibles. Permettre l'intégration de manière souple et efficace d'applications avec d'autres ressources de l'entreprise ou d'autres entreprises est malheureusement extrêmement complexe à réaliser. La gestion des infrastructures se concentre sur trois objectifs: le développement (et la réutilisation), l'automatisation de l'administration et la virtualisation de l'environnement. Le développement vise à aider les développeurs et les architectures à créer de nouvelles applications en réutilisant un maximum de briques déjà créées. L'automatisation des tâches d'administration vise à réduire la complexité de la gestion des applications ainsi que d'en améliorer la disponibilité, tout en réduisant les coûts. La virtualisation de l'environnement est la capacité à offrir une seule et même vue cohérente de toutes les ressources disponibles.

L'approche à service (Service Oriented Computing ou SOC) propose un nouveau moyen pour développer des applications flexibles. Cette approche utilise des *services* pour construire des applications plus rapidement et composables [137]. Ainsi, « l'orientation service » propose des mécanismes permettant la mise en place d'applications plus facilement intégrables et gérables [138].

L'approche à service n'est pas relative à une technologie en particulier. Cette approche propose un style architectural pouvant être utilisé dans de nombreux domaines comme la télécommunication [139], l'interaction de processus métier [140], voir même dans le contexte de l'informatique résidentielle [23]. Le succès du SOC est vraisemblablement lié au faible couplage qu'il favorise. Le SOC permet de développer des applications modulaires telles que le proposait Parnas, il y a 30 ans [141]. En effet, dans la vision SOC, les fonctionnalités sont fournies sous la forme de « *service* » préalablement spécifié. Le SOC propose un motif d'interaction en trois étapes permettant de découpler les fournisseurs de service et les consommateurs de service. Ainsi un consommateur de service *découvre* un service et peut directement l'utiliser sans dépendre explicitement d'un fournisseur spécifique. Ceci a évidemment un intérêt crucial pour l'intégration de systèmes. Cependant, cette propriété est également intéressante afin de mettre en place des applications dynamiques.

Néanmoins, la mise en place d'applications suivant le paradigme de l'approche à service reste très compliquée. Des problèmes cruciaux tels que la conformité d'un fournisseur (service conformance) ou la responsabilité (service governance) rentrent en jeux et restent aujourd'hui non traités [142]. De plus, il existe encore de nombreux problèmes non résolus dans l'ingénierie des applications à service.

Ce chapitre détaillera les principes des approches à service ainsi que leurs apports pour créer les applications dynamiques. Nous nous intéresserons plus particulièrement à la partie exécution d'une application à service, point critique pour créer des applications dynamiques. De plus, ce nouveau paradigme a ouvert la voie à un nouveau type de modèles à composant reposant sur les principes du SOC. Ce nouveau type de modèles à composant, appelé modèle à composant à service sera également présenté et étudié dans ce chapitre.

1. Approche à service et architectures à service

L'approche à service est un nouveau paradigme qui utilise les *services* comme la brique de base pour créer des applications [143]. Le but de ce style architectural est d'utiliser des entités faiblement couplées afin d'améliorer la *composabilité* des applications. Cette section décrira en détail les principes de ce style architectural ainsi que son apport pour la création d'applications dynamiques. Aujourd'hui de nombreuses technologies permettent de mettre en place des applications à service. Cette section détaillera et comparera quelques-unes de ces technologies.

1.1. Principes de l'approche à service

L'approche à service est un style architectural qui utilise les *services* comme entités de base. Le but principal de l'approche à service est de permettre la définition et l'utilisation de brique logicielle peu dépendante [142]. En réduisant ces dépendances, chaque élément peut évoluer séparément. Ainsi les applications décrites en termes de *services* exhibent une plus grande flexibilité que les applications monolithiques. Ce faible couplage entre les composants de l'application provient du motif d'interaction proposé par l'approche à service. En effet, ce style architectural est basé sur 3 acteurs :

- Les ***fournisseurs de services*** qui offrent des services
- Les ***consommateurs de services*** qui requièrent et utilisent des services offerts par des fournisseurs
- Un ***courtier de service*** permettant aux fournisseurs de publier leurs services et aux consommateurs de découvrir et de sélectionner les services qu'ils veulent utiliser.

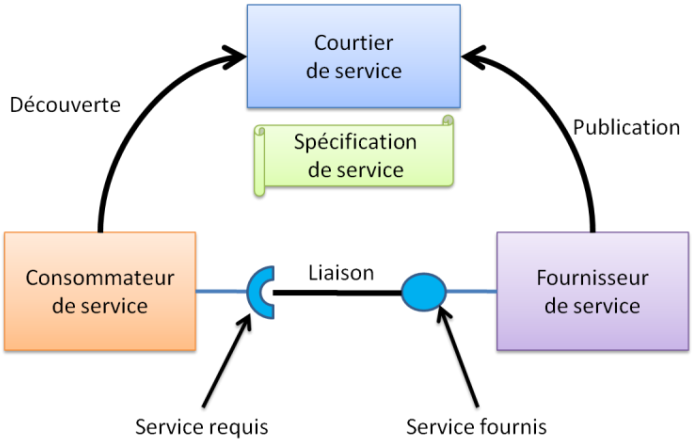


Figure 33. Acteurs et interactions dans l'architecture à service

Un autre élément central de l'approche à service est la ***spécification de service*** contenant la description de la fonctionnalité offerte par le service. Celle-ci peut être purement syntaxique ou contenir des éléments concernant le comportement ou la sémantique du service. La spécification de service est la seule information partagée par un fournisseur et un consommateur. Les fournisseurs de service implémentent une (ou plusieurs) spécification(s) de service. Les consommateurs de service savent comment interagir avec des fournisseurs implémentant la spécification de service qu'ils requièrent. Ainsi, l'approche à service propose trois primitives d'interaction (Figure 33):

- La ***publication*** : les fournisseurs de service s'enregistrent auprès du courtier.
- La ***découverte*** : les consommateurs de service interrogent le courtier afin de trouver les fournisseurs de services qu'ils requièrent.
- La ***liaison*** : une fois découvert, le consommateur de service peut se lier à un fournisseur de service et utiliser le service qu'il fournit.

Un point important est que ce principe d'interaction peut avoir lieu à n'importe quel moment dans le cycle de vie du logiciel; c'est-à-dire qu'il peut avoir lieu durant le développement afin de trouver un service adéquat (approche généralement utilisée avec les services web), mais également à l'exécution. Nous nous intéresserons plus attentivement à ce cas dans la suite de ce chapitre.

Grâce à ce principe d'interaction, les applications à service ont des caractéristiques intéressantes telles que :

- Le ***faible couplage*** : comme la seule information partagée entre les fournisseurs et les consommateurs est la spécification de service, le couplage entre ces deux entités est faible.
- La ***liaison tardive*** : la liaison entre les fournisseurs et les consommateurs a lieu seulement lorsqu'un fournisseur est trouvé et lorsque le consommateur le demande.
- Le ***masquage de l'hétérogénéité*** : grâce au faible couplage, un consommateur n'a pas à connaitre les détails d'implémentation du fournisseur de service, ni à sa localisation précise.

Afin de concevoir des applications à service plus complexe, il est nécessaire de composer ces services ensemble afin de fournir des services de plus haut niveau, c'est-à-dire qu'un fournisseur peut requérir autres services afin de fournir le sien. Il existe aujourd'hui deux tendances dans la composition de services. La première est la ***composition comportementale.*** Ce style de composition tire ses sources des procédés logiciels. Une composition comportementale de services combine des services en décrivant le procédé logique permettant de fournir un service de plus niveau. La composition de services web utilise majoritairement ce style de composition et plus exactement le langage « Business Process Execution Language for Web Services »[144]. Cette composition est ensuite exécutée par un moteur d'orchestration qui gérera l'exécution des activités décrites dans la composition. Le deuxième style de composition est la ***composition structurelle***. Ce style de composition se rapproche des langages de description d'architecture. Une composition structurelle de service décrit l'architecture d'un composant qui fournit un service. Cette architecture montre les *composants* participants et leurs interconnexions. Aujourd'hui, « Service Component Architecture » est l'un des seuls modèles de composition structurelle existants [145]. Ces deux styles de compositions peuvent être combinés : une composition comportementale peut être l'implémentation d'un service utilisé dans une composition structurelle.

Un concept important du SOC est l'***architecture à service*** (Service Oriented Architecture ou SOA)[137]. Un SOA est un ensemble de technologies permettant de mettre en place des applications suivant le paradigme de l'approche à service. Généralement, un SOA est un intergiciel permettant aux applications de fournir, publier, découvrir et utiliser des services. Un SOA implante les primitives de l'approche à service en utilisant diverses technologies. Par exemple, les services web sont un SOA, alors qu'OSGi en est un autre. Ces deux SOA utilisent des technologies très différentes. Les choix des technologies choisis pour mettre en place un SOA dépendent du domaine métier et des objectifs poursuivis.

Papazoglou a défini dans [142], la notion de SOA étendu. En effet, un SOA pur n'adresse pas les problématiques de composition et de gestion nécessaire à la création d'applications flexibles. Un SOA étendu prend en compte ces préoccupations (Figure 34). Tout d'abord un SOA étendu se base sur un SOA *basique*. Ensuite, un SOA étendu propose des fonctionnalités pour le support de la composition. Parmi ces fonctionnalités, il doit être possible de coordonner des services, de gérer les compositions ainsi que d'assurer la conformité de la composition par rapport au service rendu. Une fois réalisé, un ***service composite*** peut ensuite être utilisé comme un service « normal ». Au dessus de la couche de composition, un SOA étendu fourni des fonctionnalités permettant la gestion des applications à service. Cette couche fournit les mécanismes de management des applications tel que le déploiement, la supervision, l'évolution… Deux fonctionnalités sont particulièrement cruciales : la capacité à vérifier la conformité des compositions de services, ainsi que la capacité à vérifier différents indices de qualité devant être atteints. De manière transversale, un SOA étendu

doit permettre la vérification et la gestion des propriétés non fonctionnelles telles que la sécurité ainsi que de diverses formes de qualité de service.

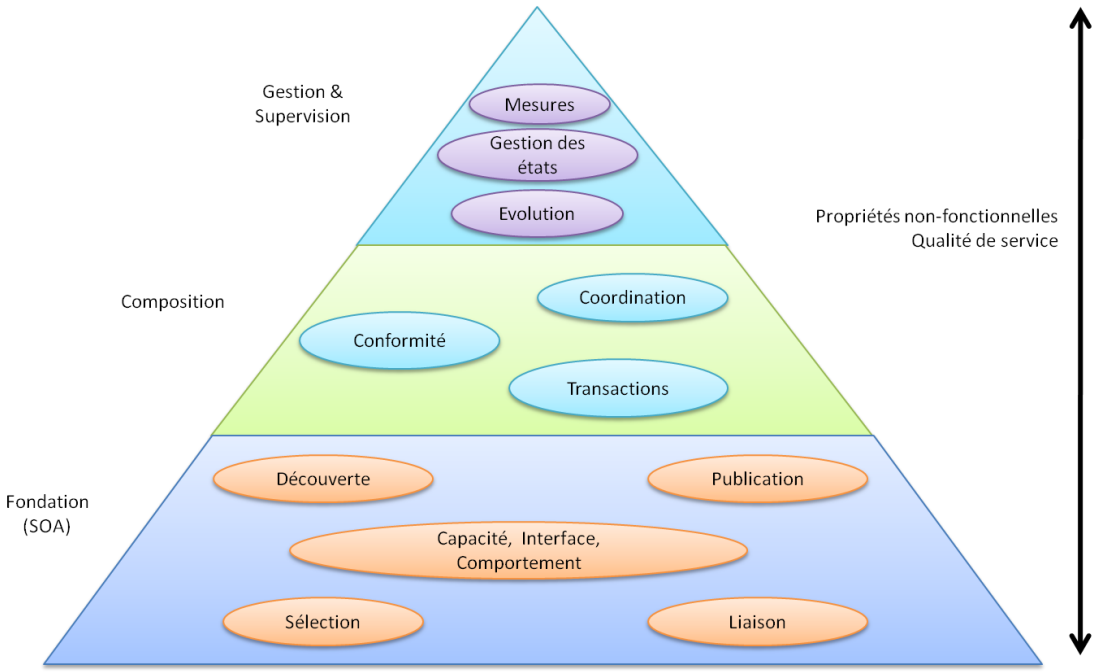


Figure 34. Fonctionnalité d'un SOA étendu selon Papazouglou

Aujourd'hui, il existe de nombreux SOA ainsi que de nombreux SOA étendus. Il devient critique de pouvoir les faire communiquer. En effet, bien qu'à l'intérieur d'un SOA, les entités peuvent communiquer dans le langage supporté par ce SOA, des entités de deux SOA différents ne peuvent souvent pas communiquer.

1.2. Approche à service et dynamisme

Le motif d'interaction proposé par l'approche à service peut avoir lieu à l'exécution. Ainsi, les fournisseurs de services s'enregistrent auprès du courtier durant leur exécution, alors que les consommateurs découvrent les fournisseurs durant leur exécution. Décliner ce motif d'interaction à l'exécution permet de construire des applications supportant le dynamisme et plus particulièrement le dynamisme issu de l'environnement et du contexte.

Cependant, l'approche à service dynamique requiert deux nouvelles primitives :

- Le ***retrait de service*** : lorsqu'un service n'est plus disponible, le fournisseur le retire du courtier
- La ***notification*** : lorsqu'un fournisseur de service arrive et publie un service, les consommateurs sont notifiés de cette arrivée. À l'inverse, lorsqu'un fournisseur de service disparaît, les consommateurs doivent également être notifiés afin de pouvoir réagir face à ce départ.

Ainsi, grâce à ces deux primitives, il est possible pour les consommateurs de services de gérer le départ d'un fournisseur (Figure 35) ainsi que l'arrivée d'un nouveau fournisseur implémentant une spécification de service compatible (Figure 36).

En fonction des capacités du courtier, un consommateur peut également sélectionner un fournisseur parmi plusieurs disponibles ainsi que décider de choisir un autre fournisseur afin de s'adapter à un changement de contexte.

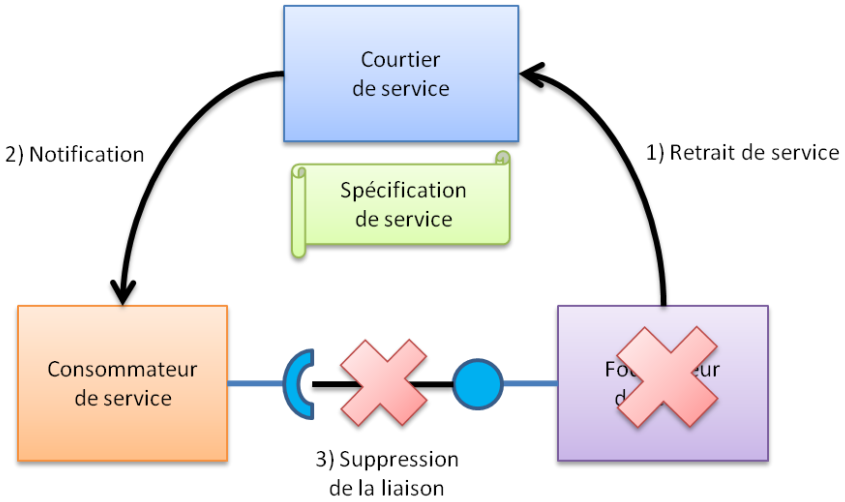


Figure 35. Exemple de départ de service

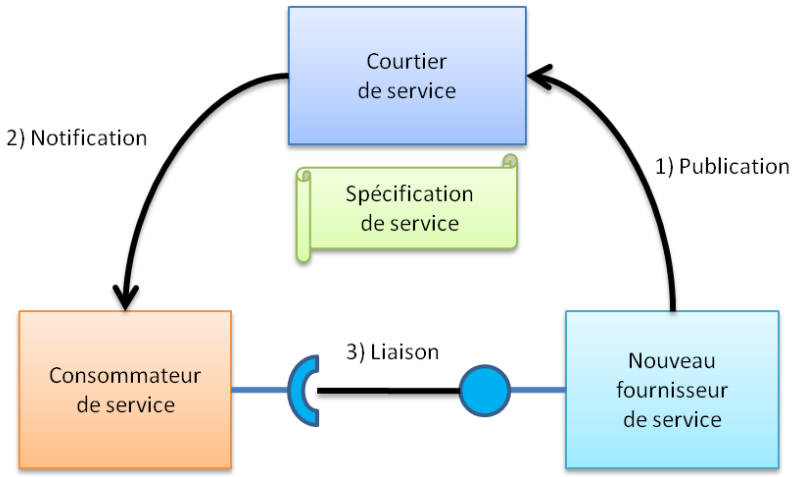


Figure 36. Arrivée d'un nouveau fournisseur fournissant le même service

Tout comme Papazoglou a défini le SOA étendu, il est possible de définir le SOA dynamique étendu. Un SOA dynamique étendu (Figure 37) reprend les concepts du SOA étendu, mais se focalise particulièrement sur la gestion du dynamisme. Tout d'abord, le SOA sur lequel repose un SOA dynamique étendu doit implanter les primitives nécessaires pour le dynamisme (c'est-à-dire la publication et le retrait de service dynamiques ainsi que la notification). Ensuite, les mécanismes de compositions proposés par un SOA étendu dynamique doivent être capables de gérer l'arrivée et le départ des services utilisés, et donc assurer la conformité des services fournis en fonction de ces événements. Plus particulièrement, la composition dans un SOA étendu dynamique doit être capable de gérer la structure de la composition afin de déterminer qu'elles sont les services manquants ou à remplacer. Finalement, la partie gestion et supervision doit être capable de vérifier en permanence la cohérence du système sous-jacent en fonction des critères de l'application. De manière transversale, un SOA étendu dynamique doit permettre la vérification et la gestion des propriétés non-fonctionnelles telles que la sécurité ainsi que la qualité de service. Cependant, effectuer tous ces traitements tout en gérant le dynamisme reste aujourd'hui extrêmement complexe. Peu d'intergiciels implémentent les trois couches du SOA étendu dynamique.

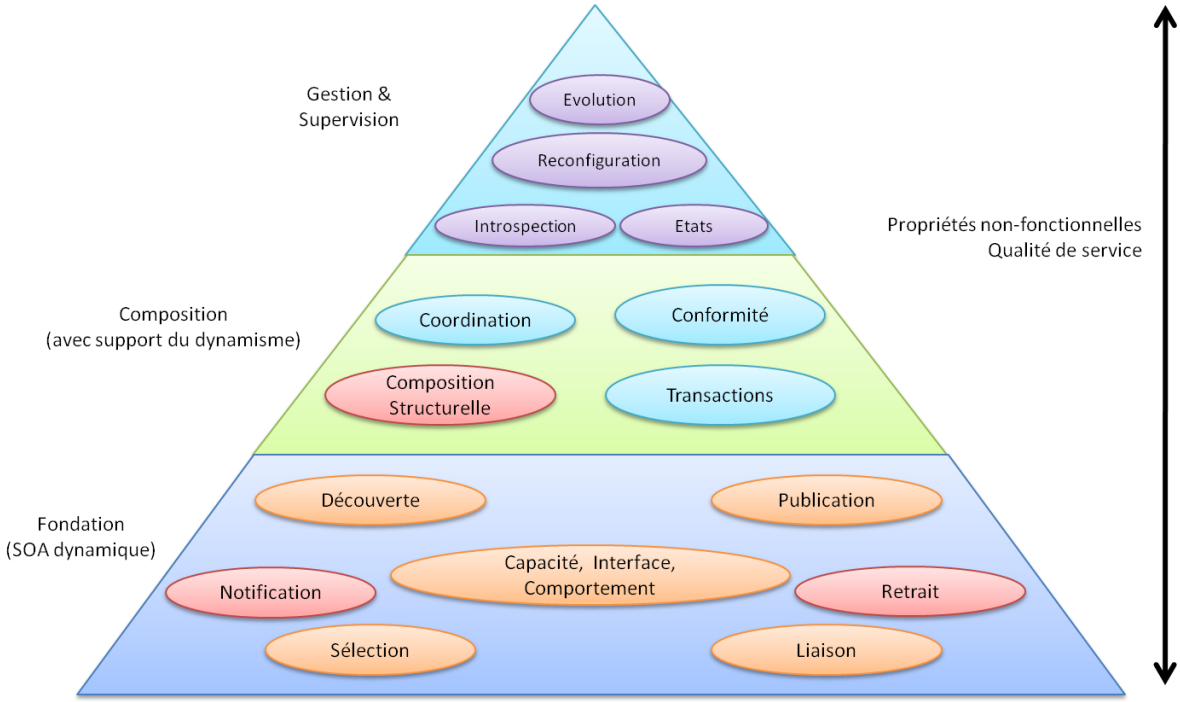


Figure 37. Fonctionnalités d'un SOA étendu dynamique

La projection du motif d'interaction proposée par l'approche à service à l'exécution fournit les bases afin de créer des applications dynamiques. En effet, grâce aux primitives de l'architecture à service dynamique, il est possible de créer des applications supportant le dynamisme issu de l'environnement et du contexte. L'approche à service dynamique ouvre donc de nouvelles voies afin de créer des applications dynamiques. De plus, les SOA dynamiques étendus fournissent l'infrastructure pour exécuter et gérer ces applications.

2. SOA supportant l'approche à service dynamique

La section précédente a montré que l'approche à service semblait être intéressante afin de créer des applications dynamiques lorsque le motif d'interaction *publication-découverte-liaison* se déroulait à l'exécution. Aujourd'hui de nombreux SOA et SOA étendus supportent ce type d'interaction et permettent donc de mettre en place des applications dynamiques à service. Cette section a pour objectif de caractériser et de comparer ces différentes technologies. Tout d'abord, les critères retenus seront décrits, ensuite des technologies supportant l'approche à service dynamique seront comparées en fonction de ces critères.

2.1. Caractérisation des SOA supportant l'approche à service dynamique

La section précédente a introduit la notion de SOA étendu dynamique. Afin de caractériser les SOA supportant le dynamisme, nous avons sélectionné les critères suivants relatifs aux éléments clés des SOA étendus et des SOA étendus dynamiques (Tableau 15) :

- Le support et le contenu de la spécification de service
- Le type de courtier de service
- L'implantation des primitives de l'approche à service dynamique
- Les infrastructures proposées pour la composition de service.

Tout d'abord, nous nous intéresserons aux spécifications de services. Le premier sous-critère concerne le langage utilisé pour écrire ces spécifications. En effet, celles-ci peuvent être décrites en XML ou dans un autre langage de description. Le second sous-critère s'intéresse au contenu de ces spécifications. Les

spécifications de service peuvent être purement syntaxiques ou contenir des informations comportementales ou sémantiques [111].

Critères	Valeurs
Langage utilisé pour les spécifications de service	Java, XML, langages spécifiques….
Information contenue	Syntaxique, Comportementale, Sémantique
***Forme* du courtier**	Existe réellement et peut être interrogé par n'importe quel consommateur, Chaque consommateur doit maintenir la liste des services disponibles
Localisation du courtier	Distant/Centralisé, Unique/Multiple …
Implantation de la publication et du retrait de service	Manuel / Automatique, Appel de procédure (distante ou non), Envoi de message…
Type de découvertes supportées	Active, Passive, Les deux
Langage de requête et filtrage	Non-supporté, LDAP, X-Query …
Implantation de la notification	Bail, Évènements (locaux /distribués) …
Protocole de liaison	Java, Java RMI, SOAP, IIOP …
Type de composition supporté	Non supportée, Comportementale et/ou Structurelle
Infrastructure proposée pour exécuter les compositions	Aucune, Moteur d'orchestration, Annuaire hiérarchique

Tableau 15. Critères caractérisant les architectures à service dynamiques

Une des principales différences entre toutes les technologies permettant l'approche à service concerne les courtiers de service. Une des caractéristiques principales des architectures à service réside dans la généralisation du courtier. Cependant, en fonction des technologies ce courtier peut avoir des formes très différentes. Un grand nombre de technologies et de protocoles peuvent être utilisés comme courtier tel que DNS [146], DNS/SD [147], ebXML [148]. En effet celui-ci peut être unique ou multiple, exister physiquement ou être maintenu par tous les consommateurs, être distant ou centralisé...

Le troisième critère s'intéresse à l'implantation des différentes primitives de l'approche à service. En effet, toutes ces technologies n'implémentent pas ces primitives de la même manière. Ceci a un impact sur la gestion du dynamisme. Les protocoles de découverte de service utilisés peuvent supporter la découverte active (le consommateur de service interroge le courtier directement) ou/et la découverte passive (le courtier notifie le consommateur de service). Les courtiers peuvent également proposer des langages requêtes avancés afin de permettre à un consommateur de filtrer les fournisseurs de service. La publication et le retrait de service peuvent également être supportés différemment. Enfin, l'invocation de service utilise des protocoles très divers en fonction des technologies.

Enfin, le dernier critère s'intéresse à la composition de service, c'est-à-dire aux moyens mis en place afin de fournir des fonctionnalités de plus haut niveau à partir de service existants. Nous avons remarqué que la plupart des SOA supportant le dynamisme n'étaient pas *étendus*. Les deux sous-critères proposés permettent de voir si le SOA étudié a les caractéristiques requises pour fournir la couche de composition des SOA étendus. Le premier décrit le type de composition supporté lorsque le SOA supporte la création de services composites. Le second sous-critère décrit les infrastructures mises en place pour exécuter et vérifier ces compositions (coordination, conformité, supervision). En effet, la plupart des technologies ne proposent pas d'infrastructure permettant d'exécuter directement ces compositions. Dans ce cas, les développeurs doivent coder directement l'exécution de leur composition.

La suite de cette section positionnera des technologies suivant l'approche à service dynamique en fonction des critères présentés.

2.2. CORBA

CORBA, acronyme de « Common Object Request Broker Architecture », est une architecture logicielle, pour le développement de composants et d'Object Request Broker (ORB) [149, 150]. Ces composants, qui sont assemblés afin de construire des applications complètes, peuvent être écrits dans des langages de programmation distincts, être exécutés dans des processus séparés, et être déployés sur des machines distantes.

CORBA est une norme créée en 1992, initiée par différents constructeurs et éditeurs dont Sun, Oracle, et IBM regroupés au sein de l'Object Management Group. La spécification CORBA décrit un langage de description d'interface (Interface description language ou IDL) dans sa première version. Dans sa version 2 (1996) est introduit un standard de communication (IIOP). C'est avec la version 3, que CORBA a permis la mise en place d'applications dynamiques suivant les principes des approches à service dynamique. En effet, la version 3 (2002) de CORBA spécifie 16 types de services (nommage et annuaire des objets, cycle de vie, notification d'événements, transaction, relations et parallélisme entre objets, stockage, archivage, sécurité, authentification et administration des objets, gestion des licences et versions…). Parmi ces services, on remarquera un annuaire. Ce service est la clé dans la mise en place d'application dynamique à base de service avec CORBA.

```
service PrinterServiceDescription { // description
    interface PrinterService ;
    mandatory property string building ;
    property short floor ;
    mandatory property string type ;
    mandatory property string language ;
    property string name ;
} ;


interface PrinterService { // interface de service
    typedef unsigned long JobID ;
    JobID print (in string data) ;
} ;
```

Figure 38. Exemple de spécification de service en CORBA

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 (Figure 38). 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). La spécification de service contient des informations syntaxiques (interface de service). Cependant, à l'aide des propriétés que doivent publier les fournisseurs, il est possible de caractériser chaque fournisseur de service. De plus, CORBA supporte l'héritage de service. Une spécification de service peut hériter d'une autre spécification.

L'annuaire de service de CORBA a la particularité de pouvoir interagir avec d'autres annuaires créant ainsi des *fédérations* d'annuaires. Pour le consommateur de service, cette fédération apparaît comme un seul annuaire *virtuel*. Lorsqu'une requête est effectuée, tous les annuaires de la fédération sont interrogés afin d'augmenter le nombre de réponses.

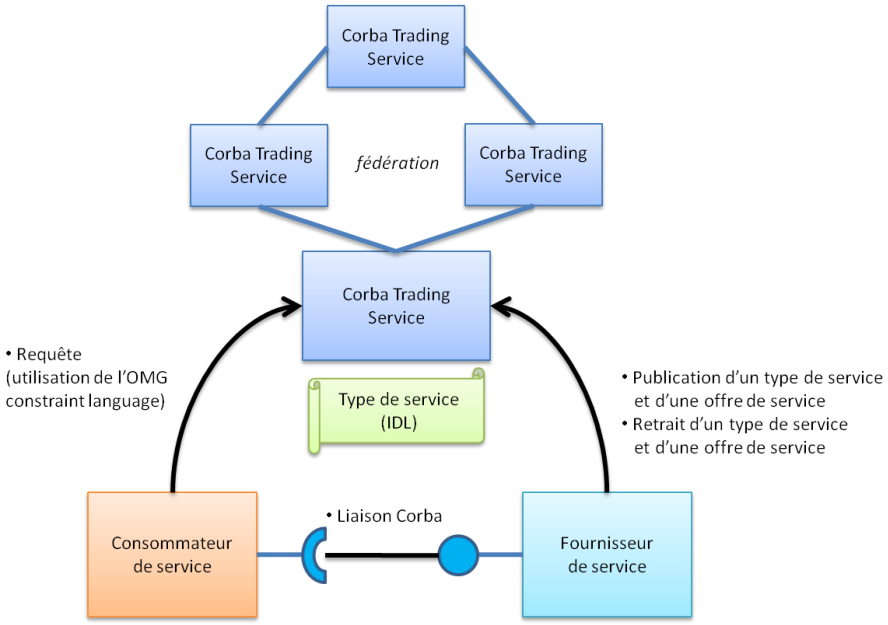


Figure 39. L'architecture SOA de CORBA

La publication d'un service en CORBA se fait en deux étapes. Tout d'abord, la spécification de service (appelé *type de service*) doit être ajoutée dans l'annuaire de spécification de service. Pour cela, il est nécessaire d'interagir avec le service d'annuaire de CORBA (accessible via l'ORB), afin de récupérer une référence sur l'annuaire de spécification de service et de publier le type de service (Figure 39). Un fournisseur ne peut publier un service si son type n'est pas préalablement publié. Lorsque le type de service est publié, les fournisseurs doivent publier leurs *offres de service.* Ils communiquent avec le service d'annuaire de CORBA qui leur permettra de publier leur offre de service. Les fournisseurs doivent publier des propriétés si le type de service qu'ils fournissent le spécifie. Les fournisseurs peuvent également changer ces valeurs à l'exécution.

De la même manière le retrait de service se fait en deux étapes. D'abord, les fournisseurs peuvent retirer leur service en utilisant la fonction « *withdraw* » de l'annuaire de service. Ensuite, l'administrateur peut décider de supprimer le type de service en appelant la méthode « *remove_type* » sur l'annuaire de spécifications de service.

CORBA supporte la recherche active. Un consommateur de service peut rechercher un service en se connectant à l'annuaire de service. La requête spécifie le type de service recherché (c'est-à-dire la spécification), des contraintes sur les propriétés, l'ordre de classement des fournisseurs, la politique de recherche (principalement la portée de la recherche) ainsi que le nombre maximum d'offres retournées. Les contraintes sur les propriétés sont décrites en utilisant l'OMG Constraint Language. En réponse à la requête, le consommateur reçoit une liste d'offres. À partir de cette offre, le consommateur peut accéder au service via l'ORB. Ensuite, le consommateur de service peut utiliser le service en invoquant les méthodes décrites dans l'interface du service. L'interaction utilise généralement IIOP, mais peut également utiliser Java RMI. Une fois l'utilisation terminée, le consommateur doit explicitement relâcher le service via la méthode « *remove* ».

Bien que les fournisseurs de services puissent publier et retirer dynamiquement leurs offres de services, CORBA ne permet pas aux consommateurs d'être notifiés de ces départs et de ces arrivées. Un consommateur doit vérifier la présence du service dès qu'il l'utilise et libérer ce service dès que celui-ci ne lui est plus utile.

La composition de service n'est pas supportée par défaut par CORBA. En effet, afin de fournir un service à partir d'autres services, les développeurs doivent coder entièrement cette composition et donc gérer le dynamisme manuellement. Bien que des travaux permettant la composition comportementale [151] ou

structurelle [152] au dessus de CORBA ont déjà été réalisés, aucun ne gère l'aspect potentiellement dynamique de ces compositions.

En conclusion, CORBA fournit un environnement permettant le développement d'application répartie suivant le paradigme de l'approche à service. Cependant, le fait que CORBA ne supporte pas la recherche passive empêche le développement d'applications réellement dynamique. CORBA possède néanmoins des caractéristiques intéressantes comme la fédération d'annuaire, un mécanisme à deux annuaires, un langage de contrainte… Le tableau ci-dessous positionne CORBA par rapport aux critères choisis.

Critères	Valeurs
Langage utilisé pour les spécifications de service	Langage spécifique (IDL)
Information contenue	Syntaxique (interface) et des propriétés
***Forme* du courtier**	CORBA Trading Service
Localisation du courtier	Généralement distants, Plusieurs courtiers peuvent être réunis sous la forme d'une *fédération* CORBA a la particularité d'avoir 2 types d'annuaire : un pour les types de service, un autre pour les offres de service
Implantation de la publication et du retrait de service	Publication et retrait manuels utilisant des appels de méthode sur le CORBA Trading Service
Type de découvertes supportées	Active seulement
Langage de requête et filtrage	OMG Constraint Language
Implantation de la notification	Pas de notification
Protocole de liaison	Principalement IIOP et Java RMI
Type de composition supporté	Non supportée de base
Infrastructure proposée pour exécuter les compositions	Pas d'infrastructure proposée

Tableau 16. Positionnement de CORBA comme SOA dynamique

2.3. Jini

Jini est une spécification, proposée par Sun en 1999[10], ciblant les réseaux locaux ad hoc dynamiques [153, 154]. Ces réseaux sont formés de dispositifs communicants pouvant apparaître et disparaître à tout moment. Le but de Jini est de rendre ce type de réseaux flexible et de permettre le développement d'applications au dessus de ces réseaux. Jini définit :

- Un ensemble de composants qui fournissent une infrastructure afin d'administrer ces réseaux
- Un modèle de programmation permettant la mise en place d'application au dessus de ces réseaux.
- Des services utilisables par l'ensemble des acteurs du réseau (à la fois humain, matériel ou logiciel)

Dès la première version de la spécification, Jini promeut l'approche à service dynamique et fournit un environnement complet basé sur Java et sur Java RMI afin de développer des applications suivant ce paradigme. Jini définit le concept de *fédérations.* Une fédération est un groupe de ressources (matérielles ou logicielles) reliées dans un même réseau ad hoc dynamique. Chaque entité d'une fédération peut fournir ou requérir des *services*. Un service étant une fonctionnalité utilisable à distance telle qu'un service d'impression (offert par une imprimante) ou un service de traduction (offert par un logiciel de traduction). Les services Jini sont décris grâce à des interfaces Java donc contiennent simplement des informations syntaxiques (Figure 40).

[10] Jini est aujourd'hui implémenté au sein du consortium Apache dans le projet Apache River (http://incubator.apache.org/river/RIVER/index.html)

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

Figure 40. Exemple de spécification de service avec Jini

Afin de trouver les services requis, Jini fournit un service de recherche (*lookup service*) qui joue le rôle du courtier. Une particularité de Jini est que cet annuaire est également un service. De plus, un service de recherche peut contenir d'autres services de recherche. Ce système se rapproche des *fédérations* de CORBA.

Afin de s'enregistrer dans un service de recherche, un fournisseur doit d'abord découvrir l'annuaire avant de pouvoir l'utiliser. Pour cela, un protocole de *découverte* et de *jonction* est mise en place (*discovery and join protocol)*. Ensuite un fournisseur publie son service sous la forme d'un *service item*. Un *service item* contient principalement l'*objet de service* (l'objet servant le service) et des propriétés (appelé *entrées* ou *attributs de service*) sur le fournisseur. Cet ensemble n'étant pas contraint, les fournisseurs peuvent définir leurs propriétés spécifiques. Dans ce *service item,* le type de service (l'interface) n'est pas spécifié. Le service de recherche découvrira les services fournis par réflexion. Une fois le *service item* crée le fournisseur peut publier son service. Un élément important de Jini est la notion de *bail*. Lors de la publication, un fournisseur indique la durée du bail. Lorsque ce bail expire, le fournisseur doit renouveler le bail, sinon l'offre de service est supprimée du service de recherche. Lors de la publication, le fournisseur reçoit un objet (*service registration)* lui permettant ensuite de *renouveler/annuler* le bail ou de modifier les propriétés. Annuler le bail signifie retirer le service de l'annuaire.

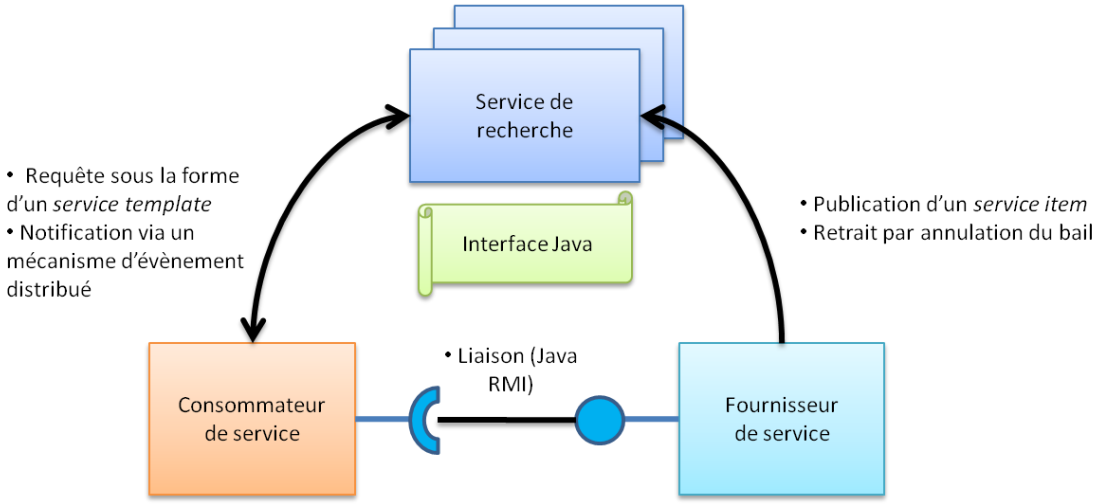


Figure 41. L'architecture SOA de Jini

Les consommateurs de services doivent également rechercher le service de recherche. Ceci est fait en utilisant le même protocole que pour les fournisseurs. Une fois localisé, les consommateurs peuvent rechercher des services en spécifiant le type de service recherché (en spécifiant l'interface de service) ainsi que les valeurs exactes des attributs (Jini ne propose pas de langage de contrainte). La requête est émise sous la forme d'un *gabarit de service* (*service template)*. En réponse à cette requête, le consommateur reçoit la liste des services disponibles. Chaque offre de service est fournie sous la forme d'un *service item*. À partir de cette réponse, le consommateur peut directement utiliser un fournisseur. Les interactions utiliseront principalement Java RMI, mais peuvent également utiliser d'autres protocoles.

Jini supporte également la découverte passive. Un consommateur peut être notifié des arrivées ou des départs des fournisseurs de service. Pour cela un consommateur s'enregistre auprès du service de recherche. Il fournit le gabarit de service compatible avec le gabarit. Il peut ainsi être informé des arrivées et des départs des fournisseurs de service l'intéressant. Là encore, cette demande de notification est régie par un système de baux. Le consommateur doit renouveler son bail afin de continuer à être notifié.

Tout comme CORBA, Jini ne propose pas de modèle de composition. Jini laisse à la charge du développeur comment composer les services et comment gérer le dynamisme des compositions. Huang a proposé dans [155], un langage de composition afin de pouvoir exécuter des compositions comportementales au dessus de Jini. Cependant, une fois la composition écrite, celle-ci est traduite en Java. De plus, le dynamisme n'est pas géré.

En conclusion, Jini propose toute l'infrastructure afin de créer des applications dynamiques suivant l'approche à service. Le dynamisme est totalement gérable, car Jini supporte à la fois la recherche active et passive. De plus, le principe de bail permet de « garantir » la disponibilité d'un fournisseur. Cependant, bien que novateur à l'époque, Jini n'a pas eu le succès escompté. Ceci est probablement dû à des défauts dans l'implémentation de référence. Le tableau ci-dessous positionne Jini par rapport aux critères choisis.

Critères	Valeurs
Langage utilisé pour les spécifications de service	Java (interface Java)
Information contenue	Syntaxique (interface)
***Forme* du courtier**	Lookup Service fournit sous la forme d'un service Jini
Localisation du courtier	Généralement distant, Un courtier peut contenir d'autres services de recherche. Plusieurs courtiers peuvent également être disponibles sur le réseau
Implantation de la publication et du retrait de service	Publication et retrait manuels via des appels de méthode sur le Lookup Service
Type de découvertes supportées	Active et Passive
Langage de requête et filtrage	Pas de langage de contrainte, tests sur l'égalité des attributs
Implantation de la notification	Notification via un appel de méthode
Protocole de liaison	Principalement Java RMI
Type de composition supporté	Non supportée de base
Infrastructure proposée pour exécuter les compositions	Pas d'infrastructure proposée

Tableau 17. Positionnement de Jini comme SOA dynamique

2.4. OSGi™

OSGi est une spécification de plate-forme à service[42]. Cette spécification est définie au sein d'un consortium appelé OSGi Alliance[11] réunissant des acteurs tel qu'IBM, Nokia, Motorola, Oracle … À l'origine, cette spécification ciblait les passerelles résidentielles. Cependant, OSGi est désormais utilisé dans de nombreux domaines comme :

- les téléphones portables tels que Sprint Titan [156]
- les serveurs d'applications tels que OW2 Jonas [157] ou Sun Glassfish
- les applications à plug-ins telles que l'IDE Eclipse [158]

La spécification OSGi définit une plate-forme Java de services non distribuée ainsi que des moyens permettant de réaliser le déploiement des fournisseurs et des consommateurs de services. OSGi™ fournit à la fois :

- Des mécanismes de déploiement : OSGi définit des unités de déploiement appelées *bundles* qui sont interconnectées les uns des autres par la plate-forme. Ces unités sont déployées et administrées à l'exécution. Il est donc possible d'installer, de démarrer, d'arrêter de mettre à jour ou de supprimer des *bundles* à l'exécution sans interrompre la plate-forme. Un *bundle* peut contenir du code Java et des ressources. Chaque *bundle* décrit les *paquetages Java* requis

[11] http://www.osgi.org/Main/HomePage

et fournis ainsi que les numéros de version. La plate-forme est également capable d'exécuter plusieurs versions différentes d'un même *paquetage Java*.

- Des mécanismes pour délivrer et consommer des services : la plate-forme implante les primitives nécessaires afin de développer des applications suivant l'approche à service dynamique. Plus particulièrement, chaque *bundle* peut fournir des services et également consommer des services fournis par d'autres *bundles*. Comme OSGi permet la gestion dynamique des unités de déploiement, les fournisseurs de service (contenus dans les *bundles*) peuvent apparaître et disparaître à l'exécution.

De plus, la spécification OSGi définit des services de sécurité, d'évènement, de journalisation, d'administration… qui permet la mise en place d'applications sophistiquées.

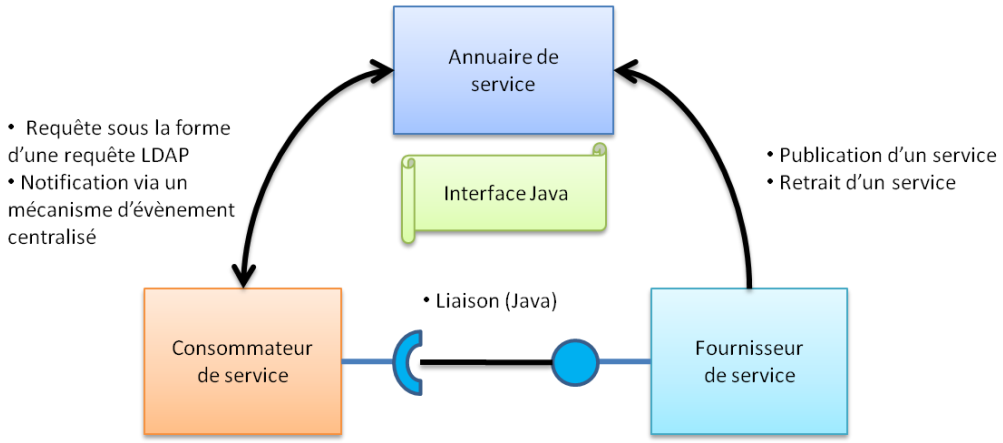


Figure 42. Architecture SOA d'OSGi™

Afin de supporter l'approche à service, la plate-forme OSGi™ définit un *annuaire de service* (service registry). Un service OSGi est spécifié sous la forme d'une interface Java. Les fournisseurs de service ont accès à l'annuaire via un objet spécial (le *bundle context*) injecté à l'activation du *bundle*. Le fournisseur doit spécifier le ou les services fournis (donc l'ensemble des interfaces fournies en tant que service) ainsi qu'un ensemble de propriétés donnant des informations sur le fournisseur. Suite à l'enregistrement, le fournisseur reçoit un talon d'enregistrement (service registration) lui permettant de modifier les propriétés du service ainsi que de retirer le service.

Un consommateur de service utilise également un *bundle context* afin d'accéder à l'annuaire de service. Afin de trouver un service, le consommateur peut demander à l'annuaire l'ensemble des fournisseurs services disponibles en fonction du service recherché ainsi que d'un filtre. Le filtre est exprimé dans le langage LDAP et peut porter sur n'importe quelle propriété. En réponse à cette requête, l'annuaire renvoie l'ensemble des offres de services compatibles avec la requête. Chaque offre de service est décrite sous la forme d'une référence de service. Lorsque le consommateur a choisi son service, il peut avoir accès au service en demandant à l'annuaire l'objet de service associé à la référence choisie. L'annuaire renvoie une référence directe sur le fournisseur utilisable directement par le consommateur.

Lorsque le consommateur n'utilise plus le service, il doit explicitement relâcher le service et ne plus garder aucune référence sur l'objet de service utilisé. Ceci est une contrainte extrêmement importante dans le développement d'applications OSGi. En effet, comme OSGi est une plate-forme centralisée, les *bundles* désinstallés doivent être déchargés de la mémoire. Or, ceci n'est possible que si toutes les références sur les objets des classes contenues dans ce *bundle* sont relâchées. Si, un consommateur ne relâche pas une des références, le *bundle* ne peut pas être désinstallé complètement ce qui empêchera la mise à jour du *bundle* ou la réinstallation d'un *bundle* contenant des classes similaires.

OSGi supporte également la recherche passive. Un consommateur peut être notifié des arrivées et des départs des fournisseurs de services (ainsi que des modifications éventuelles des propriétés publiées). Cependant bien que cette notification soit facultative, elle est vivement recommandée, car permet de relâcher les fournisseurs de service utilisés lorsqu'ils disparaissent.

OSGi ne définit pas clairement de modèle de composition. Toute composition doit être gérée par le développeur ainsi que le dynamisme influençant cette composition. Malheureusement, la mise en place de telle composition est très complexe. En effet, le dynamisme des services peut valider ou invalider une composition à tout moment. Le développeur doit gérer l'enregistrement et le retrait des services fournis par sa composition en fonction des arrivées et des départs de services ainsi que vérifier le relâchement des fournisseurs de services disparaissant. De plus, le double niveau de dépendances (dépendances inter-*bundles* et dépendances de services) rend le développement d'applications sur OSGi très délicat. En conclusion, OSGi™ fournit tous les mécanismes pour créer des applications à service dynamique. De plus, la plate-forme OSGi™ propose des fonctionnalités d'administration et de déploiement qui en font une plate-forme extrêmement populaire aujourd'hui. Cependant le modèle de développement délicat fourni par OSGi™ et le manque d'infrastructure pour exécuter des compositions sont des problèmes récurrents dans le développement d'application dynamique au dessus de cette plate-forme. Ce tableau ci-dessous positionne OSGi™ par rapport aux critères choisis.

Critères	Valeurs
Langage utilisé pour les spécifications de service	Interface Java
Information contenue	Syntaxique (interface)
***Forme* du courtier**	Annuaire de service
Localisation du courtier	Centralisé et Unique
Implantation de la publication et du retrait de service	Publication et retrait manuel
Type de découvertes supportées	Active et Passive
Langage de requête et filtrage	LDAP
Implantation de la notification	Évènement envoyé aux consommateurs
Protocole de liaison	Java (référence directe)
Type de composition supporté	Non supportée de base
Infrastructure proposée pour exécuter les compositions	Pas d'infrastructure proposée

Tableau 18. Positionnement d'OSGi par rapport aux critères

2.5. Les Services Web

Les services web sont de loin le SOA le plus populaire aujourd'hui. D'après [159] et [160], les services web sont apparus pour permettre à des applications hétérogènes s'exécutant au sein de différentes entreprises de pouvoir interopérer 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. Au dessus des concepts de bases des services web se sont créés de nombreuses spécifications et technologies transformant ce SOA en SOA étendu.

Dans les services web « standards », les spécifications de service sont décrites dans un langage spécifique appelé langage de description de service web (Web Service Description Language ou WSDL). La spécification d'un service web décrit l'interface de service, mais également les types de données employés par le service, le protocole de transport utilisé pour communiquer avec ce service ainsi que des informations permettant de localiser le fournisseur de service. Cette dernière donnée est assez étonnante, car elle ne permet pas le couplage-faible promu par l'approche à service. À l'aide d'autres spécifications telles que WS-

Choreography, Semantic Web Services Architecture, cette description peuvent également contenir des informations sur l'ordre des fonctions à appeler ou des informations sur la sémantique du service.

Les services web ont défini un *registre de service*, appelé UDDI (Universal Description, Discovery and Integration). Cet annuaire supporte la publication de descriptions de service (appelé *types de services*) ainsi que les fournisseurs de services (appelé *enterprises*). 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.).

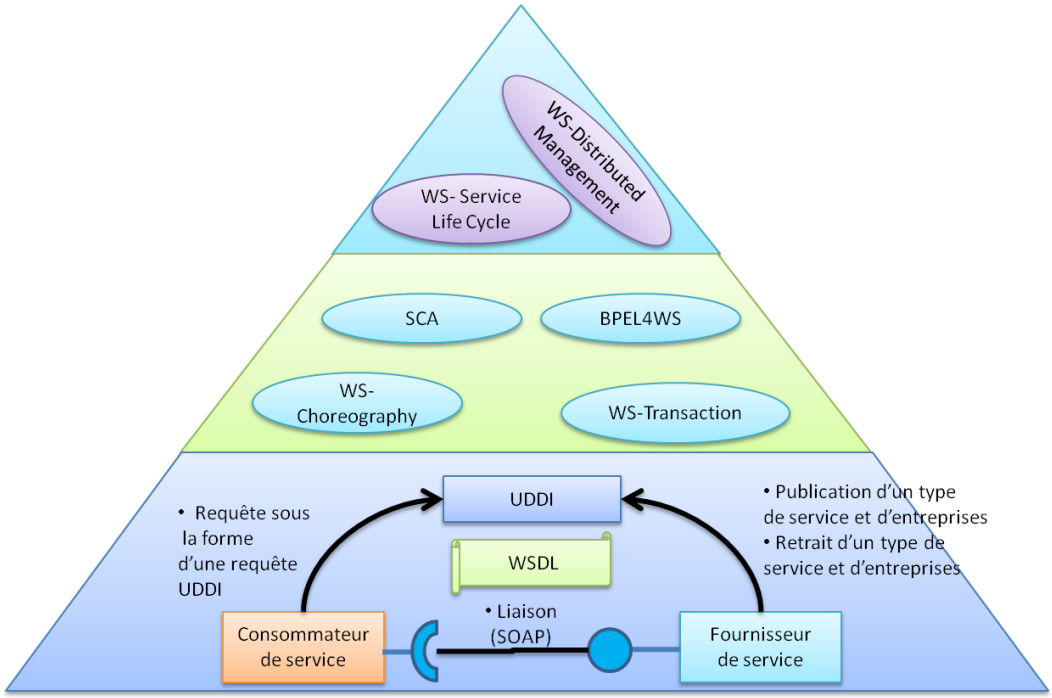


Figure 43. Architecture SOA étendu des services web

Il est courant que le registre UDDI se publie lui-même comme service web. Afin de publier un service, un fournisseur peut communiquer avec ce registre en utilisant le protocole standard de communication des services web, appelé SOAP (Simple Object Access Protocol). Cependant, il doit en connaitre la localisation. L'enregistrement d'un service se fait en deux étapes. Tout d'abord, le fournisseur doit publier son type de service. Ensuite, il doit se publier. Lors de l'enregistrement, le fournisseur peut se décrire afin de donner de plus amples informations sur le service fourni. UDDI est généralement 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 nœud du réseau de registres. Les fournisseurs peuvent également se retirer de l'annuaire ainsi que de retirer les types de services fournis. Une particularité des services web est qu'UDDI ne possède pas le même modèle de description que le WSDL.

La consommation de service dans les services web ne passe pas forcément par le registre. En effet, la plupart des applications utilisant ce SOA, indique directement la localisation des fournisseurs de service. Ceci est dotant plus facilité par la présence de la localisation dans la description de service. La recherche du fournisseur s'effectue durant le développement de l'application. Néanmoins, il est tout de même possible d'utiliser le registre UDDI pour trouver des fournisseurs à l'exécution. Comme pour le fournisseur, le consommateur peut communiquer avec le registre en utilisant le protocole de communication SOAP. Le registre UDDI fournit diverses opérations de découvertes :

- *find_business* : retourne des informations sur un ou plusieurs fournisseurs de services.
- *find_service* : retourne des informations sur des services fournis par des entreprises enregistrées.

Les méthodes de découverte permettent de faire des recherches sur des informations approximatives à travers des expressions 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 ordonnées suivant plusieurs critères (ordre alphabétique, date d'enregistrement, présence de certificat, etc.). La localisation du WSDL est disponible via le registre UDDI. Grâce à cette information, il est possible à un consommateur d'utiliser le service.

UDDI ne supporte pas la découverte passive. En effet, les services web ne définissent pas de mécanisme de notification le permettant. Dans les extensions à UDDI, beaucoup ont été proposées afin de supporter d'autres types de requêtes telles que des requêtes sémantiques [161, 162].

La composition comportementale de services web est fondée sur deux activités différentes : l'orchestration et la chorégraphie. L'orchestration de services fait référence à l'activité de création de processus, exécutable ou non, qui utilise 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[163].

La composition de service web est basée sur une technique de flot (workflow). Par exemple, BPEL4WS (Business Process Execution Language for Web Services) [144] 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 permet 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.

De plus, les services web sont composables structurellement via le modèle de composition proposé dans Service Component Architecture. Ce modèle sera étudié plus amplement dans la section suivante. Cependant, à la fois la composition comportementale et structurelle ne supporte pas le dynamisme.

Bien que les services web soient fondés sur les concepts de l'approche à services, l'interaction propre à cette approche n'est actuellement pas suivie de façon systématique. De plus, les services web n'implantent pas les primitives nécessaires pour gérer correctement le dynamisme. 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 section. De plus, le principe de registre et plus particulièrement UDDI ne sont pas utilisés. La majorité des organisations qui fournissent des services web aujourd'hui le font sans passer par des registres. Le tableau suivant positionne ce SOA par rapport aux critères retenus.

Critères	Valeurs
Langage utilisé pour les spécifications de service	WSDL
Information contenue	Syntaxique (interface, protocole, localisation) Des extensions sont proposées pour supporter des informations comportementales ou sémantiques
***Forme* du courtier**	Registre de service (UDDI)
Localisation du courtier	Décentralisé
Implantation de la publication et du retrait de service	Publication et retrait manuel
Type de découvertes supportées	Active
Langage de requête et filtrage	Langage spécifique à UDDI
Implantation de la notification	Non supportée
Protocole de liaison	SOAP
Type de composition supporté	Comportementale (Orchestration, Chorégraphie) Structurelle (SCA)
Infrastructure proposée pour exécuter les compositions	Moteurs d'orchestration, Bus à service…

Tableau 19. Positionnement des services web

2.6. UPnP & DPWS

Universal Plug and Play (UPnP) [164] est un ensemble de protocoles réseaux spécifié par l'UPnP forum [165]. Le but d'UPnP est de permettre aux dispositifs physiques de la maison de se connecter entre eux de manière transparente et de simplifier la mise en place de réseau domotique. UPnP spécifie un ensemble de protocoles basés sur les protocoles standards d'Internet pour mettre en place une architecture à service dynamique spécialisé dans la domotique, ainsi que spécifie des profiles en fonction du domaine. Chaque profil décrit la description des dispositifs du domaine. Par exemple, dans le profil audio/vidéo est décrit la description d'une télévision (media renderer) ainsi que d'un disque dur multimédia (média server).

Grâce à ces protocoles, un dispositif compatible UPnP peut dynamiquement rejoindre un réseau, obtenir une adresse IP, annoncer son nom, fournir la liste de ses capacités (services), détecter les autres dispositifs ainsi que découvrir leurs services. Un concept clé dans UPnP est le concept de dispositif (*device*). C'est un dispositif qui peut être découvert, c'est également un dispositif qui fournit des services.

Lorsqu'un dispositif rejoint un réseau, il s'annonce auprès de tous les autres acteurs présents sur le réseau. Cette annonce est diffusée sur le réseau local. Les informations échangées lors de la découverte sont très réduites et concernent particulièrement le dispositif en lui-même (nom, modèle…). Un acteur du réseau désirant découvrir les dispositifs de ce réseau peut également mettre une requête afin d'obtenir des informations sur l'ensemble des dispositifs présents. Lorsqu'un dispositif s'arrête, il émet également un message diffusé sur le réseau pour notifier son départ. Le protocole de découverte proposé par UPnP est Simple Service Discovery Protocol (SSDP) [166]. Un point important est qu'UPnP ne possède pas d'annuaire de services réels. Chaque client doit maintenir cette liste en fonction des évènements émis sur le réseau.

Une fois qu'un dispositif est découvert, il est possible d'interagir avec lui afin de lui demander des informations plus spécifiques telles que les services hébergés, les informations sur le vendeur ou le constructeur, une URL de présentation[12] … Une particularité d'UPnP est que les dispositifs publient leurs variables d'état. De plus, pour chaque service, le dispositif décrit la liste des actions appelables. Cette description est décrite en XML dans un format propre à UPnP et ne contient donc que des informations syntaxiques. Ensuite, il est enfin possible d'interagir avec le dispositif via les services proposés. Les actions décrites sont invocables en utilisant SOAP (le même protocole que pour les services web).

[12] Cette adresse permet d'obtenir une page web de contrôle du dispositif

UPnP propose également un moyen afin d'être notifié des changements d'état d'un dispositif. Lorsqu'une variable d'état est modifiée, le dispositif émet une notification sur le réseau contenant l'identifiant du dispositif, la variable d'état et la nouvelle valeur. Il s'agit d'ailleurs du seul moyen pour interagir de manière asynchrone avec UPnP. UPnP utilise le protocole General Event Notification Architecture (GENA)[167].

Bien qu'UPnP commence à avoir un certain succès, plusieurs problèmes sont à corriger. Tout d'abord, UPnP utilise un ensemble de protocoles hétérogènes. De plus, UPnP ne spécifie rien sur la sécurité et sur l'authentification. Cependant, dans le contexte de l'informatique résidentiel, la sécurité est une propriété cruciale. De plus, UPnP utilise du HTTP sur de l'UDP (également appelé HTTPU/HTTMU) [168]. Ce protocole est aujourd'hui obsolète et n'a jamais été standardisé[13].

Devices Profile for Web Services (DPWS) est considéré aujourd'hui comme la suite d'UPnP. Cette spécification, lancée en 2004, vise à corriger les problèmes d'UPnP. Bien que les concepts restent les mêmes, l'ensemble des protocoles choisis est homogène et provient des services web. De plus, DPWS propose des concepts de sécurité à l'aide de WS-Security et WS-Policy. À l'instar des services web, DPWS utilise HTTP sur TCP.

Critères	UPnP	DPWS
Langage utilisé pour les spécifications de service	Format spécifique à UPnP	WSDL avec extension
Information contenue	Syntaxique (interface, variable d'état). Les services fournis sont décrits dans la description du dispositif	Syntaxique (interface, évènement) et comportementale (politique). Les services fournis sont décrits dans la description du dispositif
***Forme* du courtier**	Pas de courtier. Chaque client doit maintenir son courtier	Pas de courtier. WSD-API propose un courtier pour Windows Vista
Localisation du courtier	Pas de courtier centralisé	Pas de courtier centralisé
Implantation de la publication et du retrait de service	Publication et retrait par émission d'évènement	Publication et retrait par émission d'évènement
Type de découvertes supportées	Active & Passive	Active & Passive
Langage de requête et filtrage	Pas de langage de requête	Pas de langage de requête
Implantation de la notification	Émission d'évènement	Émission d'évènement
Protocole de liaison	SOAP	SOAP
Type de composition supporté	Non supportée	Non supportée, bien qu'il soit possible d'appliquer la composition de services web.

Tableau 20. Positionnement d'UPnP et de DPWS

Cependant, bien que corrigeant les principaux problèmes d'UPnP, DPWS suit toujours le même type de découverte qu'UPnP (ceux sont les dispositifs qui sont découverts et non pas les services[14]). Ainsi, un client devra tout d'abord découvrir l'ensemble de dispositifs présents, leur demander leurs informations respectives et enfin essayer de trouver le service requis. De plus, chaque client doit maintenir son propre annuaire de service. WSD-API proposé par Microsoft et intégré dans Windows Vista [169] permet justement de corriger les deux problèmes en fournissant un annuaire unifié interrogeable par les applications. D'autres travaux tel que [170] rajoute la notion de contexte au protocole de découverte utilisé.

[13] HTTPU est un Internet-Draft qui a expiré en 2001.

[14] UPnP permet également la recherche active de service. Ce type de requête permet d'obtenir la liste des dispositifs hébergeant ce service. La recherche passive de service n'est néanmoins pas autorisée.

Bien que DPWS spécifie précisément les dispositifs et la façon d'interagir avec eux, de nombreuses capacités sont optionnelles [171]. Cet ensemble d'options risque à terme de poser des problèmes de compatibilité entre des clients et des dispositifs. Le tableau ci-dessus positionne UPnP et DPWS par rapport aux critères choisis.

2.7. Synthèse

Cette section a présenté quelques technologies SOA. Nous pouvons remarquer que ces technologies soit ne supportent pas le dynamisme (services web), soit ne fournissent pas les fonctionnalités des SOA étendus dynamiques. En effet, alors que ces SOA implantent les primitives de l'approche à service dynamique, ils ne fournissent pas les fonctionnalités nécessaires à la composition de service ni à la gestion des applications à service. Ces taches doivent être gérées par les développeurs qui généralement mettent en place des solutions ad hoc. Cependant, afin de créer un service composite, les développeurs doivent traquer les services requis afin de pourvoir fournir leur service. Le modèle de développement devient alors très complexe et délicat. En effet, les SOA présentés ont tendance à combiner le code métier et le code gérant l'aspect dynamisme. Ce mélange va à l'encontre du principe de séparation des préoccupations.

L'approche à service a donné naissance à un nouveau type de modèle à composant appelé modèle à composant à service. Ces modèles à composant reprennent les principes de l'approche à service afin d'exhiber des propriétés telles que le faible couplage, le dynamisme et la flexibilité. Ils gardent néanmoins les avantages des modèles à composant (composition, reconfiguration …). La prochaine section étudiera cette nouvelle sorte de modèle à composant.

3. Modèles à composant à service

L'étude réalisée dans le chapitre précédent révèle certains aspects de l'approche à composant qui représentent des limitations pour supporter le dynamisme :

- Les applications sont assemblées durant la conception de l'application, en réalisant une composition à partir des types de composants disponibles au préalable. Cette situation est cependant indésirable lors de l'introduction, pendant l'exécution, de nouveaux types de composants (non disponibles durant la conception). Le type de dynamisme le plus couramment géré est l'évolution d'un type de composant.
- La composition de composants réalisée de façon déclarative rend difficile l'expression de changements pouvant avoir lieu pendant l'exécution. La composition donne généralement lieu à une architecture statique. Peu de modèles à composant expriment le dynamisme dans l'architecture de l'application.
- Le retrait d'un type de composant ou d'une instance de composants utilisés pendant l'exécution n'est pas une hypothèse de l'approche à composant.

D'un autre côté, l'approche à service dynamique, présentée dans ce chapitre, offre certains aspects avantageux par rapport au support du dynamisme :

- Le motif d'interaction de l'approche à service supporte la substituabilité des fournisseurs. Un consommateur peut donc utiliser n'importe quel fournisseur disponible offrant le service requis.
- Le dynamisme des services est une hypothèse qui fait partie de l'approche à service dynamique: à tout moment un fournisseur de services peut arrêter de fournir ses services ou bien de nouveaux fournisseurs de services peuvent publier leurs services.

- La composition de services n'est pas réalisée par rapport à des fournisseurs de service spécifiques, la composition de service s'exprime en termes de services et ce n'est qu'au moment de l'exécution que des fournisseurs de services sont localisés et liés.

C'est pourquoi l'approche à service dynamique a donné naissance à un nouveau type de modèle à composant alliant les avantages de l'approche à composant et de l'approche à service. Cette section décrit les principes de ces modèles à composant à service. De plus, cette section comparera quelques-uns de ces modèles afin de déterminer s'ils peuvent être considérés comme des SOA étendus dynamiques.

3.1. Principes des modèles à composant à service

Afin d'aider au développement de systèmes dynamiques, les composants à service proposent d'utiliser des composants pour implémenter des services [172]. Cette motivation émerge du besoin de séparer le code de gestion du dynamisme (et donc des aspects services), du code métier implémentant la logique des services fournis. Le but est de déléguer les mécanismes liés à l'approche à service dynamique au conteneur du composant. Celui-ci interagira à l'exécution avec le courtier de service du SOA sous-jacent afin de trouver les services requis et publier les services fournis.

Les principes des modèles à composant à service ont été introduits dans [173]. Ces principes sont les suivants:

- Un service est une fonctionnalité fournie
- Un service est caractérisé par une spécification de service qui peut contenir des informations syntaxiques, comportementales et sémantiques ainsi que des dépendances sur d'autres spécifications de service
- Un composant implémente des spécifications de service et peut dépendre d'autres services
- Le motif d'interaction de l'approche à service est utilisé pour résoudre les dépendances de service
- Les compositions de service sont décrites en termes de spécifications de service
- Les spécifications de service fournissent la base de la substitution

Le modèle résultant est à la fois flexible et puissant. Il permet la substitution de service, car les compositions sont exprimées en terme de spécification de service et non pas en terme d'implémentation spécifique (approche employée dans l'approche à composant). Cette notion de substitution est basée sur un double niveau de dépendance. En effet, les spécifications de service peuvent exprimer des dépendances sur d'autres spécifications (dépendances de spécification). Les composants peuvent également dépendre d'autres services (dépendances d'implémentation). Les modèles à composant traditionnels ne supportent que les dépendances d'implémentation empêchant la substitution d'un type de composant par un autre. L'approche à service ne décrit généralement pas des dépendances de spécification, ce qui a malheureusement pour effet de limiter la composition structurelle de service. L'utilisation du motif d'interaction de l'approche à service est utilisée pour résoudre les dépendances de service à l'exécution. Ceci permet de retarder le choix des implémentations de service (c'est-à-dire les fournisseurs de service) jusqu'à l'exécution.

Lorsqu'une application est construite en suivant le paradigme de l'approche à service, l'application est décomposée en un ensemble de services. La sémantique et le comportement de ces services sont soigneusement décrits en dehors des futures implémentations (dans les spécifications de service). En suivant ce principe, il est possible d'implémenter les services constituant l'application indépendamment les uns des autres. De plus, il est possible d'avoir différentes implémentations interchangeables. Ces différentes implémentations peuvent être utilisées pour, par exemple, supporter différentes plates-formes ou différents besoins non-fonctionnels.

De plus, le fait qu'un service représente une fonctionnalité locale ou distante n'a pas à être spécifié dans les dépendances de service. En effet, si un service représente une fonctionnalité distante, l'implémentation de ce service sera vraisemblablement un *proxy* donnant accès à cette fonctionnalité. Ce service sera alors traité comme les autres services locaux par le composant.

Dans les modèles à composant traditionnels, la sélection des composants constituant une composition est effectuée durant la conception de cette application. Dans une composition de service, le processus de sélection a lieu à l'exécution. L'exécution d'une composition démarre au moment ou toutes les dépendances de services sont satisfaites. Lorsque certains services requis ne sont plus disponibles, la composition est arrêtée.

3.2. Modèles à composant à service & SOA étendus dynamiques

La section précédente a décrit les principes des modèles à composant à service. Or, ceux-ci peuvent être utilisés afin de créer des SOA étendus dynamiques. En effet, ces modèles dépendent nécessairement d'un SOA dynamique sous-jacent (Figure 44). Cependant, il propose un modèle de développement afin de séparer les mécanismes de ce SOA et la logique-métier (encapsulée dans les composants)

Ensuite, ils fournissent les fonctionnalités nécessaires afin de créer des compositions de service tout en gérant le dynamisme. Enfin, comme ces services composites seront également des composants à service, il est possible de les superviser à l'exécution. Il est également possible d'introspecter ces services composites afin de déterminer si la composition actuelle remplit les contraintes métier de l'application. De plus, les préoccupations non-fonctionnelles de l'application peuvent être gérées soit par la composition, soit par les composants implémentant les services. Ces actions peuvent également être réalisées par un gestionnaire autonomique reposant sur les mécanismes du modèle à composant sous-jacent.

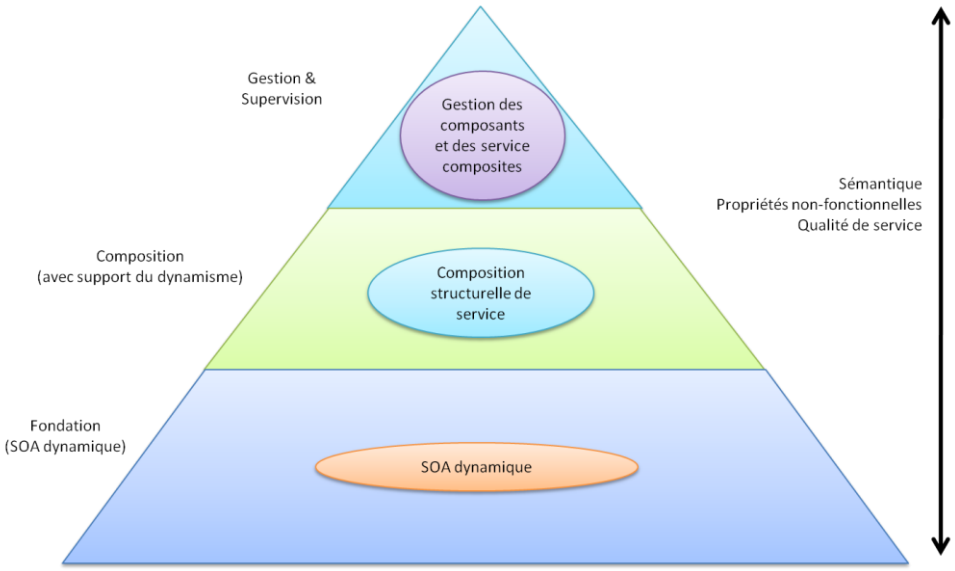


Figure 44. SOA étendu et modèles à composant à service

Bien que les principes des modèles à composant à service tendent à créer des compositions de service structurelles, il est également possible de composer les services de manière comportementale. En effet, une composition comportementale de service serait alors utilisée comme une implémentation d'un service. Cette approche est utilisée dans SCA où il est possible de définir des composants utilisant une composition BPEL comme implémentation.

3.3. Caractérisation des modèles à composant à service et études de l'existant

Cette section a pour but de définir une caractérisation des modèles à composant à service ainsi que de positionner les implémentations existantes. Un point important qui sera étudié et si ces modèles à composant peuvent être considérés comme des SOA étendus dynamiques.

3.3.1. Caractérisation des modèles à composant à service

Le chapitre précédent a introduit une caractérisation des approches pour créer des applications dynamiques. Cette section reprend cette caractérisation et détermine le positionnement global des modèles à composant à service. En tout cas, ces modèles ont d'autres particularités qui seront rajoutées dans cette caractérisation.

Tout d'abord, la caractérisation du chapitre précédent proposait trois critères sur les politiques d'adaptations. Dans les modèles à composant à service, les adaptations sont initialisées par des notifications du SOA dynamique sous-jacent. Ces notifications informent le composant de la disponibilité ou de la disparition d'un fournisseur. Ensuite, la politique d'adaptation réside dans l'expression des dépendances de services. En effet, ce sont ces dépendances qui ont principalement en charge le dynamisme et les adaptations à effectuer (changement de fournisseurs, invalidation de l'instance …) Cependant, celle-ci peut se faire de manière déclarative, dans du code … Le dernier critère concernant l'expression de la politique d'adaptation concerne la gestion de ces politiques. Dans le contexte de modèles à composant à service, ces politiques sont exprimées dans les dépendances de service. Cependant, l'ajout, le retrait et la reconfiguration des dépendances de service ne sont pas forcément supportés.

Les trois autres critères concernaient l'infrastructure d'exécution. Le premier critère intéressera au positionnement du gestionnaire d'adaptation. Dans les modèles à composant à service, il est inclus dans le conteneur des instances. Le deuxième critère se focalisait sur les mécanismes utilisés pour gérer les adaptations. Dans les modèles à composant à service, il existe plusieurs tendances. Cependant, tous utilisent les motifs d'inversion de contrôle et d'injection de dépendances. Les technologies afin de réaliser ces motifs varient d'un modèle à un autre. Le dernier critère concerne la gestion des interruptions lors des adaptations. Dans les modèles à composant à service, ces interruptions sont souvent gérées instance par instance (et pas de manière globale). Cependant, grâce au faible-couplage et au principe de découverte, cette gestion permet de minimiser, voir d'annuler les interruptions de service lors d'un changement de fournisseurs.

Critères	
Initialisation des adaptations	Notification provenant du SOA sous-jacent
Localisation de la logique d'adaptation	La politique d'adaptation est exprimée dans les dépendances de service
Gestion des politiques d'adaptation	Variable
Positionnement du gestionnaire d'adaptation	Le gestionnaire d'adaptation se situe dans le conteneur des instances
Mécanisme utilisé pour l'adaptation	Variable, mais s'appuie toujours sur les motifs d'inversion de contrôle et d'injection de dépendance
Gestion des interruptions	Généralement minimisée

Tableau 21. Application de la caractérisation des approches pour les applications dynamiques aux modèles à composant à service

Le tableau ci-dessus décrit le positionnement des modèles à composant à service par rapport aux critères caractérisant les approches permettant de mettre en place des applications dynamiques. Cependant, les modèles à composant à service ont également leur propre particularité.

Tout d'abord, le SOA sous-jacent est un critère important pour distinguer les modèles à composant à service. Bien que la plupart se basent sur OSGi™, il est envisageable de créer un modèle à composant à service au dessus de n'importe quel SOA dynamique. Le SOA choisi doit nécessairement fournir la notification.

Les dépendances de services sont un point clé des modèles à composant à service. En effet, la politique d'adaptation est exprimée dans ces dépendances de services. Celles-ci doivent être exprimées en terme d'une spécification de service (car suivent le paradigme de l'approche à service), mais peuvent posséder d'autres attributs tels que le filtrage, l'optionalité, la cardinalité, l'ordre de classement ainsi que la politique de liaison.

Un troisième point important est la gestion du cycle de vie des services publiés. En effet, en fonction de l'état des dépendances de service, il est souvent nécessaire de publier ou de retirer les services fournis. Tous les modèles à composant à service ne proposent pas cette fonctionnalité. De plus, cette fonctionnalité lorsqu'elle est supportée peut être limitée. En effet, en appliquant les principes d'inversion de contrôle, le code du développeur ne peut pas influencer le retrait ou la publication d'un service.

Un point important caractérisant les modèles à composant à service concerne les mécanismes de composition. En effet, certains modèles ne fournissent pas de mécanisme de composition et se sont focalisés sur le modèle de développement. D'autres fournissent des mécanismes de composition ainsi que le modèle de développement. Cependant, celle-ci ne suit pas forcément le paradigme de l'approche à service, et donc n'apporte pas la flexibilité attendue. D'un autre côté, certains modèles ne supportent que la composition comportementale.

Le dernier critère caractérisant les modèles à composant à service étudie les mécanismes mis en place par le modèle à composant afin de pouvoir introspecter et reconfigurer les applications. Ceci est utilisé afin d'atteindre le troisième niveau de la pyramide du SOA étendu dynamique. En effet, grâce à ces interfaces de contrôle il est possible de vérifier la cohérence de l'application par rapport aux contraintes métier. Il doit être également possible de reconfigurer les instances de l'application, et plus particulièrement les dépendances de services, afin de modifier leur comportement.

Critères	
SOA sous-jacent	OSGi™, Service Web, Support du dynamisme
Description des dépendances de service	Optionalité, Filtrage, Cardinalité, Ordre de classement, Politique de liaison
Gestion du cycle de vie des services	Non géré, Géré, Géré en collaboration avec le contenu de l'instance
Support de la composition	Non supporté, Supporté, mais non exprimé en terme de service, Supporté et exprimé en terme de service, Composition comportementale
Infrastructure de supervision	Introspection, Introspection et reconfiguration (avec interruption), Introspection et reconfiguration dynamique

Tableau 22. Critères spécifiques aux modèles à composant à service

3.3.2. Declarative Services

Declarative Services est un modèle à composant à service au dessus OSGi™ fortement inspiré de Service Binder [173, 174]. Declarative Services est spécifié dans la spécification OSGi™ R4 [175].

Declarative Services a pour but principal de faciliter le développement d'application à service sur OSGi™. En effet, le modèle de développement d'OSGi™ étant extrêmement délicat, il est apparu nécessaire de fournir un modèle à composant permettant de le simplifier[176]. Les composants au dessus de Declarative Services sont décrits dans un descripteur XML et peuvent requérir et fournir des services. Les dépendances de services

sont injectées dans l'implémentation du composant via des méthodes *bind* et *unbind* appelées respectivement lors de l'apparition d'un fournisseur et lors de la disparition d'un fournisseur. Malgré ce modèle d'injection, le développeur doit gérer dans son code ces appels (asynchrones). Ainsi, le modèle de développement de Declarative Services ne sépare pas totalement la gestion du dynamisme et la logique-métier (Figure 45).

```
public class Example {
        LogService log;

        protected void bindLog(LogService log) {
                this.log = log;
        }
        protected void unbindLog(LogService log){
                this.log = log;
        }
        protected void activate(ComponentContext context ) {
                this.log.log(LogService.INFO, "Activation...");
        }
        protected void deactivate(ComponentContext context ){
                thislog.log(LogService.INFO, "Deactivation...");
        }
}
```

Figure 45. Modèle de développement proposé par Declarative Services

Declarative Service fournit un modèle de dépendance relativement riche. Toute dépendance de service doit définir le service requis (c'est-à-dire l'interface de service). De plus, celles-ci peuvent être obligatoires ou optionnelles, filtrées, multiples ou simples et dynamiques ou statiques. Ce dernier attribut définit la politique de liaison :

- Dynamique : l'instance choisit un autre fournisseur dès que celui utilisé disparaît.
- Statique : lorsque la liaison est établie avec un fournisseur, si celui-ci disparaît, l'instance est détruite. Cette politique de liaison est souvent utilisée lorsque le dynamisme doit être limité.

Grâce à ce modèle de dépendance, Declarative Services est capable de gérer le cycle de vie des services fournis. Lorsque les dépendances obligatoires ne sont pas satisfaites, les services fournis par l'instance ne sont pas publiés. Ceux-ci seront fournis dès que les dépendances seront satisfaites. Pour cela, le modèle définit le cycle de vie des composants. Une fois démarrée et configurée, une instance est soit valide ou invalide. Une instance valide fournit ses services, alors qu'une instance invalide ne peut pas fournir de service.

Cependant, bien que proposé dans Service Binder, Declarative Services ne propose pas de modèle de composition. Afin de créer des services composites, le développeur doit coder cette composition. De la même manière, Declarative Services ne permet pas d'introspecter les instances et de découvrir l'architecture actuelle du système. Cependant, Declarative Services décrit comment les instances peuvent être reconfigurées. Cette reconfiguration oblige l'instance à être arrêtée puis redémarrée. De plus, cette reconfiguration n'influe pas sur les dépendances de services qui ne peuvent pas être reconfigurées.

Critères	
SOA sous-jacent	OSGi™
Description des dépendances de service	Optionalité, Filtrage, Cardinalité, Ordre de classement, Politique de liaison
Gestion du cycle de vie des services	Gérée
Support de la composition	Non supporté
Infrastructure de supervision	Reconfiguration des instances (avec arrêt et redémarrage)
Gestion des politiques d'adaptation	Fixe
Mécanisme utilisé pour l'adaptation	Injection des dépendances via des appels de méthode par réflexion

Tableau 23. Caractérisation de Declarative Services

3.3.3. Dependency Manager

À l'inverse de Declarative Services, Dependency Manager [177] est un modèle à composant non déclaratif. Les composants sont décrits à l'aide d'une API et non pas à l'aide d'un descripteur. Le but de Dependency Manager est relativement proche de celui de Declarative Services, c'est-à-dire simplifier le modèle de développement afin de créer des applications à service dynamique plus facilement. Tout comme Declarative Services, Dependency Manager se base sur OSGi™.

Dependency Manager propose une séparation claire entre le code métier et le code spécifique à OSGi™ (dont le code gérant le dynamisme). De plus, il automatise une grande partie du code lié à OSGi™ tout en proposant une infrastructure très flexible. En effet, via l'API proposée, le développeur peut configurer ces instances de composant de manière très fine. De plus, il est possible intégralement de reconfigurer une instance (y compris les dépendances de services).

Un composant est décrit grâce à l'API. Celui-ci est tout d'abord configuré. Durant cette configuration, les services fournis ainsi que les services requis sont spécifiés. Les services requis ciblent une spécification de service définie (c'est-à-dire une interface de service). Ensuite, ils peuvent être optionnels ou obligatoires et supporter le filtrage.

Une caractéristique importante du Dependency Manager est l'utilisation du motif de conception *Null Object*. Afin de simplifier le modèle de développement, lorsqu'un service optionnel n'est pas disponible, le Dependency Manager injecte un objet spécial sur lequel les méthodes du service requis peuvent être appelées. Ces appels ne font aucune action. Ceci évite de tester en permanence la disponibilité du service.

Dependency Manager gère les cycles de vie des instances et des services publiés (Figure 46). Le cycle de vie proposé par le Dependency Manager est plus complexe que celui de Declarative Services. Cependant, le principe reste similaire. Une instance devient valide lorsque tous les services requis obligatoirement sont disponibles. Les services fournis sont ensuite publiés dans l'état *starting*. Lorsqu'un des services requis n'est plus disponible, les services fournis sont retirés de l'annuaire (dans l'état *stopping*).

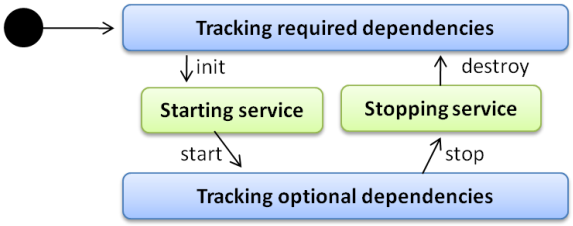


Figure 46. Cycle de vie des composants gérés avec le Dependency Manager

Critères	
SOA sous-jacent	OSGi™
Description des dépendances de service	Optionalité, Filtrage
Gestion du cycle de vie des services	Gérée
Support de la composition	Non supporté
Infrastructure de supervision	Reconfiguration des instances et des dépendances de service
Gestion des politiques d'adaptation	La politique d'adaptation peut être changée via les dépendances de service
Mécanisme utilisé pour l'adaptation	Injection des dépendances via des appels de méthode par réflexion

Tableau 24. Caractérisation du Dependency Manager

Dependency Manager ne propose pas de mécanisme pour la composition de service. Ainsi le développeur doit implémenter sa composition à la main en utilisant l'API proposée. Les instances créées avec le Dependency Manager sont reconfigurables dynamiquement. En plus, à la fois la logique-métier peut être reconfigurée, mais également les dépendances de service.

3.3.4. Spring Dynamic Modules

Spring Dynamic Modules [178], anciennement connu sous le nom de Spring-OSGi™, est un modèle à composant à service au dessus d'OSGi™. Ce modèle est issu du portage du canevas Spring [106] sur la plate-forme OSGi™. Ce portage a deux buts :

- Spring profite des capacités de déploiement et de modularité fournies par OSGi™
- Spring fournit un modèle à composant très connu utilisable au dessus d'OSGi™

Afin de supporter le dynamisme, Spring Dynamic Modules a rajouté les notions de service au canevas Spring. Ainsi un *bean* (c'est-à-dire un composant Spring) peut fournir et requérir des services OSGi™. L'idée principale de Spring Dynamic Modules est de créer des groupes de composants (appelé contexte d'application ou *application context*) à l'intérieur de *bundles* OSGi™[15] (Figure 47). À l'intérieur de ces contextes d'application, les composants ne communiquent pas avec des services, mais utilisent les mécanismes fournis par Spring. Cependant, deux groupes (c'est-à-dire deux *bundles*) communiquent via la couche service fourni par OSGi.

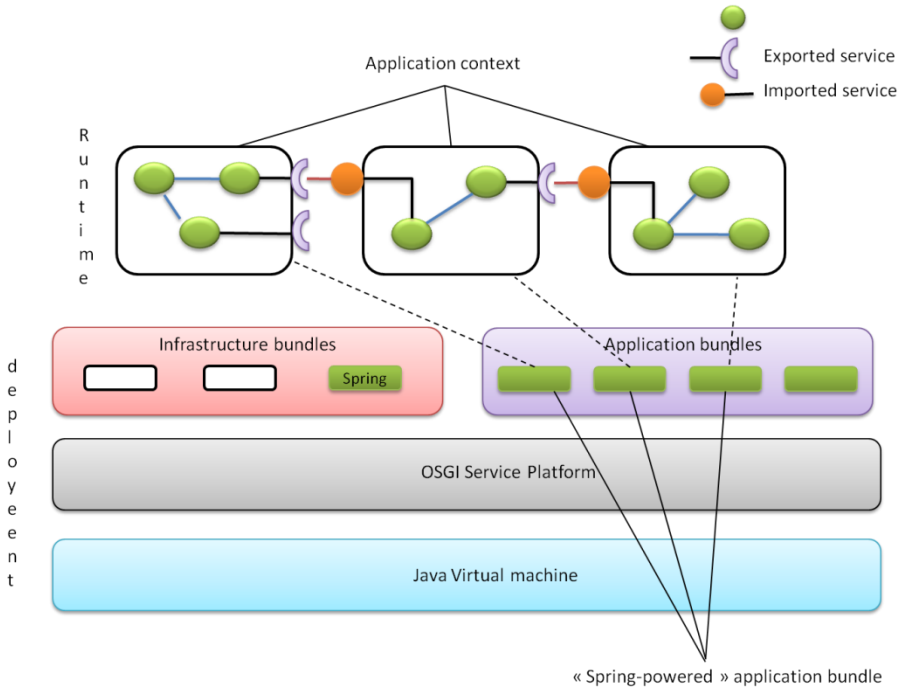


Figure 47. Fonctionnement de Spring Dynamic Modules

Les dépendances de services supportées par Spring Dynamic Modules sont décrites dans le descripteur du contexte d'application et avec les *beans* composant cette application. Ces dépendances doivent spécifier l'interface de service requise. De plus, cette dépendance peut spécifier les attributs suivants : le filtre, le nom du fournisseur désiré, l'optionalité, la cardinalité, l'ordre de classement des fournisseurs et un temps d'attente maximum. Le nom du fournisseur permet de définir un couplage fort entre cette dépendance et un fournisseur explicite. Dans les autres approches, ceci est généralement possible via le filtre. Le temps d'attente est utilisé lorsqu'un composant tente d'accéder à un service qui n'est pas disponible. Si le service n'est toujours pas accessible après le temps spécifié, une exception est levée.

[15] Spring Dynamic Modules ne supporte qu'un contexte d'application par *bundle*. De plus, celui-ci ne peut pas être réparti sur plusieurs *bundles*.

Les dépendances de service sont injectées via des méthodes. De plus, Spring injecte que des proxies interceptant les appels sur les objets de services. Lorsqu'un fournisseur est disponible, une méthode est appelée pour injecter un proxy sur ce fournisseur dans l'implémentation du *bean.*

Tout comme Declarative Services et Dependency Manager, Spring Dynamic Modules gère le cycle de vie des services fournis. Lorsque les dépendances de services d'un *bean* ne sont pas satisfaites, les services fournis par ce *bean* ne sont pas publiés.

Une propriété importante de Spring Dynamic Modules est qu'il propose un mécanisme de composition structurelle. En effet, à l'intérieur de chaque contexte d'application (c'est-à-dire *bundle*) les *beans* sont organisés structurellement. Cependant, cette composition ne suit pas le paradigme de l'approche à service (vu qu'à l'intérieur d'un contexte d'application, les composants sont connectés via l'infrastructure de Spring et non pas via des services. De plus, elle n'est pas reconfigurable. Ces compositions ne supportent que le dynamisme provenant des services OSGi™ injectés dans le contexte d'application.

En conclusion, Spring Dynamic Modules propose une infrastructure efficace pour créer des applications dynamiques. En effet, le modèle de développement est simple. Il s'appuie d'ailleurs sur les principes de POJO[179]. Cependant, le modèle de composition proposé est très limité. En effet, celui-ci ne permet pas la substitution de fournisseur de service (dans une composition) et n'est pas reconfigurable.

Critères	
SOA sous-jacent	OSGi™
Description des dépendances de service	Optionalité, Filtrage, Classement, Durée d'attente maximum
Gestion du cycle de vie des services	Gérée
Support de la composition	Composition structurelle à l'intérieur d'un contexte d'application
Infrastructure de supervision	Reconfiguration des instances en utilisant l'infrastructure Spring
Gestion des politiques d'adaptation	Fixe
Mécanisme utilisé pour l'adaptation	Injection des dépendances via des appels de méthode par réflexion. À noter que Spring Dynamic Modules injecte systématiquement des proxies.

Tableau 25. Caractérisation de Spring Dynamic Modules

3.3.5. Service Component Architecture

Service Component Architecture (SCA) [145] est un ensemble de spécifications qui décrivent un modèle structurel pour bâtir des applications utilisant une architecture orientée service. Cette solution s'articule autour de solutions standards telles que les services web [180]. Le but de SCA est de simplifier l'écriture d'application indépendamment des produits et langages utilisés. Ils existent plusieurs implémentations de SCA telles que IBM WebSphere, BEA Aqualogic, ou Apache Tuscany.

SCA encourage l'utilisation de l'approche à service afin d'architecturer les applications. Une application SCA est composée de composants offrant et consommant des fonctionnalités à travers des interfaces de service [181]. La construction d'une application SCA se fait en deux étapes :

- Implémenter les composants fournissant et requérant des services
- Assembler ces composants dans une composition

SCA propose un modèle de composition structurelle afin d'assembler les composants. Ainsi, dans SCA, une application est en fait un assemblage de type de composant (*component type*) communiquant via des services. Un type de composant peut avoir plusieurs implémentations, mais toutes doivent suivre le « patron » défini dans le type de composant. Les implémentations seront choisies lors du déploiement de l'application.

Le modèle de composition proposé par SCA est hiérarchique. Les composants utilisés dans une application peuvent eux-mêmes être une composition (Figure 48). Une composition peut définir sa hiérarchie ou celle-ci peut être amenée en utilisant une implémentation qui est elle-même une composition.

A l'intérieur d'une composition (*composite*), les composants sont liés à l'aide de *wires.* Ces connecteurs sont fixes et ne peuvent pas évoluer durant l'exécution de l'application. Une composition peut importer et exporter des services. Les services importés seront accessibles à l'intérieur de la composition. Cependant, les composants ayant accès à ces services devront être choisis statiquement[16]. De la même manière, une composition peut exporter un service qui sera alors accessible à l'extérieur de la composition.

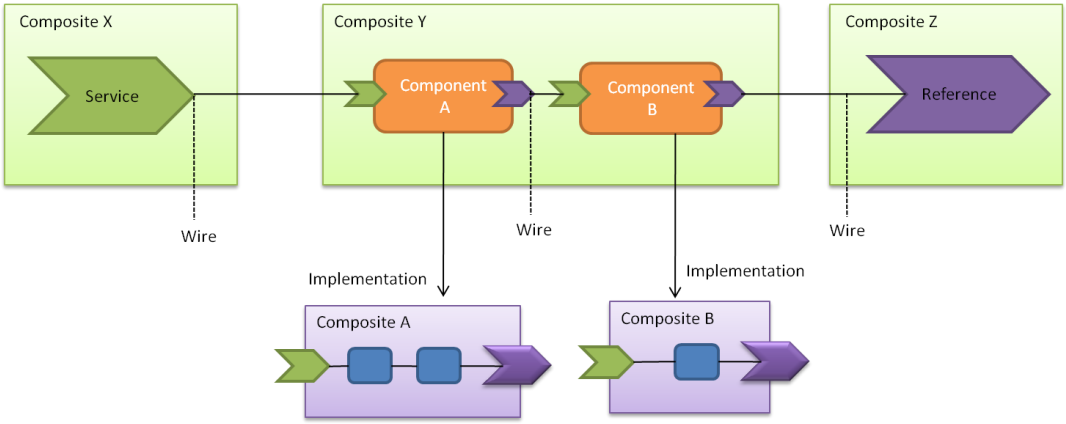


Figure 48. Exemple d'application utilisant SCA

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
 <service name="MyValueService">
        <interface.java interface="services.myvalue.MyValueService"/>
 </service>
 <reference name="customerService" multiplicity="1..n">
        <interface.java interface="services.customer.CustomerService"/>
 </reference>
 <reference name="stockQuoteService">
        <interface.java interface="services.stockquote.StockQuoteService"/>
 </reference>
 <property name="currency" type="xsd:string">USD</property>
</componentType>
```

Figure 49. Exemple de type de composant SCA

```
  <composite name="MyValueComposite" >
      <component name="MyValueServiceComponent">
          <implementation.java class="services.myvalue.MyValueServiceImpl"/>
          <property name="currency">EURO</property>
          <reference name="customerService"/>
          <reference name="StockQuoteService"/>
      </component>
      <reference name="CustomerService"
          promote="MyValueServiceComponent/customerService">
          <interface.java interface="services.customer.CustomerService"/>
          <binding.sca/>
      </reference>
      <reference name="StockQuoteService"
          promote="MyValueServiceComponent/StockQuoteService">
          <interface.java interface="services.stockquote.StockQuoteService"/>
          <binding.ws port="http://www.stockquote.org/StockQuoteService#
           wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
      </reference>
  </composite>
```

Figure 50. Exemple de composition avec SCA

[16] À noter que SCA propose une option pour inférer ces connexions. Cependant, une fois décidées, celles-ci ne peuvent pas changer.

Les dépendances de service sont décrites dans les types de composant. Ainsi, un type de composant doit décrire l'interface requise, la cardinalité, l'optionalité et peut définir des politiques concernant les services non-fonctionnels à assurer, ainsi qu'un identifiant unique (Figure 49). Une particularité de SCA réside dans l'interface de service. En effet, SCA supporte également les interfaces bidirectionnelles (Codépendance) et conversationnelles (conversation entre les consommateurs et les fournisseurs).

Une fois définie, la composition décrit les connecteurs reliant les composants entre eux en se basant sur les identifiants des références. Plus particulièrement, le composite définira le fournisseur du service et le type de liaison (Figure 50). Ainsi la composition déclarera le type de composant et associera les références avec soit un service fourni par une autre instance, soit avec un service importé (Figure 50). Dans tous les cas, la composition peut définir le type de liaison (par exemple service web, ou JMS …).

Une caractéristique intéressante de SCA c'est que la spécification ne définit rien sur les langages de programmation supportée. Les composants SCA peuvent être implémentés dans n'importe quels langages et paradigmes de programmation[17]. De plus, elle définit comment créer d'autres types de liaison. SCA supporte également la définition de politiques concernant la sécurité des accès aux données, le contexte transactionnel et la fiabilité de propagation de messages.

Bien que le langage de composition proposé par SCA soit extrêmement intéressant pour créer des applications à service, SCA ne spécifie rien sur le dynamisme. De plus, le fait de choisir les implémentations des types de composants ainsi que les liaisons entre composants au déploiement empêche toute substitution à l'exécution. Le tableau ci-dessous récapitule les caractéristiques de SCA par rapport aux critères énoncés.

Critères	
SOA sous-jacent	Non spécifié (dépend des types de liaison utilisés)
Description des dépendances de service	Optionalité, Cardinalité
Gestion du cycle de vie des services	Non spécifié (peut dépendre des implémentations)
Support de la composition	Composition structurelle
Infrastructure de supervision	Les instances sont configurées. La reconfiguration dépend des infrastructures d'exécution choisies
Gestion des politiques d'adaptation	Fixe
Mécanisme utilisé pour l'adaptation	L'injection dépend de l'implémentation utilisée. Cependant, l'adaptation est très limitée

Tableau 26. Caractérisation de SCA

3.3.6. Synthèse

Cette section a détaillé quelques-uns des modèles à composant à service les plus populaires. Ces modèles combinent les principes de l'approche à service et les modèles à composant. Ils permettent de développer des applications possédant un plus grand degré de *composabilité* que les applications développées avec des techniques habituelles.

Cependant, aucun d'entre eux ne fournit l'infrastructure nécessaire pour créer des applications dynamiques. En effet, la plupart de ces modèles se sont focalisés sur le modèle de développement et le masquage du dynamisme. Ils ne proposent pas de moyen pour créer des compositions flexibles. Lorsque la composition structurelle est supportée, celle-ci ne supporte pas le dynamisme.

En conclusion, les modèles à composant à service sont très prometteurs. Néanmoins, aujourd'hui, aucun ne fournit l'infrastructure complète pour être considéré comme une architecture à service dynamique étendue.

[17] SCA supporte également l'implémentation de type de composant basé sur une composition comportementale telle que BPEL.

4. Synthèse

Ce chapitre a présenté les concepts de l'approche à service. Cette approche propose de mettre en place des applications communiquant via des services. Le principal avantage de l'approche à service est qu'il permet un couplage-faible entre les différents composants formant l'application. L'approche à service propose également un motif d'interaction favorisant la liaison tardive. En effet, tout service doit être découvert avant d'être utilisé. Ces deux caractéristiques sont justement extrêmement importantes pour la gestion du dynamisme. Afin de supporter le dynamisme, le motif d'interaction proposé par l'approche à service doit être effectué durant l'exécution.

Papazoglou a défini le concept d'architecture à service étendue. Un SOA est un ensemble de technologies et de protocoles permettant la mise en place d'application suivant le paradigme de l'approche à service. Cependant, un SOA ne fournit pas les fonctionnalités nécessaires pour composer des services ainsi que de gérer ces applications. Un SOA étendu propose justement les fonctionnalités nécessaires pour créer, exécuter et gérer des applications à service. Ce chapitre a introduit une variation du SOA étendu afin de le personnaliser pour exécuter et gérer des applications dynamiques.

Néanmoins, développer des applications suivant le paradigme de l'approche à service dynamique tend à complexifier le code de l'application. En effet, en plus de la logique-métier de l'application, le développeur doit gérer la découverte et le dynamisme des fournisseurs de service. Les modèles à composant à service combinent les avantages des modèles à composant (séparation des préoccupations, composition structurelle) et de l'approche à service (faible-couplage, dynamisme). Ces nouveaux modèles à composant semblent proposer de bonnes bases pour développer des applications à service dynamiques. Cependant, ceux-ci ne proposent pas les fonctionnalités requises pour être un SOA étendu dynamique. En effet, le support de la composition est souvent absent ou très limité. De plus, les infrastructures de gestion sont la plupart du temps très limitées et ne prennent pas en compte le dynamisme.

La seconde partie de cette thèse s'intéresse à proposer un nouveau modèle à composant à service fournissant :

- Un modèle de développement simple
- Un modèle de composition structurelle supportant les trois types de dynamisme
- Une infrastructure extensible pour l'exécution, la supervision et la gestion d'application tout en supportant le dynamisme

Deuxième partie

Contribution

Chapitre 5
iPOJO:
Un modèle à composant à service

La première partie de ce manuscrit s'est intéressée à l'étude des approches permettant de créer des applications dynamiques. Après avoir détaillé et caractérisé les différentes approches pour créer des applications dynamiques, cette partie s'est particulièrement intéressée à l'approche à service. En effet, nous nous sommes aperçus que l'approche à service offre de nouveaux concepts particulièrement intéressants dans ce contexte.

La deuxième partie de cette thèse introduit un nouveau modèle à composant à service appelé iPOJO. Bien que ce modèle se base sur les récents efforts effectué dans le domaine[18] [173, 175, 177, 178], il apporte un modèle de développement beaucoup plus simple [176] et un langage de composition permettant la conception d'application dynamique. iPOJO est également doté de fonctionnalités d'introspection, de reconfiguration et d'extensibilité. Ainsi, iPOJO propose une infrastructure complète pour la conception, le développement et l'exécution d'application dynamique.

En effet, afin de créer des applications dynamiques, il faut être capable de développer des composants « atomiques » mais également de composer ces composants, de superviser les applications, ainsi que si nécessaire les reconfigurer. Parallèlement, le dynamisme n'est pas la seule contrainte non-fonctionnelle. Il faut donc supporter la gestion d'autres propriétés telles que la persistance, la sécurité ou la qualité de service. A l'inverse des autres approches se focalisant principalement sur le modèle de développement, iPOJO vise à proposer une infrastructure permettant de développer, de composer, d'administrer et d'exécuter des applications supportant tous les types de dynamisme.

Cette section rappellera les limites des approches actuelles pour créer des applications dynamiques et introduira les concepts et les exigences de l'architecture à service dynamique étendue. Ensuite la notion de modèle à composant à service sera rappelée ainsi que comment ceux-ci sont utilisés afin de définir une architecture à service dynamique étendue.

[18] iPOJO est d'ailleurs considéré comme la suite de Service Binder.

1. Approches pour la création d'applications dynamiques

La première partie de cette thèse a étudié les approches utilisées pour créer des applications dynamiques. Nous nous sommes aperçus que les approches actuelles sont limitées. En général, elles ne gèrent pas tous les types de dynamisme ou ne supportent pas la reconfiguration dynamique. Cette section rappelle les principales approches et montre les limites associées à ces approches.

1.1. Gestion du dynamisme dans le code

Une approche très populaire afin de créer des applications dynamiques consiste à gérer directement le dynamisme dans le code de l'application. Cette gestion a l'avantage d'être très flexible et permet la mise en place de mécanisme spécifique à l'application.

La gestion du dynamisme issue de l'environnement ou du contexte peut être assurée en implantant les reconfigurations dans le code de l'application. L'application requiert alors de connaitre les évènements déclenchant les adaptations et doit être capable de les collecter. Cette approche met en évidence un besoin récurrent dans les applications dynamiques : la nécessité de mettre en place une infrastructure pour la supervision de l'environnement ou du contexte. L'application peut alors ensuite recevoir (de manière asynchrone) les différents évènements et réagir en conséquence. Il s'agit typiquement de l'approche proposée par OSGi[42].

Faire évoluer une application est délicat à réaliser avec cette approche. En effet, l'application doit alors être capable de se mettre à jour toute seule et de reprendre son exécution après la mise à jour. Les applications à plug-in suivent généralement cette approche. L'application est capable de détecter les différents greffons à ajouter. Ceux-ci peuvent apparaître ou disparaître (être installé ou désinstallé). Cependant, faire évoluer l'application elle-même demande un redémarrage.

1.2. Modèle à composant & Reconfiguration dynamique

Une application dynamique est une application supportant la modification, c'est-à-dire la reconfiguration, de son architecture durant son exécution. La deuxième approche permettant de créer des applications dynamiques consiste à utiliser des modèles à composant supportant la reconfiguration dynamique. Les modèles à composant supportant la reconfiguration dynamique permettent justement cette manipulation. Ils simplifient grandement la gestion du dynamisme car abstraient les mécanismes sous-jacents nécessaires (gestion d'état, interception …).

Cependant bien que les modèles à composant soient aujourd'hui très répandus, ceux supportant la reconfiguration dynamique sont plus rares. Lorsque celle-ci est supportée, les adaptations sont souvent soit manuelles (exécuter par un administrateur) soit déléguées à un gestionnaire d'adaptation externe qui est chargé de la reconfiguration automatique de l'application. Ce gestionnaire d'adaptation doit alors connaitre et superviser l'architecture de l'application afin de la manipuler correctement.

L'évolution de composant interne à l'application est le type de dynamisme le plus souvent géré avec des modèles à composant. Une instance de composant est généralement remplacée par une instance de la même implémentation mais de version supérieure. Il est parfois nécessaire de changer l'implémentation afin de gérer un nouveau contexte.

1.3. Approche à service dynamique

Le chapitre précédent a également introduit l'approche à service dynamique. Grâce au style d'interaction proposé par cette approche (publication – découverte – liaison), les consommateurs et les

fournisseurs de services peuvent évoluer de manière séparée. Ceci permet de créer des applications dynamiques : un consommateur est notifié des fournisseurs disponibles et donc réagir en fonction de ces évènements.

L'approche à service propose ainsi la flexibilité manquante dans les modèles à composant traditionnels. Cependant, la mise en place d'application à service dynamique est très complexe. En effet, elle repose nécessairement sur une architecture qui alourdit considérablement le code de l'application.

Les modèles à composant à service combinent des concepts de la programmation par composant et de l'approche à service afin de simplifier le développement d'application à service dynamique. Les principes vont beaucoup plus loin et donnent les bases afin de permettre la composition structurelle de service. En effet, tel que défini dans [173], ces principes sont :

1. Un service est une fonctionnalité fournie
2. Un service est caractérisé par une spécification qui peut contenir des informations syntaxiques, comportementales et sémantiques ainsi que des dépendances sur d'autres spécifications de service
3. Un composant implémente des spécifications de service et peut dépendre d'autres services
4. Le style d'interaction de l'approche à service est utilisé pour résoudre les dépendances de service
5. Les compositions sont décrites en termes de spécifications de service
6. Les spécifications de service fournissent la base de la substitution

Le principe 5 propose de décrire les compositions en termes de spécifications de service et donc de permettre la substitution (dynamique) des fournisseurs de services ou des implémentations de service (principe 6). Cependant, afin d'être composable, une spécification de service doit décrire des dépendances. Ces dépendances ciblent d'autres spécifications de service (principe 2).

Le chapitre 4 a néanmoins montré que les modèles à composant actuels ne permettaient pas ou n'implémentaient pas tous ces principes. Ainsi, il n'est pas possible de concevoir des applications utilisant des services en les composants structurellement. Cette limitation provient de deux caractéristiques des SOA utilisées. Premièrement, ceux-ci ne permettent pas la description de dépendances de service dans les spécifications. Donc, il n'est pas possible de composer ces spécifications. Ensuite la notion d'implémentation de service n'est généralement pas supportée.

1.4. Problématique & Approche

Bien que les modèles à composant proposent des mécanismes de composition afin de concevoir des applications et ont tendance à proposer des modèles de développement simples, ils ne permettent pas la mise en place d'applications supportant tous les types de dynamisme. D'un autre côté, l'approche à service semble très prometteuse afin de créer des applications dynamiques. Cependant, celle-ci ne propose pas de manière de concevoir des applications *structurellement* ni d'administrer ces applications. Les récents efforts dans les modèles à composant a service ont tenté de rassembler les concepts de la programmation par composant et de l'approche à service. Cependant, ceux-ci ne proposent pas de modèle de composition ni de fonctionnalité de gestion. Lorsque la composition est supportée, celle-ci ne supporte pas le dynamisme.

La problématique de cette thèse est la construction d'applications dynamiques. Comme nous venons de le voir, les approches actuelles ne permettent pas de créer des applications dynamiques aisément ni de les administrer (Tableau 27).

Approche	Avantages	Limitations
Programmation par composant	Composition structurelle Reconfiguration dynamique (administration & manipulation) Modèle de développement	Manque de flexibilité Support du dynamisme issue de l'environnement et du contexte difficile
Approche à service dynamique	Faible couplage Liaison tardive Dynamisme	Pas de modèle de composition structurel Pas de fonctionnalité d'administration
Modèles à composant à service	Modèle de développement Dynamisme (dans le modèle de développement)	Pas de composition structurelle (ou non dynamique) Pas de fonctionnalité d'administration

Tableau 27. Avantages et limitations des approches actuelles pour créer des applications dynamiques

Cette thèse explore donc une nouvelle solution pour créer des applications dynamiques et les administrer. Nous proposons ainsi de combiner des concepts d'architecture logicielle, de la programmation par composant et de l'approche à service (au centre de la figure ci-dessous). Le but est de proposer un modèle de composant à service permettant de concevoir des applications dynamiques et de les administrer tout en proposant un modèle de développement simple pour le développeur et un modèle de composition intuitif pour l'assembleur. La gestion du dynamisme est alors automatisée à la fois dans le code du développeur et dans la gestion des compositions. Les applications conçues en suivant la méthode proposée supportent automatiquement les trois types de dynamisme introduits dans les chapitres précédents.

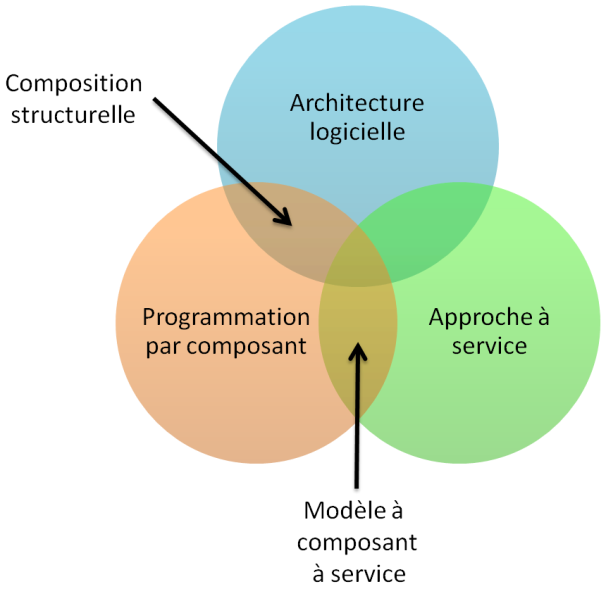


Figure 51. Positionnement de la thèse par rapport aux différents domaines concernés

2. iPOJO : un modèle à composant à service

Cette thèse propose un modèle à composant à service ayant pour but de proposer une infrastructure permettant le développement d'applications dynamiques en suivant les principes des SOA dynamiques étendus. Plus particulièrement, cette thèse combine des concepts d'architecture logicielle [69], de la programmation par composant [102] et de l'approche à service [137] afin de définir un modèle à composant permettant de concevoir, d'implémenter et d'exécuter des applications dynamiques telles que défini précédemment.

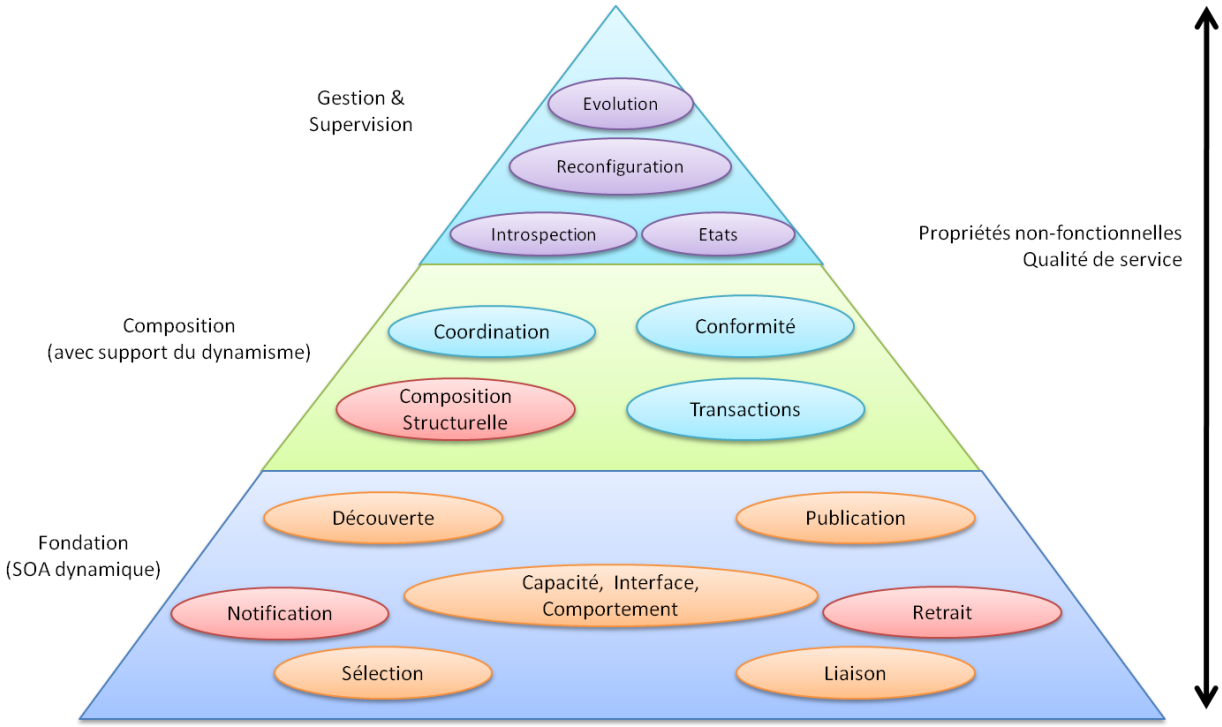


Figure 52. Principes des SOA dynamiques étendus

Ce modèle à composant doit fournir les fonctionnalités définis dans les architectures dynamiques étendues (rappelées dans la figure ci-dessus). Cette section décrit les exigences que doit remplir ce modèle à composant à service afin de fournir une infrastructure permettant la conception, l'implémentation, l'exécution et l'administration de systèmes dynamiques. En effet, tel que nous l'avons vu dans le chapitre précédent, il n'existe pas aujourd'hui d'infrastructure fournissant l'ensemble de ces fonctionnalités.

2.1. Un modèle et une machine d'exécution

La première exigence lors du développement d'iPOJO fut de concevoir à la fois un modèle et une machine d'exécution fortement liés. Ainsi, tous les éléments du modèle *existeront* lors de l'exécution. Ceci est primordial pour comprendre lors de l'exécution la structure de l'application. Ainsi, celle-ci sera décrite avec les concepts introduits dans le modèle.

Ce besoin a beaucoup contraint à la fois le modèle et la machine. En effet, les exigences étaient issues des deux parties : le modèle devait être simple tout en permettant l'expression du dynamisme, la machine d'exécution devait être capable d'exécution des applications utilisant le modèle défini et de gérer le dynamisme lors de ces exécutions.

2.2. Une architecture à service dynamique et isolation

Un modèle à composant à service repose nécessairement sur un SOA dynamique. En effet, les primitives liées à l'approche à service dynamique sont requises afin de gérer correctement le dynamisme. La machine d'exécution proposée repose sur OSGi™. En effet, la plate-forme de service OSGi™ propose toutes les primitives de l'approche à service dynamique nécessaire à la création d'application dynamique.

Cependant, dans les architectures à service traditionnelles, les services sont publiés de manière globale. Ainsi tous les consommateurs peuvent accéder à l'ensemble des services. Cette structure plate ne permet pas d'isoler certains services privés à application. Or, cette isolation est nécessaire afin de composer des applications telles que le proposent les mécanismes de composition des modèles à composant. L'architecture à

service dynamique sous-jacente doit donc permettre d'isoler certains services dans un annuaire de service attachée à une application. Bien qu'iPOJO repose sur OSGi™, il permet d'isoler certains services dans des *architectures à service* privées à une application. iPOJO reste néanmoins entièrement compatible avec OSGi™.

2.3. Un modèle de développement simple

Ensuite, le modèle à composant proposé fournit un modèle de développement simple. En effet, tout comme l'ont fait les précédents modèles à composant à service, il est nécessaire de masquer la complexité du SOA dynamique sous-jacent. iPOJO vise à proposer le modèle de développement *le plus simple possible* (basé sur l'idée de POJO [182]).

Celui-ci masque le dynamisme mais également les problèmes inhérents au dynamisme (gestion d'état, gestion des erreurs, synchronisation). Pour cela, la machine d'exécution d'iPOJO fournit une machine d'injection et de d'introspection permettant une gestion transparente de ces préoccupations pour le développeur d'application (Figure 53).

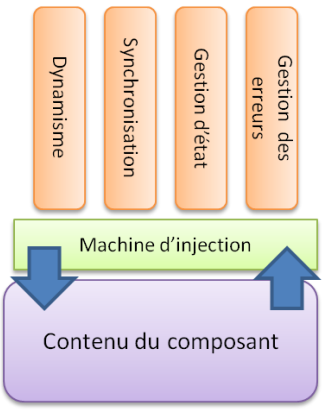


Figure 53. Utilisation d'une machine d'injection afin de gérer le dynamisme et les propriétés non-fonctionnelles liées

2.4. Un langage de composition structurelle

Tel que défini dans la pyramide du SOA dynamique étendu, iPOJO fournit un langage de composition structurelle. Afin de concevoir des applications dynamiques, iPOJO propose de composer les services structurellement. La principale différence par rapport aux langages de composition traditionnels vient du fait que la composition ne cible pas d'implémentation particulièrement, mais est exprimée en termes de spécifications de service. Ceci a l'avantage de découpler la composition des implémentations. A l'exécution l'infrastructure d'exécution choisira une implémentation disponible.

Les applications conçues ainsi sont gérées de manière dynamique. En effet, l'infrastructure d'exécution gère la disponibilité des implémentations de services, ainsi que les services importés et exportés dans et par la composition (Figure 54).

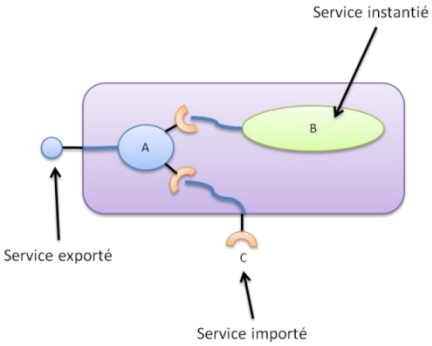


Figure 54. Un exemple de composition structurelle de service

En plus de fournir une machine d'exécution capable d'exécuter ces compositions, il est nécessaire que cette machine vérifie en permanence la conformité de la structure de la composition par rapport à la description faite dans l'architecture de cette composition. Cette vérification doit porter sur les services et instances internes à la composition, mais également sur les services importés et fournis. En effet, avec le dynamisme, les instances et services internes peuvent être remplacés « à la volée », les services importés peuvent évoluer en fonction de la disponibilité des fournisseurs et enfin, les services fournis ne doivent être publiés que si la composition est capable de fournir réellement le service.

2.5. Des fonctionnalités d'introspection et reconfiguration dynamique

iPOJO doit également proposer des mécanismes afin d'introspecter l'état du système et de le reconfigurer. En effet, dans le contexte des applications dynamiques il est crucial de pouvoir remonter l'architecture du système actuelle afin de comprendre les connexions, les services requis manquants… De plus, il est souvent nécessaire de reconfigurer une application ou une instance de composant.

La découverte et la visualisation de l'état du système permet de connaitre précisément les interconnexions et la structure des applications exécutées. Ainsi un administrateur peut à tout moment vérifier l'intégrité de son système, traquer les services requis manquant, etc.

iPOJO supporte également la reconfiguration dynamique. Chaque composition et composant atomique peuvent être reconfigurés. Cependant, celle-ci n'est pas exprimée de manière traditionnelle [61] mais introduit le concept de service dans ces reconfigurations. Ainsi, un administrateur ne manipulera pas directement les connexions ou les instances mais spécifiera les contraintes à appliquer sur ces dépendances de service. iPOJO modifiera alors les services utilisés afin d'être compatible avec les nouvelles contraintes.

Tout comme pour les modèles à composant supportant la reconfiguration dynamique, ces reconfigurations peuvent être effectuées par un administrateur humain ou par un gestionnaire d'adaptation externe. Dans ce cas, le gestionnaire ne modifie que les dépendances de services ce qui évite de manipuler directement l'architecture de l'application.

2.6. Des mécanismes d'extension

Le SOA dynamique étendu veut également gérer des propriétés non fonctionnelles autres que celles liées au dynamisme et à la configuration. Cependant, chaque domaine d'application a son propre ensemble de propriétés non-fonctionnelles. Afin de prendre en compte ces préoccupations, iPOJO propose des mécanismes permettant d'étendre le modèle de base. En effet, il permet d'introduire de nouveaux *handlers*[19] spécifiques à un besoin non-fonctionnel. Ce mécanisme d'extensibilité est disponible à la fois pour les composants atomiques et pour le langage de composition. Fournir un modèle à composant extensible n'est pas nouveau, cependant ces nouveaux traitants peuvent profiter des propriétés non-fonctionnelles déjà traitées et donc supporter le dynamisme. En effet, chaque nouveau traitant est un composant iPOJO et peut donc dépendre ou fournir des services. Le dynamisme de ces services sera géré par l'infrastructure d'exécution.

Ainsi le modèle à composant proposé peut être étendu avec de nouvelles propriétés non-fonctionnelles telles que la persistance, la sécurité, l'ordonnancement ou la qualité de service… Les traitants associés à ces préoccupations peuvent utiliser des services dynamiques gérés par iPOJO également[20].

[19] Dans cette thèse, nous nous permettons d'utiliser le vocable anglais généralement admis.

[20] En fait, toutes les propriétés non-fonctionnelles gérées par iPOJO suivent ce modèle, y compris la gestion du dynamisme.

3. Organisation de ce manuscrit

Le chapitre a rappelé les problèmes des approches actuelles permettant la mise en place d'application dynamique. Face à ces limites, nous proposons une nouvelle approche combinant des concepts d'architecture logicielle, des modèles à composant et de l'approche à service dynamique. iPOJO, le modèle à composant à service ainsi proposé, a l'avantage de proposer un modèle de développement simple ainsi qu'un modèle de composition intuitif. Le dynamisme est extrait de ces deux supports et sera géré par l'environnement d'exécution. Cette approche permet d'implanter les fonctionnalités des architectures à service étendues dynamiques.

La suite de ce manuscrit décrira en détails les différentes contributions de cette thèse, c'est-à-dire comment sont remplies les exigences citées précédemment.

Tout d'abord les principes d'iPOJO seront présentés dans le chapitre suivant. Cette description montrera spécifiquement les points remarquables différenciant iPOJO des approches actuelles pour créer des applications dynamiques.

Le chapitre suivant se consacrera au modèle de développement proposé et se focalisera plus particulièrement sur le support du dynamisme. Dans ce même chapitre sera présenté le langage de composition et comment ces compositions sont exécutées par la machine d'exécution. Ce chapitre décrira également comment le dynamisme est masqué dans le modèle de développement et comment celui-ci est géré dans les compositions.

Le chapitre 8 décrira les fonctionnalités d'introspection et de reconfiguration supportées par iPOJO. Les différences notoires par rapport aux principes de la reconfiguration dynamique seront mises en avant. Ce chapitre décrira également les mécanismes d'extensibilité proposés par iPOJO et comment il est possible d'introduire de nouveaux besoins non-fonctionnels sans pour autant nuire à la gestion du dynamisme.

Chapitre 6
Principes & Caractéristiques

Le chapitre 4 a montré que les modèles à composant actuels ne permettaient pas ou n'implémentaient pas tous les principes présentés dans [173]. Les problèmes de ces modèles sont divers. La plupart se sont focalisés sur la simplification du modèle de développement aux dépens des autres fonctionnalités requises telles que la composition ou l'administration. Lorsqu'un modèle de composition est proposé, celui-ci ne supporte pas le dynamisme.

L'approche choisie dans cette thèse consiste à proposer un nouveau modèle à composant à service combinant des concepts de l'approche à service, des modèles à composant et d'architecture logicielle. Le chapitre précédent a présenté les contraintes à remplir afin de proposer une infrastructure complète pour concevoir, implémenter, exécuter et administrer des applications dynamiques. Afin de remplir toutes ces exigences, il est nécessaire d'introduire un certain nombre de concepts clés dans le modèle à composant à service. Ce chapitre introduit ces concepts. Il décrit les concepts fondamentaux sur lesquels sont bâtis le modèle à composant et la machine d'exécution (Figure 55). Ce nouveau SOA dynamique se base sur OSGi™, mais l'étend afin de rajouter ces concepts[21].

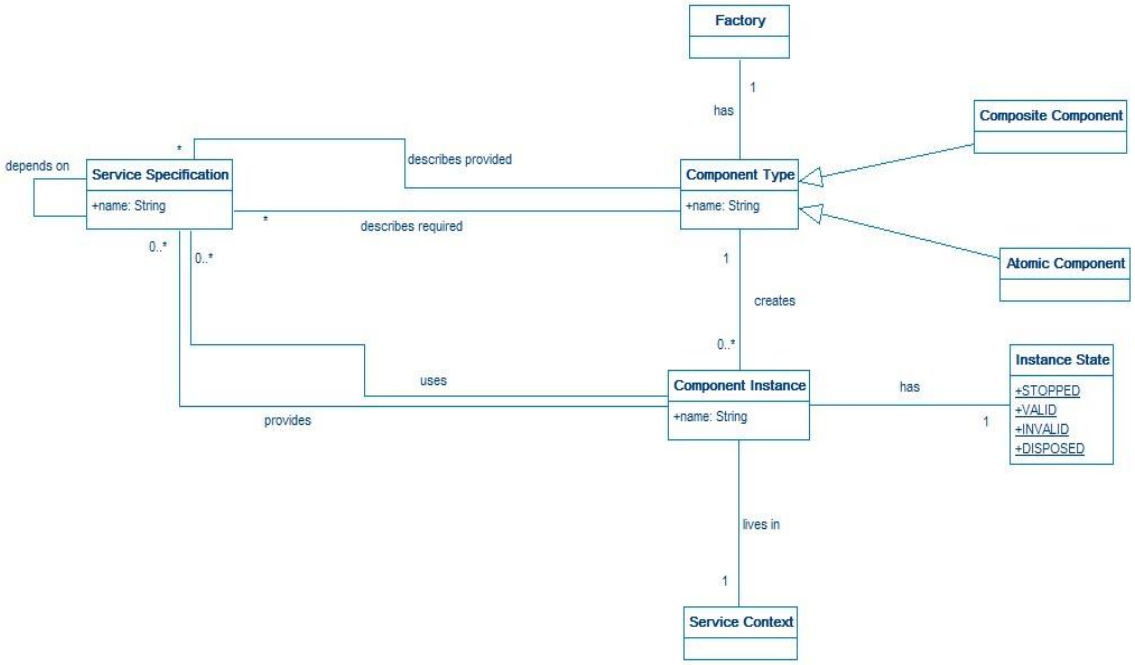


Figure 55. Modélisation des relations entre les spécifications de service, les types de composant et les instances de composant

[21] À noter que le modèle proposé est cependant totalement compatible avec la spécification OSGi.

1. Types de composants et Instances

Tout d'abord, dans le modèle proposé, les fournisseurs et consommateurs de services sont implémentés sous la forme de ***types de composants***. Ces types de composants décrivent les spécifications de service requises et fournies, ainsi que différentes informations sur le contenu du composant et sur les aspects non fonctionnels telles que l'état, ou la configuration. Un type de composant décrivant des services fournis est également appelé ***implémentation de service***.

Les types de composants du modèle à composant à service proposés se divisent en deux classes :

- Les types de composants ***atomiques***
- Les types de composants ***composites***

Le contenu d'un composant composite sera composé d'autres instances et de services. Les types de composants composites permettent de structurer une application en plusieurs niveaux. En effet, un composite peut également contenir d'autres composites, ce qui rend le modèle hiérarchique.

A chaque type de composant (quelle que soit sa classe) est attachée une ***fabrique***. Une fabrique permet la création d'***instance de composant***. Bien évidemment, le type de composant d'une instance est le type associé à la fabrique ayant servi à la création de l'instance. Un point important d'iPOJO est que ces fabriques sont elles-mêmes des services. De cette manière il sera possible de questionner le registre de service afin de découvrir dynamiquement les implémentations de service disponibles.

Chaque instance possède une configuration qui lui est propre. Cette configuration est donnée à la fabrique lors de la demande de création de l'instance. Ainsi, des instances créées avec la même fabrique (et donc, qui ont le même type de composant) peuvent néanmoins se comporter différemment. Les instances interagissent à l'exécution en suivant les principes de l'approche à service. Les instances publient les services fournis. Ensuite, les instances requérant des services recherchent les fournisseurs disponibles fournissant ce service. Une fois trouvés, les instances peuvent l'utiliser. L'application du paradigme de l'approche à service permet d'avoir des liaisons faiblement couplées entre les instances. En effet, la seule donnée commune est la ***spécification de service***.

Ainsi, seules les instances fournissent et requièrent réellement les services décrits dans le type de composant. Comme l'illustre la figure ci-dessous, un type de composant décrit les services fournis et les dépendances de services. Ces descriptions spécifient la manière dont doivent être gérés ces aspects à l'exécution et plus particulièrement les spécifications de services requises et fournies. L'instance de composant *possède* ces ***dépendances de service*** et ces ***services fournis***. D'ailleurs, ces deux entités régiront le comportement de l'instance face au dynamisme. Les instances d'une implémentation de service sont également appelées ***instances de service,*** car deviennent des fournisseurs de ce service.

Remarquons que les dépendances de service et les services fournis sont associés à un ***contexte de service***. Un contexte de service peut être vu comme un annuaire de service à part entière. Chaque instance vit dans un contexte de service qui lui donne accès au service requis et dans lequel elle publie ses propres services. De plus, un contexte de service fournit les primitives nécessaire au dynamisme. Cette notion de contexte de service permet d'isoler certains services tout en respectant les principes de l'approche à service. Ce concept primordial dans iPOJO sera présenté ultérieurement dans ce chapitre.

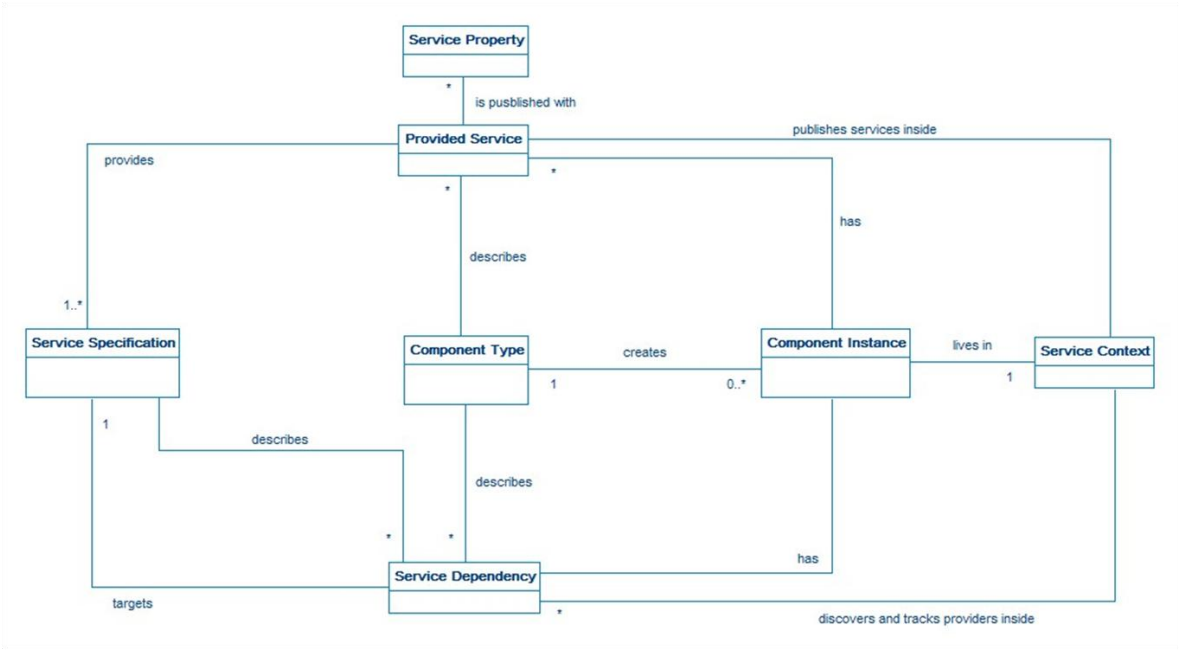


Figure 56. Relation entre type de composant, instance, dépendances de services et services fournis

Une fois créée, une instance peut être valide ou invalide (Figure 57). Une instance est ***valide*** lorsque toutes ses dépendances de service sont satisfaites. Une instance valide publie et fournit ses services. Une instance ***invalide*** n'est pas utilisable. De plus, elle n'est pas découvrable, car les services fournis sont retirés de l'annuaire de service. En fonction de l'arrivée et du départ de fournisseurs de service, l'état d'une instance peut osciller entre invalide et valide.

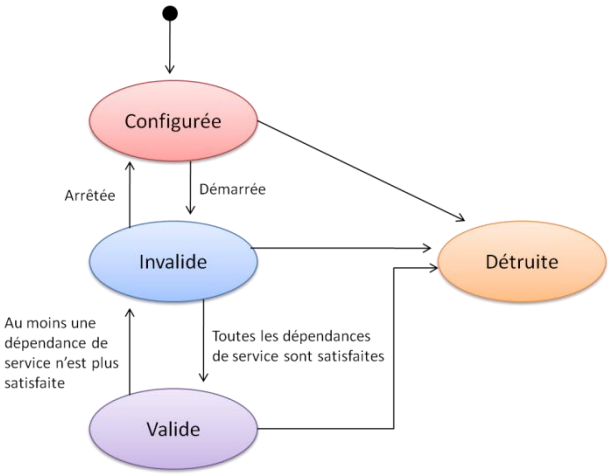


Figure 57. Cycle de vie des instances de composant

2. Spécification de service & Composition

Afin d'implanter la composition structurelle et le support du dynamisme, il est nécessaire que les spécifications de service expriment des dépendances sur d'autres spécifications. En effet, des spécifications sans dépendances ne sont pas composables, puisqu'elles ne décrivent pas leurs requis [183]. iPOJO propose de décrire trois informations dans les spécifications de service (Figure 58) ayant pour objectif de rendre composable les spécifications de services:

- Des dépendances sur d'autres spécifications
- Un modèle de l'état du service
- Des propriétés de service

Ces trois informations sont ajoutées à l'interface de service contenant la description syntaxique (liste des opérations disponibles) du service[22]. Cette section décrit la nécessité de ces trois informations et introduit la notation qui sera utilisée dans cette thèse.

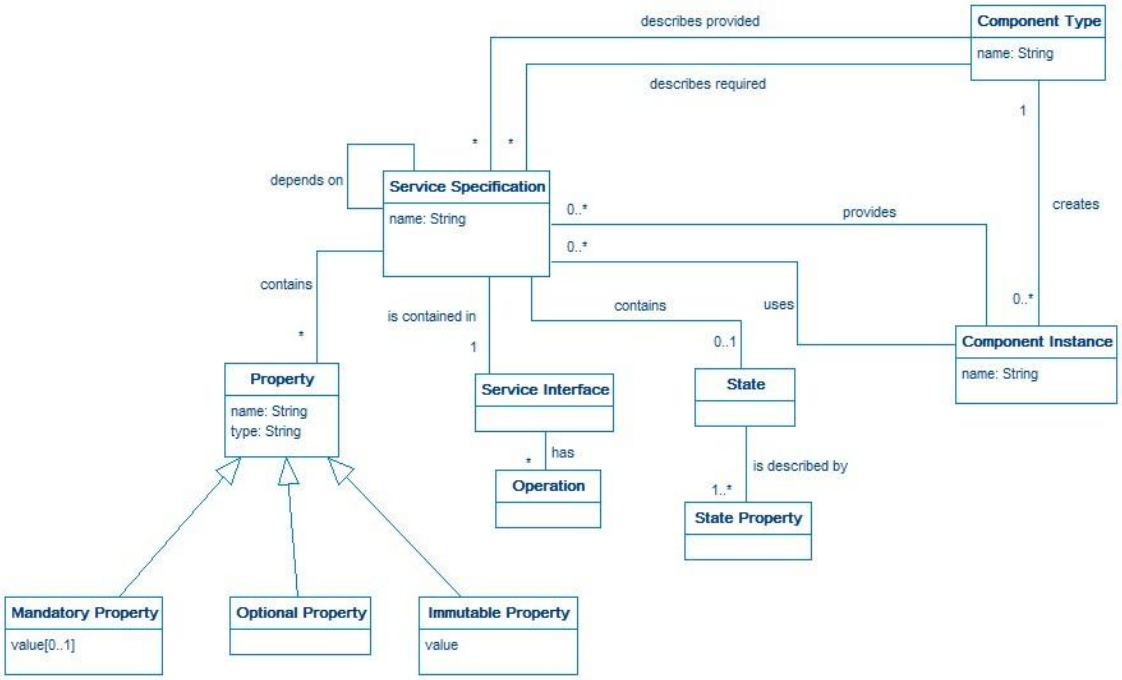


Figure 58. Modèle de spécification de service

2.1. Dépendance de service

Une spécification de service doit décrire des *dépendances* afin d'être composable. Ainsi, il sera possible à la fois de construire une application avec ce service, mais également de substituer une implémentation de ce service par une autre. Ces dépendances de niveau spécification sont indispensables afin de permettre la composition structurelle de service. Cette thèse utilisera la notation ci-dessous afin de représenter une spécification de service et une dépendance de service.

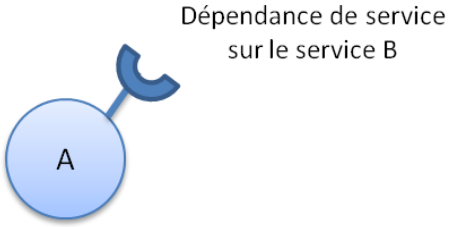


Figure 59. Exemple de spécification de service
avec une dépendance de niveau spécification

```
public interface A {

    public static final String specification="specification { " +
                    "requires { " +
                        "$specification=\"org.apache.felix.ipojo.service.B \" " +
                    "}
            }";

    public void doSomething();
}
```

Figure 60. Description d'une dépendance de service dans la spécification de service

[22] OSGi utilise les interfaces Java comme spécification de service. Donc, iPOJO repose sur le même système.

Comme l'illustre la figure ci-dessus, cette description est insérée dans l'interface de service. Ainsi, une application peut être composée en assemblant des spécifications de services. Grâce aux dépendances de services, l'assembleur peut vérifier la cohérence de la composition (Figure 61).

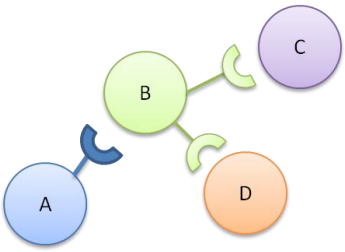


Figure 61. Un exemple de composition structurelle de service cohérente

Cependant, introduire ce niveau de dépendance ne supprime pas les dépendances de niveau implémentation. Les composants implémentant un service dépendent des services décrits dans la spécification, mais peuvent également dépendre de services ou de fonctions propres à cette implémentation (Figure 62). Ce double niveau de dépendance n'existe généralement pas dans les modèles à composant traditionnels.

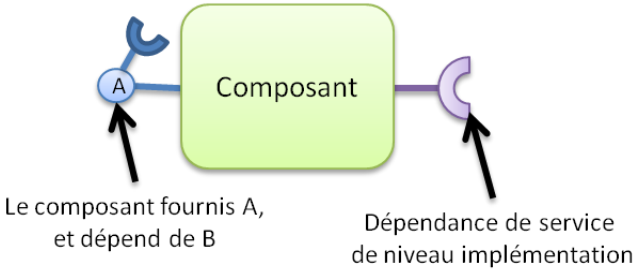


Figure 62. Distinction entre les dépendances de niveau spécification et de niveau implémentation

Par exemple, imaginons un composant nécessitant un service de tri et fournissant un service permettant d'afficher une liste de données. Dans les modèles à composant traditionnels, il n'est pas possible de spécifier si le service de tri est là pour tirer les données affichées ou s'il s'agit d'un détail d'implémentation. Avec les dépendances de niveau spécification, cette distinction est possible.

2.2. Modèle de l'état

Une deuxième information nécessaire afin de gérer le dynamisme est la description du modèle de l'état. Ainsi, un service peut posséder un état. Toutes implémentations de ce service doivent se conformer à ce modèle. Ce modèle de l'état est crucial lorsqu'une implémentation est remplacée par une autre, tel que le propose le dernier principe des modèles à composant à service, sans pour autant perdre l'état de ce service. Cependant, cette gestion est minimale et n'apporte qu'une solution limitée au problème du transfert d'état.

Grâce à cette description (Figure 63), lorsqu'une implémentation de service est remplacée, l'état de l'instance de service peut être récupéré et ensuite réinjecté dans la nouvelle instance. Cette récupération et réinjection est également envisageable lorsqu'une implémentation disparaît et qu'une autre devient disponible plus tard. Donc, en ajoutant le modèle de l'état dans la spécification de service, il est possible de gérer des *sessions* malgré le dynamisme des services, ainsi que de rendre des services ou une composition persistant (sans néanmoins dépendre d'une implémentation de ce service en particulier).

```
public interface A {

    public static final String specification="specification { " +
                    "requires { " +
                        "$specification=\"org.apache.felix.ipojo.service.B\" " +
                     "}" +
                     "state { " +
                         "property { " +
                             "$name=\"property1\" " + "$type=\"java.lang.String\" " +
                         "} " +
                    "} " +
          "}";

          public void doSomething();
}
```

Figure 63. Un exemple de spécification de service décrivant son état

Cependant, bien que la notion de spécification de service dans les spécifications de service soit indispensable à la composition de service, l'ajout d'un modèle de l'état ne l'est pas. Lorsque celui-ci est absent, les instances de service sont considérées sans état. De plus, ce mécanisme est très restreint. Le transfert d'état repose également sur les implémentations qui doivent supporter l'injection et la sauvegarde de l'état.

2.3. Propriétés de service

iPOJO propose de spécifier des propriétés de service dans la spécification de service. Tout fournisseur de ce service devra publier ces propriétés afin d'être valide. Ces propriétés servent généralement à caractériser le service avec des propriétés standards. Elles peuvent être des propriétés du modèle d'état ou non. Par exemple, pour un service représentant un dispositif physique, la spécification peut contenir la propriété *device.location* indiquant la localisation du dispositif. Ces propriétés sont utilisées par l'assembleur et le développeur afin d'exprimer des contraintes sur les services requis.

```
public interface A {

    public static final String specification="specification { " +
                    "requires { " +
                        "$specification=\"org.apache.felix.ipojo.service.B\" " +
                     "}" +
                     "property { " +
                       "$name=\"property.name\" " + "$type=\"java.lang.String\" " +
                    "} " +
                     "property { " +
                       "$name=\"property2.name\" " + "$type=\"java.lang.String\" " +
                              "$optional=\"true\" " +
                    "} " +
                     "property { " +
                       "$name=\"property3.name\" " + "$type=\"java.lang.String\" " +
                              "$value=\"1.0.0\" " + "$immutable=\"true\" " +
                    "} " +
          "}";

          public void doSomething();
}
```

Figure 64. Exemple de propriétés de service décrites dans la spécification de service

Trois types de propriétés sont supportés. Les propriétés obligatoires sont nécessairement publiées par les fournisseurs implémentant le service. À l'inverse les propriétés optionnelles ne sont pas forcément publiées par les fournisseurs. Enfin, les propriétés immuables sont des propriétés obligatoirement publiées, mais dont la valeur est spécifiée dans la spécification et ne peuvent pas être changées par l'implémentation. Il peut s'agir par exemple de la version de la spécification publiée (Figure 64).

3. Un modèle de dépendance riche et flexible

Un point important caractérisant les modèles à composant à service est la description des dépendances de service[23]. iPOJO propose un modèle de dépendance de service à la fois riche et flexible commun aux deux niveaux. Ce modèle permet de décrire les dépendances de service et définit également la politique de gestion de dépendance gérée à l'exécution. En fonction de cette requête, décrite dans le type de composant, la machine d'exécution d'iPOJO découvre, traque et injecte les fournisseurs de service disponible dans l'instance de composant. Nous verrons dans les chapitres suivants comment celui-ci est étendu dans certains cas afin de remplir des exigences spécifiques, telles que la simplicité du modèle de développement.

Toute dépendance de service cible une spécification de service. De plus, à l'exécution, celle-ci peut être satisfaite ou non (Figure 65). Une dépendance de service est satisfaite dès lors qu'un fournisseur de service compatible avec la requête est disponible et accessible. Une dépendance peut également être cassée si une contrainte cruciale a été violée. Une dépendance cassée ne peut plus être satisfaite même après l'arrivée de nouveaux fournisseurs.

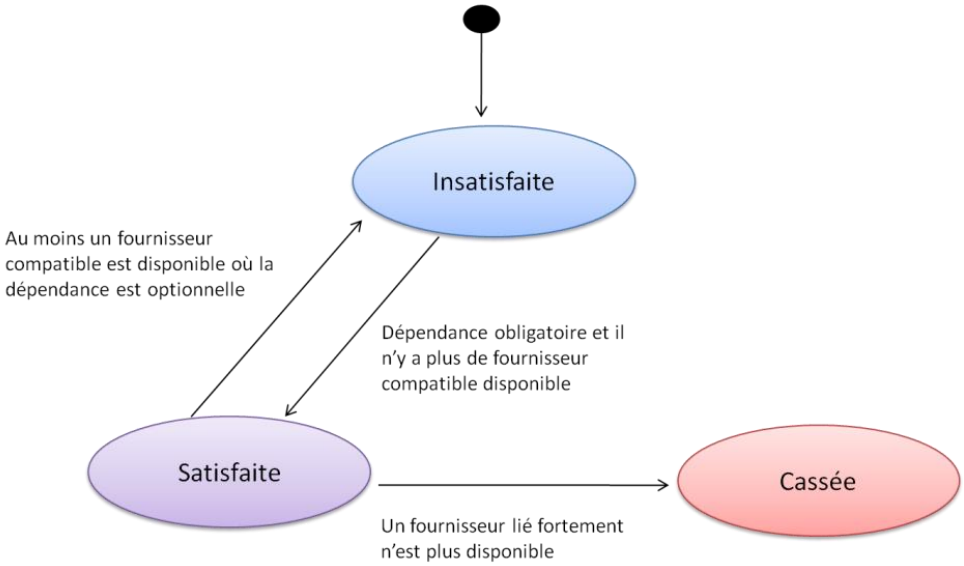


Figure 65. Cycle de vie des dépendances de service

Les dépendances de services peuvent être (Figure 66):

- Simples ou agrégées
- Obligatoires ou optionnelles
- Filtrées
- Triées
- Dynamiques, statiques ou à priorités dynamiques

Une dépendance de service peut utiliser n'importe quelle combinaison de ces attributs. Cette section décrit chacun de ces attributs et comment ils sont traités à l'exécution[24].

[23] Plus particulièrement, dans iPOJO ces dépendances de service interviennent à deux niveaux : au niveau des spécifications de service et au niveau des implémentations.

[24] Une dépendance de service possède également un id permettant de faire correspondre les dépendances de service de niveau spécification et de niveau implémentation. Cette relation sera décrite ultérieurement dans ce chapitre.

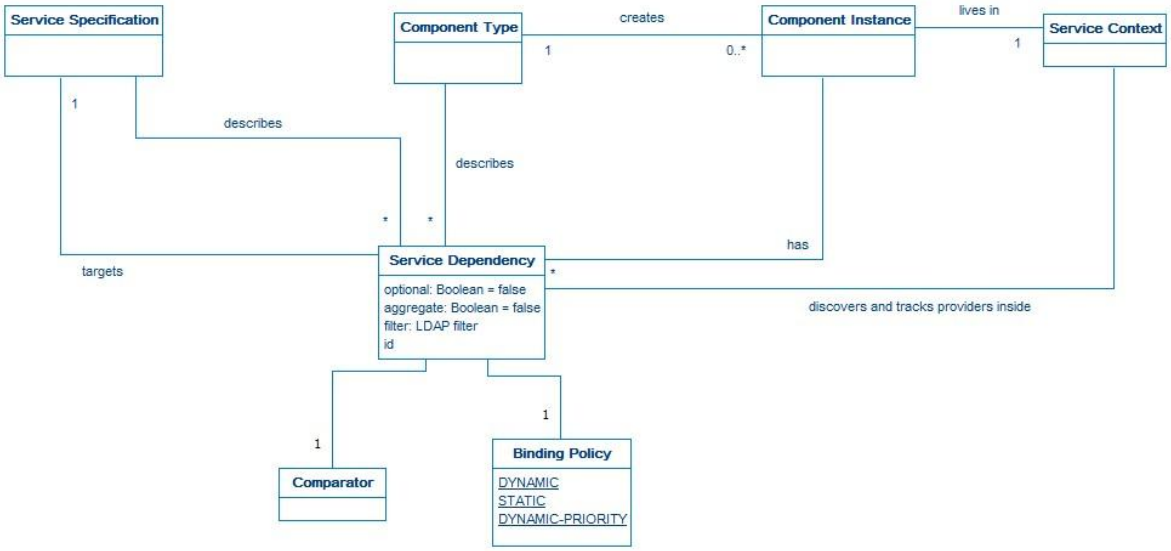


Figure 66. Modèle de dépendance de service

3.1. Simple ou agrégée

Une dépendance de service simple ne traque qu'un seul fournisseur. En revanche, une dépendance de service agrégée découvre et traque l'ensemble des fournisseurs disponibles (Figure 67). Cependant, dès la disponibilité d'un fournisseur, une dépendance de service multiple est satisfaite.

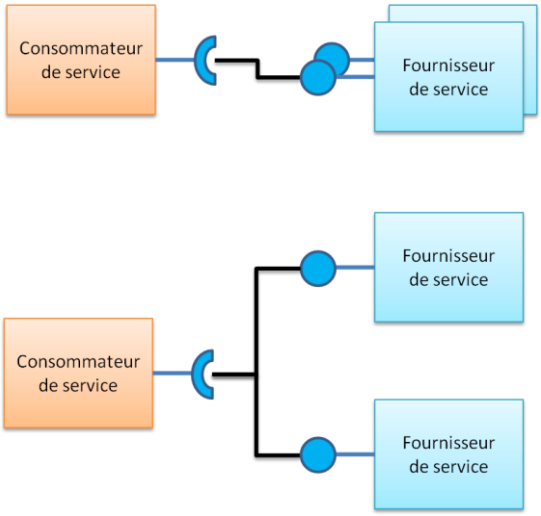


Figure 67. Dépendance de service simple (en haut) et dépendance de service agrégée (en bas)

3.2. Optionalité

Une dépendance optionnelle reste satisfaite bien qu'aucun fournisseur ne soit disponible. À l'inverse, une dépendance obligatoire nécessite la présence d'au moins un fournisseur compatible et accessible afin d'être satisfaite.

Cet attribut est très important car il permet d'insérer des points de variation dans une composition de service ou dans une implémentation. Par exemple, cet attribut permet entre autres de décrire un mode de fonctionnement dégradé pouvant être amélioré par la disponibilité de certains services.

3.3. Filtrage

Bien qu'une dépendance de service cible une spécification de service en particulier, celle-ci peut également spécifier des contraintes sur les fournisseurs. L'écriture de ces contraintes (exprimée en LDAP) est facilitée par la présence des propriétés dans la spécification de service. Cependant, l'implémentation peut choisir d'ajouter des propriétés lors de la publication. L'assembleur et le développeur peuvent utiliser les propriétés de service décrites dans la spécification de service afin d'exprimer leurs filtres.

Le filtre d'une dépendance est donc une contrainte sur les fournisseurs de service. Afin d'être utilisés, ces fournisseurs doivent être compatibles avec ce filtre. Grâce à ces filtres, il est possible d'exprimer un large spectre de contrainte. Par exemple, il peut servir à cibler une implémentation particulière ou à imposer un vendeur particulier.

Cependant, dans le cadre de propriété dynamique (que les fournisseurs peuvent changer durant leur exécution), il est nécessaire de réévaluer le filtre à chaque changement afin de déterminer la compatibilité des fournisseurs. De plus, ce filtre peut contenir des variables substituées à l'exécution. La valeur de ces variables pouvant également changer (par exemple suite à un changement de contexte), l'ensemble des fournisseurs utilisé sera recalculé.

3.4. Ordre, Tri et Sélection

Lorsque plusieurs fournisseurs de service sont disponibles, il est souvent nécessaire de choisir le fournisseur le plus adéquat parmi l'ensemble disponible. Ce choix est souvent appelé *service ranking*. Afin de supporter cette sélection, iPOJO introduit la notion de comparateur dans l'expression des dépendances de service. Ce comparateur est utilisé pour trier les fournisseurs compatibles avec la requête dans le but de sélectionner le plus adéquat. Dans le cas de dépendances agrégées, le comparateur sert à trier cet ensemble qui sera ainsi traité dans l'ordre (du meilleur au moins bon).

Le tri par défaut prévoit de favoriser les fournisseurs présents depuis le plus longtemps.

3.5. Politique de liaison

La politique de liaison permet de spécifier la réaction d'une dépendance de service face au dynamisme. iPOJO propose trois politiques de liaison : dynamique, statique et à priorité dynamique.

La politique de liaison dynamique suit le dynamisme des fournisseurs. Lorsqu'un fournisseur est disponible, il est utilisé tant que celui-ci ne disparaît pas. Si celui-ci néanmoins disparaît, un autre fournisseur est sélectionné et utilisé. Il s'agit de la politique par défaut.

La politique de liaison statique limite le dynamisme. Lorsqu'un fournisseur est utilisé, si celui-ci disparaît, la dépendance est cassée. L'arrivée d'autres fournisseurs n'a pas d'effet sur la dépendance, celle-ci doit être réinitialisée afin de redevenir valide. Cette politique supporte un dynamisme limité. Tant que le service requis n'est pas utilisé, elle suit l'arrivée et le départ des fournisseurs. Du moment qu'un fournisseur est utilisé, la dépendance est gelée (liaison forte) et ne suit plus l'arrivée d'autres fournisseurs. Cette politique de liaison sert généralement lorsque la connexion avec le fournisseur service possède un état difficile à maintenir (session).

La troisième politique appelée à priorité dynamique est une extension de la politique dynamique. Cependant, à chaque arrivée ou modification de service, le fournisseur utilisé est réévalué en suivant le comparateur spécifié. Ceci permet de garantir à un consommateur d'avoir tout le temps le « meilleur » fournisseur disponible. Lorsque le fournisseur disparaît, le deuxième fournisseur du classement est utilisé.

4. Contexte de service & SOA hiérarchique

Une classe particulière de modèles à composant supporte la composition hiérarchique. Un composant peut contenir d'autres composants et ainsi de suite. Le principal avantage de ce type de composition est qu'il permet d'abstraire la complexité des composants utilisés. Le fait qu'un composant soit un composite est totalement masqué à l'exécution.

Une spécificité de la composition hiérarchique (également appelé composition verticale) est que l'instance de composant situé dans un composite n'est pas accessible à l'extérieur de ce composite (sauf si ce modèle propose des composants partagés [123]). Cette isolation est souvent bénéfique, car permet à chaque composite de contrôler l'accès aux instances internes (Figure 68). C'est au composite de garantir cette isolation lorsque les connecteurs sont créés.

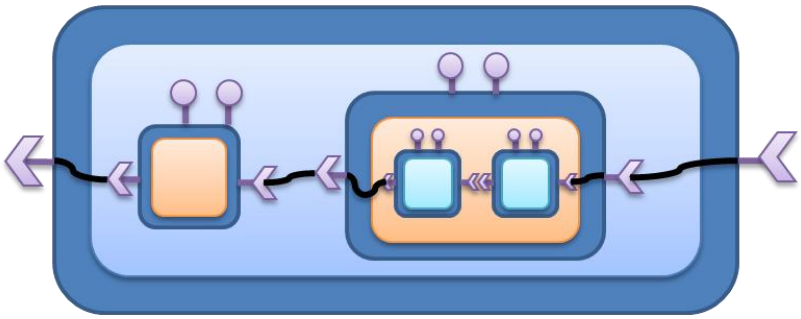


Figure 68. Un exemple de composition hiérarchique avec des composants

L'approche à service ne définit pas de principe d'isolation. En effet, un service peut être découvert, mais rien n'est dit sur la *découvrabilité* d'un service. Les principes des modèles à composant à service ne spécifient pas non plus de notion d'isolation. Cependant, il semble intéressant de combiner cette isolation dans la composition structurelle de service. Ainsi, chaque composant peut lui-même être une composition qui possède son propre lot d'instances de service isolé. L'assembleur pourra vérifier qui a accès aux services utilisés au sein de sa composition. Ceci a également l'avantage de proposer une alternative à la vision *à plat* de l'approche à service qui a tendance à créer des architectures *spaghettis* difficilement maintenables.

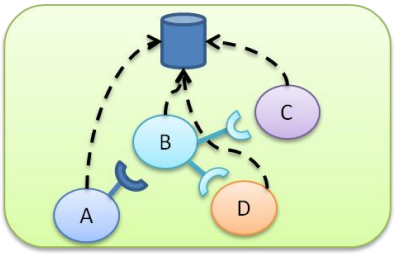


Figure 69. Chaque composite possède son propre courtier contenant les services fournis à l'intérieur du composite

Afin d'obtenir cette propriété d'isolation, il faut jouer sur la notion de découverte. En effet, un service est découvrable s'il est publié dans le courtier de service. Donc afin de garantir l'isolation, il faut que chaque composite possède son propre courtier. Ainsi, les services exposés dans un composite ne sont accessibles qu'au consommateur ayant accès à ce courtier (c'est-à-dire les consommateurs présents également dans le composite).

En rajoutant cette isolation, nous transformons le SOA utilisé par le modèle à composant à service, en un SOA hiérarchique où chaque composite est un SOA à part entière (possédant son propre courtier, des fournisseurs et des consommateurs). Ce SOA hiérarchique a la forme d'un arbre. Le SOA racine est le SOA global (Figure 70). Dans le cas d'iPOJO, celui-ci sera le SOA proposé par OSGi.

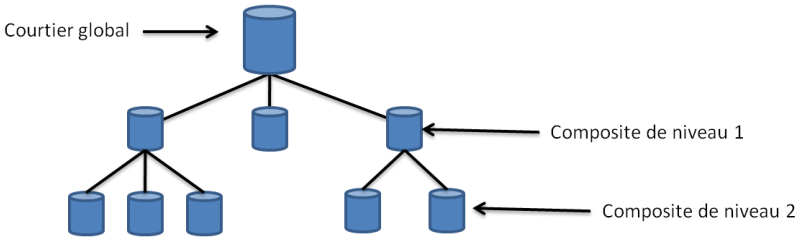


Figure 70. Structuration en arbre d'un SOA hiérarchique

Afin de permettre cette isolation et cette hiérarchie, nous définissons la notion de ***contexte de service*** comme un SOA particulier. Chaque instance de composant vit dans un contexte de service. Cette instance n'a accès qu'à ce contexte. L'instance publie et recherche ses services à l'intérieur de ce contexte. Afin de garantir l'isolation des services dans une composition, chaque composite contient donc son propre contexte de service (et vit également dans un autre contexte de service (parent)). Toutes les entités présentes dans le composite ont accès à ce contexte, c'est-à-dire à l'annuaire local et dépendent des primitives fournies par ce contexte afin de publier et de découvrir des services. Ces instances n'ont pas accès aux autres contextes de service. Chaque composition a accès à deux contextes de service : le contexte du composite parent (ou le contexte global) et son contexte interne. Cependant il n'a pas accès à ses contextes fils (c'est-à-dire créés par des composites instanciés et exécutés dans son contexte interne). Nous verrons ultérieurement comment un composite peut créer des interactions entre son contexte interne et son contexte parent.

5. Cohérence entre dépendances de spécification et d'implémentation

Tel qu'il a été introduit auparavant dans ce chapitre, iPOJO possède deux niveaux de dépendance. Le premier niveau est situé dans les spécifications de service elles-mêmes. Le second niveau est situé dans les implémentations. Ces deux niveaux décrivent des dépendances de service en utilisant le modèle de dépendance de service introduit dans ce chapitre.

Une implémentation d'un service doit respecter les dépendances de service des spécifications qu'elle fournit. Cependant, ce *respect* n'est pas une reproduction à l'identique des dépendances des spécifications. En effet, afin d'être plus flexibles, les dépendances de spécification contraignent les dépendances d'implémentation (Tableau 28).

Dépendance de spécification	Contraintes sur les dépendances dans l'implémentation
Dépendance obligatoire	Dépendance obligatoire
Dépendance optionnelle	Ignore la dépendance, dépendance optionnelle ou obligatoire
Dépendance agrégée	Dépendance simple ou agrégée
Dépendance filtrée	Dépendance filtrée avec un filtre égal ou plus contraint
Comparateur	Même comparateur
Politique de liaison statique	Politique de liaison statique
Politique de liaison dynamique	Politique de liaison dynamique, statique ou à priorité dynamique
Politique de liaison à priorité dynamique	Politique de liaison à priorité dynamique

Tableau 28. Contraintes sur les dépendances d'implémentation par rapport aux dépendances de spécification

Cependant, une implémentation peut rajouter des dépendances n'étant pas des dépendances de spécification. Dans ce cas la politique de résolution est différente. Dans le contexte d'une dépendance de spécification, cette dépendance est résolue dans le contexte de service de l'implémentation. En effet, comme les compositions s'exécutent au sein d'un contexte de service et que ces compositions sont exprimées grâce à

des services composables, l'assembleur peut garantir la disponibilité des services requis par les dépendances d'implémentation dans le contexte.

Cependant, lorsque la dépendance n'est pas une dépendance de spécification, il n'y a aucun moyen de garantir cette disponibilité. C'est pourquoi ces dépendances d'implémentation sont résolues différemment. Cette résolution s'effectue en deux étapes :

- Une recherche de fournisseur est effectuée
- Une recherche d'implémentation est ensuite effectuée

Tout d'abord, un fournisseur est cherché dans le contexte global[25]. Cette solution est généralement utilisée pour des services communs tels que des services de journalisation. Cependant, si aucun fournisseur n'est trouvé, une implémentation de ce service est recherchée. Si une implémentation est disponible, une instance de service est créée dans un contexte de service interne au contexte de service dans lequel l'instance requérant ce service vit. Ce contexte n'est accessible que par cette instance, ce qui garantit que seule cette instance a accès au service.

6. Synthèse

Ce chapitre a présenté les différents points clés introduits dans iPOJO afin de proposer une infrastructure « complète » pour créer des applications dynamiques. Ce modèle diffère des modèles à composant à service traditionnels. Tout d'abord, iPOJO propose un SOA sous-jacent hiérarchique et dynamique. Bien que celui-ci se base sur OSGi™, il est étendu afin de supporter la composition structurelle de service. Plusieurs informations nécessaires sont présentes dans la spécification de services telle que les dépendances de service. De plus, iPOJO généralise l'utilisation de types de composant et de fabriques. Ceci apporte la notion d'implémentation de service et d'instance de service également nécessaire au support du dynamisme. Il est ainsi possible de substituer une implémentation de service par un autre fournissant le même service. Enfin, iPOJO dispose d'un modèle de dépendance riche permettant des politiques de liaisons sophistiquées.

Critères	Valeurs
Langage utilisé pour les spécifications de service	Interface Java étendue
Information contenue	Interface (informations syntaxiques), dépendance de service et propriétés d'état et de publication
***Forme* du courtier**	Annuaire de service hiérarchique (contexte de service)
Localisation du courtier	Centralisé et Multiple
Implantation de la publication et du retrait de service	Publication et retrait automatisé par le gestionnaire d'instance
Type de découvertes supportées	Active et Passive
Langage de requête et filtrage	LDAP
Implantation de la notification	Évènement envoyé aux consommateurs par notification
Protocole de liaison	Java (référence directe)
Type de composition supporté	Structurelle
Infrastructure proposée pour exécuter les compositions	Support et exécution des compositions structurelles de service dynamiques

Tableau 29. Caractérisation du SOA proposé par iPOJO

[25] Il est également possible de spécifier le contexte de résolution si celui-ci ne doit pas être le contexte global. Ainsi, le fournisseur peut être trouvé dans le contexte de service courant.

Donc, iPOJO fournit un SOA caractérisable par les mêmes critères que ceux proposés dans le chapitre 4. Le chapitre suivant détaillera les deux classes de type de composants supportés. Tout d'abord seront présenté les composants atomiques ainsi que le modèle de développement proposé. Ensuite le support de la composition structurelle au sein de composant composite sera étudié. Enfin le chapitre 8 présentera les fonctionnalités introduites dans iPOJO afin de gérer la reconfiguration dynamique, l'extensibilité et l'introspection. Toutes ces fonctionnalités sont basées sur les principes et les concepts introduits dans ce chapitre.

Chapitre 7
Gestion du dynamisme dans les composants atomiques et composites

Créer des applications dynamiques avec des composants à service est un processus en deux étapes. La première étape est la conception de l'application à l'aide d'un langage de description d'architecture. Lors de l'exécution de cette application, la machine d'exécution prendra en charge le dynamisme. Dans cette thèse, nous proposons que le langage de description d'architecture possède la notion de service comme une entité de premier ordre. Cependant, la composition d'application bien qu'indispensable, n'est pas suffisante. Il faut également créer des implémentations *concrètes* de service. Ces composants atomiques sont développés en utilisant un langage de programmation standard tel que Java, un langage de script tel que Groovy ou un langage de composition comportementale tel que BPEL.

Une des idées clés des modèles à composant à service est de séparer la gestion du dynamisme de la logique-métier du composant (et donc des services fournis). Les dépendances de service ainsi que les services fournis sont alors décrits de façon déclarative dans le descripteur du composant. La grande majorité des modèles à composant à service tendent à simplifier le développement de composants atomiques. Cependant, beaucoup de taches sont encore laissées aux développeurs, telle que la *protection* de l'état interne, la synchronisation[26] ainsi que la sauvegarde et la restauration de l'état lors d'une évolution. En effet, le développeur doit généralement implémenter des méthodes de liaison (callbacks) et donc gérer les appels concurrents de ces méthodes. Ce modèle de programmation donne généralement des protocoles de synchronisation complexes et pouvant amener à des *deadlocks* ou *livelocks* [184] en plus d'introduire du code gérant le dynamisme dans la logique du composant. Externaliser la gestion du dynamisme suppose donc d'éviter ce type de modèle de développement intrusif et dangereux tout en donnant une plus grande liberté pour le développeur. En effet , il est apparu nécessaire de simplifier encore plus le modèle de développement et de permettre aux développeurs d'agir sur le cycle de vie de l'instance sans pour autant introduire du code spécifique au modèle ou du code gérant le dynamisme [185].

Dans le cadre d'une composition, celle-ci doit permettre de concevoir des applications tout en masquant le dynamisme. Le modèle de composition doit donc proposer des abstractions permettant l'assemblage d'application de manière hiérarchique de manière intuitive. La gestion de ces compositions devra gérer lors de l'exécution le choix des différents *composants* internes, garantir la cohérence de l'assemblage et supporter les différents types de dynamisme. Ainsi les applications conçues avec ce modèle de composition supporteront automatiquement le dynamisme. Il en résulte un assemblage dynamique des différents *composants* à l'intérieur d'un composite.

Ce chapitre décrit les requis d'un modèle de développement masquant le dynamisme ainsi que sa mise en place. Il sera également montrer l'impact de cette infrastructure sur des composants implémentés en Java. Ensuite, le modèle de composition proposé par iPOJO sera introduit. Les caractéristiques essentielles à la gestion du dynamisme seront présentées ainsi que la gestion de ces compositions à l'exécution.

[26] La gestion de la synchronisation est crucial afin d'attendre un état quiescent lors des reconfigurations dynamiques.

1. Objectifs d'un modèle de développement supportant le dynamisme

Afin de supporter le dynamisme de manière transparente, le modèle de développement des composants atomiques doit remplir certaines conditions. Cette section décrit les objectifs d'un tel modèle de développement.

1.1. Un modèle simple et non intrusif mansquant le dynamisme

Le premier objectif d'un modèle de développement supportant le dynamisme est qu'il soit simple. Cette simplicité implique que le modèle impose un minimum d'intrusion dans la logique-métier du composant.

Aujourd'hui cette tendance est illustrée par l'apparition de nombreux modèles à composant proposant des modèles de développement POJO. Un code suivant l'approche POJO ne contient que la logique-métier. Ce code n'est pas « pollué » par du code provenant de l'infrastructure d'exécution ou par du code nécessaire à des besoins non-fonctionnels (vis-à-vis de la logique-métier). Le code est alors géré à l'exécution en suivant les motifs de conception d'inversion de contrôle et d'injection de dépendance. La mise en place de tels modèles de développement est désormais possible grâce au progrès des infrastructures d'injection et d'interception (comme AspectJ [186]). Ainsi, le développeur doit pouvoir se concentrer seulement sur l'implémentation de la logique-métier du composant.

Cette simplicité s'étend également à la description des types de composant. Celle-ci doit être également la plus simple et intuitive possible tout en évitant les informations redondantes. Il est crucial que ce modèle de développement masque également le dynamisme. En effet, normalement à chaque fois que le développeur veut accéder à une ressource (service dans le contexte des modèles à composant à service) potentiellement dynamique, il doit vérifier sa disponibilité. Mais, ce test ne garantit en rien la disponibilité de cette ressource au moment de son utilisation. En effet, celle-ci peut disparaître entre le test et l'utilisation réelle (Figure 71).

```
if (myprovider != null) {
          // The provider can potentially leaves
          // here, and so the following invocation
          // will fail.
          myprovider.doSomething();
}
```

Figure 71. Exemple de code ne garantissant pas la disponibilité du fournisseur

Le code du développeur ne devrait pas contenir ce genre de test. L'infrastructure sous-jacente doit donc gérer et garantir la disponibilité des ressources requises au moment de l'utilisation. De plus la disparition d'une ressource utile peut entrainer une modification de l'état. Donc cette infrastructure doit également garantir l'intégrité de l'état lors des reconfigurations (et plus particulièrement lors des évolutions) sans pour autant imposer un traitement dans le code du développeur. Ceci entraine inévitablement le masquage des protocoles de synchronisation nécessaires ainsi que l'automatisation de la sauvegarde et de la restauration de l'état.

Néanmoins, dans certains cas, il est nécessaire que le code puisse réagir au dynamisme et particulièrement aux arrivées et aux départs des ressources requises. Il faut donc offrir la possibilité au développeur de gérer une partie du dynamisme.

1.2. Gestion des primitives de l'approche à service dynamique

Dans le cadre d'un modèle à composant à service, il est important que le modèle de développement ne contienne pas de code interagissant avec l'annuaire de service lors de la mise en place d'une liaison.

Cependant, il doit être possible aux développeurs d'exprimer des dépendances de service, d'utiliser des services requis, de publier et de fournir des services.

Donc bien que masqué, le code du développeur doit pouvoir interagir avec les services requis (les utiliser). Il est également nécessaire d'offrir la possibilité au développeur de modifier les propriétés de publication. Il est aussi parfois fondamental pour le code de pouvoir agir sur la publication ou le retrait des services fournis. Bien que ceci soit contraire au motif de conception d'inversion de contrôle, l'ajout de cette fonctionnalité offre une grande liberté aux développeurs.

2. Modèle des types de composants atomiques

Les contraintes citées dans la section précédente influencent directement la description de type de composants atomiques. Afin de masquer le dynamisme et les primitives de l'approche à service tout en gardant une certaine flexibilité, la description du type de composant doit refléter ces fonctionnalités (Figure 72). iPOJO propose une approche déclarative. Le code du composant (implémentation) et la description de la politique de gestion du dynamisme sont séparés. Lors de l'exécution, les instances seront gérées en fonction de cette description.

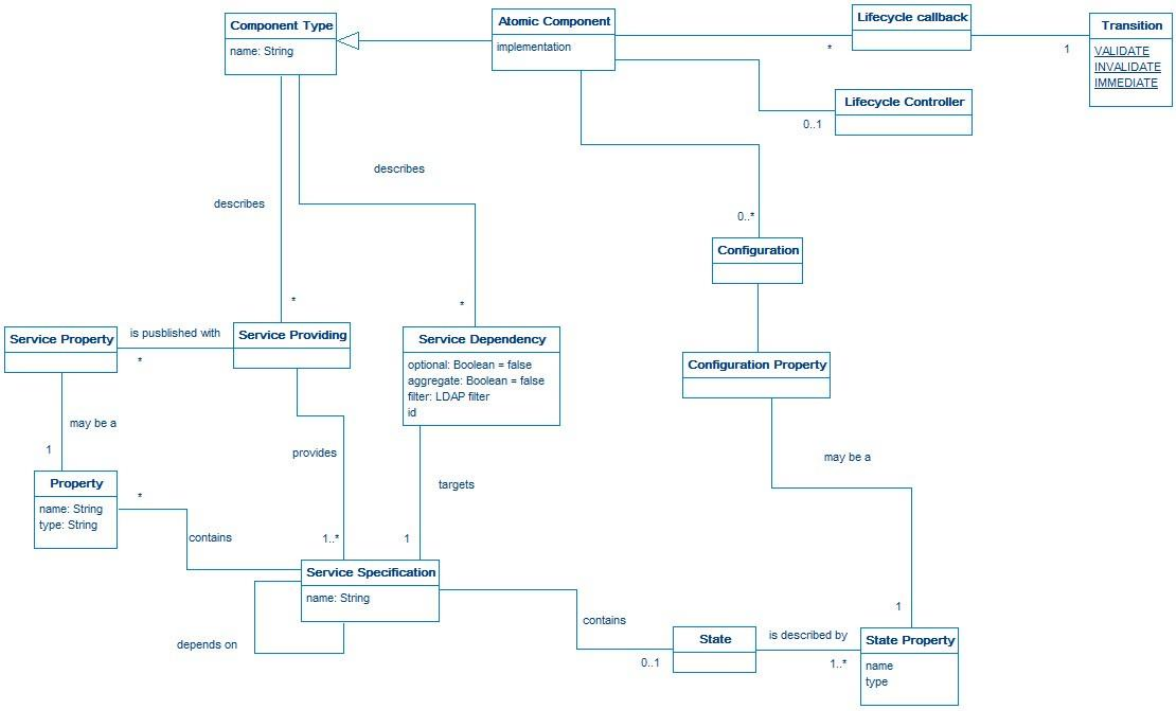


Figure 72. Elément décrivant les types de composants atomiques

Tout d'abord, un type de composant atomique a une implémentation. A ce niveau d'abstraction, les détails sur cette implémentation ne sont pas nécessaires. Les types de composants atomiques sont des types de composants et donc décrivent des dépendances de service ainsi que les services fournis. Cependant en plus de ces deux fonctionnalités, les composants atomiques peuvent exprimer une configuration, un contrôleur de cycle de vie ainsi que des fonctions d'activation et de désactivation.

La configuration d'un composant atomique décrit comment les instances peuvent être configurées et comment est composé l'état de l'instance. Ces propriétés peuvent être des propriétés de l'état des spécifications de services implémentés. Toutes propriétés d'état contenues dans les spécifications de services implémentés doivent être attachées à des propriétés de configuration. Dans le cas contraire, le type de composant n'est pas considéré comme une implémentation valide du service.

Le contrôleur de cycle de vie est un moyen pour le développeur d'influencer la publication de service. En effet, dans iPOJO, seules les instances valides publient leurs services. Ce contrôleur permet donc au code de participer au cycle de vie. Cependant, cet impact est limité. Le contrôleur ne peut pas valider une instance si des dépendances de service obligatoires ne sont pas satisfaites. Néanmoins, il peut demander l'invalidation de l'instance même si toutes les dépendances de services sont résolues.

Un composant atomique peut également avoir des fonctions d'activation (validation) ou de désactivation (invalidation). Ces méthodes servent à initialiser l'instance ainsi que de permettre aux développeurs de l'arrêter proprement. Ces fonctions ne sont pas contraires à l'approche POJO. En effet, la validation peut être vue comme l'appel au constructeur ou à une méthode d'initialisation. La désactivation peut être vue comme un appel à un destructeur[27]. iPOJO propose également une transition particulière (immédiate) afin d'initialiser le contenu d'une instance dès sa validation.

3. Types de composants atomiques pour Java

La section précédente a décrit les éléments constituant d'un type de composant atomique. Ce modèle peut être projeté sur plusieurs langages d'implémentation. Cette section décrit comment ce modèle peut être projeté sur Java et plus particulièrement montre les liens entre les éléments du modèle et une classe Java[28]. Ce modèle est le modèle idéal qui pourrait être proposé pour Java. Ensuite, les mécanismes d'injection et d'interception nécessaire afin de permettre la gestion de composant Java seront présentés. Cette section se terminera en donnant quelques exemples d'utilisation afin de montrer le modèle de développement « idéal » proposé et le confrontera aux contraintes exposées dans la première section de ce chapitre.

3.1. Type de composant atomique implémenté en Java

La projection du modèle des types de composant atomique sur Java spécialise chacun des concepts de ce modèle pour le langage de programmation Java. Par exemple, l'attribut implémentation du modèle devient le nom de la classe Java. La plupart des autres concepts interagissent avec les éléments de cette classe Java tels que les méthodes, les membres (appelés également champs ou attributs) et les interfaces implémentées.

Ainsi les méthodes d'activation et de désactivation sont des méthodes de la classe d'implémentation. Le contrôleur de cycle de vie devient un membre booléen de la classe. Lorsque celui-ci devient faux, l'instance est invalidée. Si celui-ci devient vrai, alors le développeur donne son accord pour la validation de l'instance.

Les propriétés de configuration sont attachées soit à des membres de la classe, soit à des méthodes, soit aux deux. Une propriété attachée à un membre est supervisée par le conteneur de l'instance qui peut donc gérer la sauvegarde et la restauration d'état. Une propriété attachée à une méthode permet au code d'être notifié quand la valeur de cette propriété est modifiée (lors du démarrage ou de la restauration de l'état). Le modèle de développement ressemble alors à celui proposé par les JavaBeans.

Les dépendances de service peuvent également être attachées à un membre de la classe d'implémentation. Ainsi le développeur accèdera au service en utilisant ce membre comme n'importe quel autre. Cependant, bien que cette méthode masque intégralement le dynamisme, il est parfois nécessaire au code de gérer une partie du dynamisme. iPOJO propose donc également la possibilité d'attacher une dépendance de service à deux méthodes de la classe d'implémentation. Ces méthodes seront appelées

[27] Java ne supporte pas les destructeurs. Cependant, de nombreux objets possèdent des méthodes d'arrêt tel que *close* pour les flux, *shutdown* pour les exécuteurs ou *dispose* pour les objets Swing/AWT

[28] Bien que cette projection soit spécifique à Java, elle est applicable aux autres langages de programmation par objet.

lorsqu'un fournisseur de service compatible arrive ou disparaît[29]. Ainsi, le code peut réagir au dynamisme des services requis.

Lorsqu'une dépendance de service est optionnelle, il est généralement indispensable de tester la disponibilité de ce service avant de l'utiliser. Cependant, tel qu'expliqué précédemment, ceci est à la fois contraignant n'apporte pas de réelle garantie. iPOJO implante le motif de conception *Null Object* [187] afin d'éviter ce test. Ainsi, le développeur n'aura pas à effectuer ce test et pourra directement utiliser le membre attaché à la dépendance de service. S'il n'existe pas de fournisseur disponible alors le développeur utilisera de manière transparente une *fausse* implémentation du service n'ayant aucun effet. Ce motif de conception est également étendu avec la notion d'implémentation par défaut. Le développeur peut alors choisir d'utiliser une implémentation du service qu'il fournit lorsqu'aucun fournisseur compatible n'est disponible.

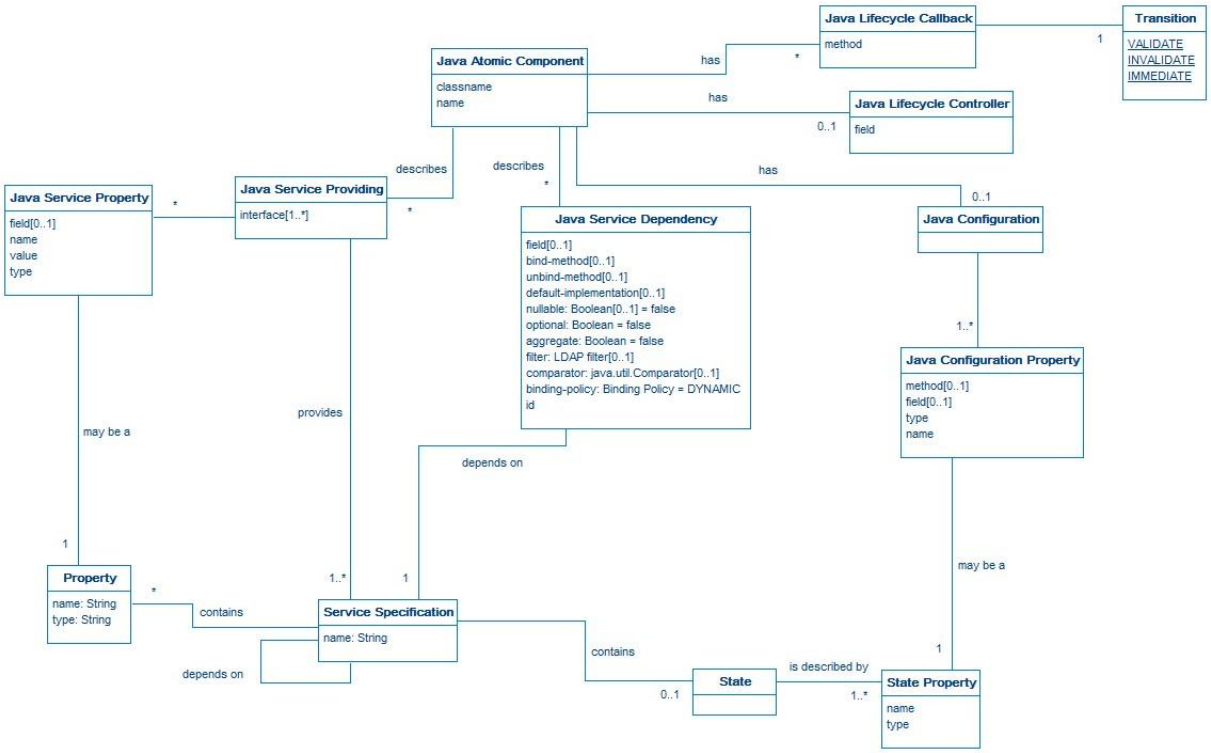


Figure 73. Modèle des types de composants atomiques implémentés en Java

Afin de fournir des services, la classe d'implémentation du composant doit impérativement implémenter les interfaces de ces services afin de garantir que toutes les opérations spécifiées soient implémentées. Le code peut manipuler certaines propriétés de service en les attachant à un membre de la classe. Chaque modification de ces membres mettra à jour la publication de service.

3.2. Une machine d'injection et d'interception

La section précédente a présenté le modèle de type de composant atomique pour Java. Ce modèle repose sur la liaison entre la description du type de composant et la classe d'implémentation de ce composant. Afin de permettre ces liaisons, il est nécessaire d'exécuter les objets de cette classe d'implémentation au dessus d'une machine d'injection et d'interception qui permettra au conteneur de superviser l'exécution de ces objets (Figure 74). Cette machine permet d'attacher les objets au conteneur les supervisant. Plusieurs implémentations de cette machine sont envisageables. Cette section adresse seulement les fonctionnalités requise et ne décrit pas une implémentation particulière.

[29] Dans le cas de dépendance de service simple, ces méthodes seront appelées seulement pour le fournisseur sélectionné.

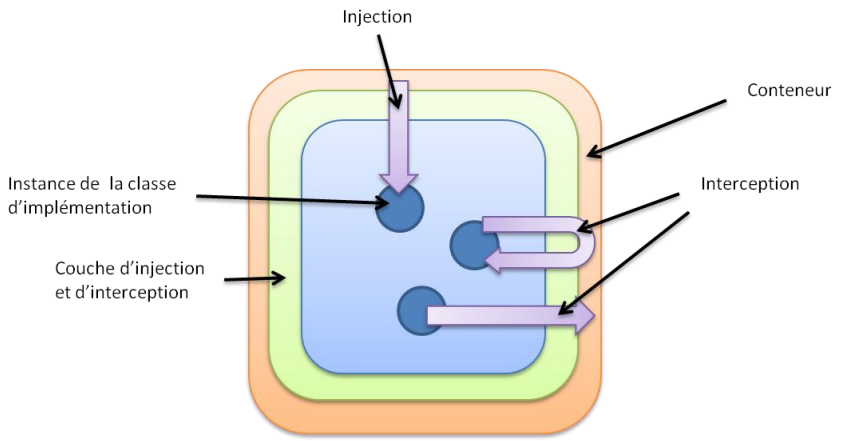


Figure 74. Principe d'une machine d'injection et d'interception

Afin de permettre cette supervision tout en proposant un modèle de développement le plus simple possible, cette machine doit fournir les capacités suivantes :

- Permettre l'appel de méthodes sur un objet par le conteneur
- Permettre la création de nouvelles instances de la classe d'implémentation
- Permettre d'injecter une valeur dans un membre d'un objet
- Permettre la notification lorsque la valeur d'un membre est modifiée
- Intercepter les appels de méthodes ainsi que les sorties et erreurs potentielles

La première capacité sert principalement à appeler les méthodes spécifiées dans la description du type de composant lorsque cela est nécessaire (liaison, activation, désactivation …). La deuxième capacité permet au conteneur de superviser la création des instances (c'est-à-dire les objets) de la classe d'implémentation. Plus particulièrement, ceci permet de retarder au maximum et de gérer de manière optimale ces créations. Ces deux capacités sont communes dans les machines d'injection et reposent sur les mécanismes de réflexion offerts par Java.

L'injection de valeur dans un membre est utile pour injecter un objet de service (objet fourni par le fournisseur de service servant véritablement ce service) ou pour restaurer un état. La notification, lorsqu' un membre est assigné, est nécessaire à la supervision des propriétés de service attachées à ce membre, au suivi de l'état ainsi qu'au contrôleur de cycle de vie. Enfin, l'interception des appels de méthodes ainsi que des sorties et des erreurs est crucial pour garantir l'état de l'objet et gérer la quiescence. Ces capacités seraient implantables en utilisant les propriétés de réflexion de Java telles que les Proxies[116]. Cependant, une telle implantation poserait des problèmes de performance et serait limitée.

Afin de lever ces limitations, la machine d'injection et d'interception d'iPOJO procède en deux étapes :

- Tout d'abord, la classe d'implémentation est manipulée lors de l'empaquetage du composant (en tant que *bundle* OSGi™). La classe résultante permet l'injection de valeur dans des champs ainsi que l'interception de méthode et la notification lorsqu'un membre est assigné.
- Ensuite cette classe est utilisée à l'exécution à la place de la classe d'origine.

La combinaison de la réflexion (pour les deux premières capacités) et de la manipulation implantent toutes les capacités requises de la machine d'injection et d'interception. La manipulation modifie la classe d'implémentation afin d'encapsuler toutes les accès aux membres et toutes les méthodes par des appels de méthodes délégués au conteneur qui peut alors superviser intégralement ce qui se passe dans les objets (Figure 75 et Figure 76). De plus, lors de la manipulation, certaines informations sur la classe sont collectées. Ces informations sont alors utilisées à l'exécution afin de vérifier la cohérence du type de composant et de la

classe d'implémentation (méthodes et membres absents), ainsi que pour vérifier que le type de composant est bien une implémentation valide des services fournis.

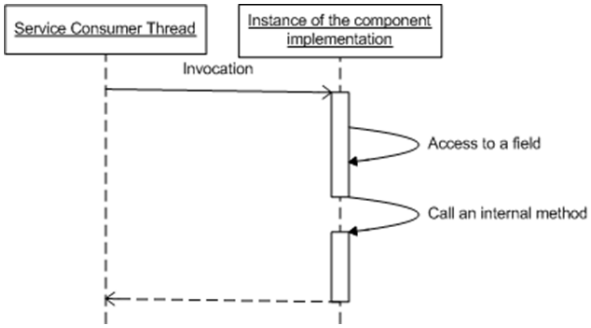


Figure 75. Séquence d'appels sans injection ni interception

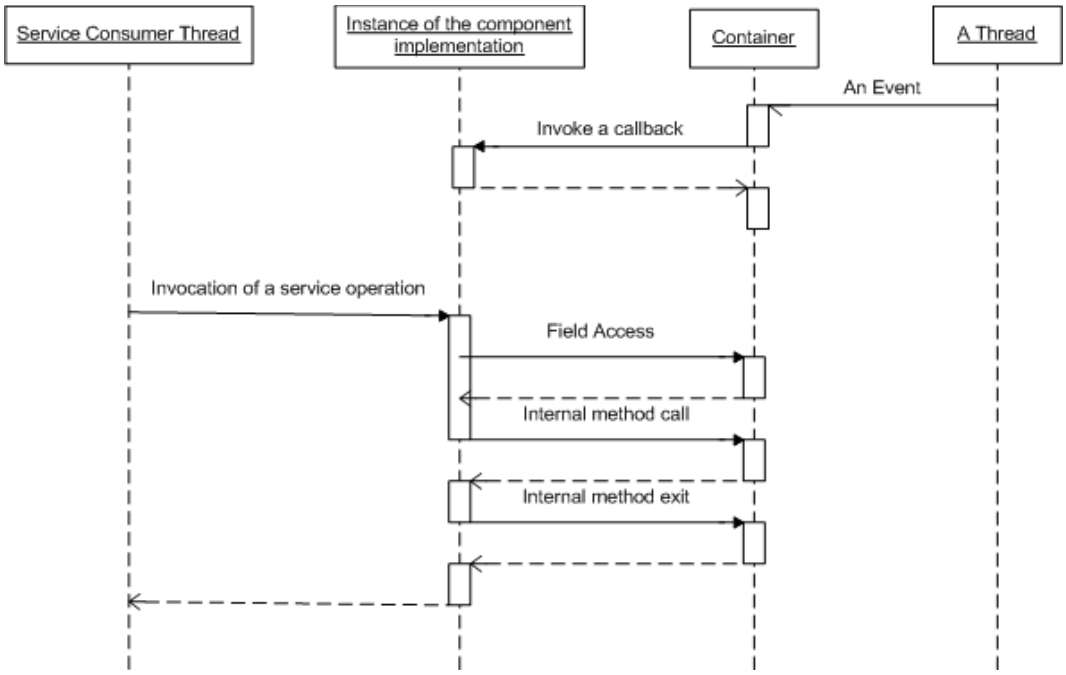


Figure 76. Séquence d'appels avec injections et interceptions

3.3. Démonstration du modèle de développement pour Java

Cette section a pour but de montrer le modèle de développement proposé et de le confronter avec les requis décrits précédemment dans ce chapitre. Plusieurs exemples sont présentés afin d'illustrer les différentes capacités du modèle.

3.3.1. Dépendances de service

Une classe Java contient à la fois des membres (parfois appelés champs) et des méthodes. Afin d'utiliser un service, iPOJO propose plusieurs alternatives. Tout d'abord, ce service peut être injecté dans un membre de la classe d'implémentation du composant. Dans ce cas, le membre de cette classe reçoit un objet de service. Cette solution a l'avantage de proposer un modèle de développement très simple garantissant la gestion du dynamisme (Figure 77). En effet, la valeur assignée dans le membre suit le dynamisme des arrivées et des départs des fournisseurs. Cette méthode permet également d'éviter les problèmes de synchronisation qui peuvent être gérés par iPOJO pour éviter par exemple qu'un service disparaisse avant la fin de la méthode ou que le code utilise différents fournisseurs au sein de la même méthode (ou du même flot d'appels). La description du composant n'a pas à redéfinir la spécification de service requise. En effet, celle-ci est collectée lors de la manipulation.

```
public class HelloConsumer {
    private Hello m_hello;
    public void doSomething() {
        System.out.println(m_hello.getMesage()); // Use the service as a regular field
    }
}

<component classname="...HelloConsumer">
    <requires field="m_hello"/>
</component>
<instance component="...HelloConsumer"/>
```

Figure 77. Dépendance de service simple

Lorsqu'une dépendance de service est optionnelle, afin d'éviter les tests superflus testant la disponibilité du service, iPOJO injecte un objet factice. Cette méthode simplifie le modèle de développement lorsque des dépendances sont optionnelles (Figure 78).

```
public class HelloConsumer {
         private Hello m_hello;
         public void doSomething() {
              System.out.println(m_hello.getMesage());   // Print nothing when the
                                                         // service is not available
          }
}

<component classname="...HelloConsumer">
    <requires field="m_hello" optional="true"/>
</component>
<instance component="...HelloConsumer"/>
```

Figure 78. Dépendance de service simple et optionnelle

Une seconde alternative permet aux développeurs d'être notifiés d'une partie du dynamisme. L'implémentation contient alors des méthodes de liaisons qui sont appelées lorsqu'un fournisseur apparaît ou disparaît. Bien que ceci offre plus de liberté aux développeurs, ce modèle de développement peut entrainer des erreurs. En effet, ces méthodes sont appelées de manières concurrentes par rapport à la logique-métier du composant. Le développeur doit alors gérer correctement ce problème (Figure 79). Cependant, là encore, le développeur n'a pas à déclarer la spécification de service requise. Celle-ci peut être collectée lors de la manipulation.

```
public class HelloConsumer {
  private Hello m_hello;
  public synchronized void bindHello(Hello h) { m_hello = h; }
  public synchronized void unbindHello() { m_hello = null; } // Must release references
  public synchronized void doSomething() {
         System.out.println(m_hello.getMesage());  // Uses the stored value
  }
}

<component classname="...HelloConsumer">
<requires>
    <callback type="bind" method="bindHello">
    <callback type="unbind" method="unbindHello">
</requires>
</component>
<instance component="...HelloConsumer"/>
```

Figure 79. Dépendance de service simple avec méthodes de liaison

3.3.2. Publication et Fourniture de service

La publication de service doit également être masquée pour le développeur. iPOJO gère cette publication automatiquement (ainsi que le retrait lorsque l'instance devient invalide). Par défaut, iPOJO publie toutes les interfaces implémentées par la classe d'implémentation du composant. Ainsi le développeur n'a pas à gérer la publication (Figure 80). De plus, les instances de cette classe d'implémentation ne seront créées que lorsqu'un consommateur requiert un des services fournis.

```
public class HelloProvider implements Hello {
            public void getMessage() {
                        return "Hello";
            }
}

<component className="...HelloProvider">
    <provides/>
</component>
```

Figure 80. Exemple de publication et de fourniture de service

Un code peut également manipuler des propriétés de service. Ainsi dès lors qu'une propriété est affectée d'une nouvelle valeur, iPOJO met à jour la publication de service afin de refléter ce changement (Figure 81).

```
public class HelloProvider implements HelloV2 {
          private String m_name;
          public void setName(String name) {
                    m_name = name; // Update the service publication
          }
          public String getMessage() {
                    return "Hello  " + m_name;
          }
}

<component className="...HelloProvider">
    <provides>
        <property name="hello.name" field="m_name">
    </provides>
</component>
```

Figure 81. Exemple de modification d'une propriété de service

3.3.3. Configuration et gestion d'état

Bien que le dynamisme issu des services soit un élément très important, il est également nécessaire de supporter la gestion de l'état lors de l'évolution d'un composant. Ainsi, le développeur peut déclarer des propriétés qui seront supervisées par iPOJO. Lors d'une invalidation ou d'un arrêt, celles-ci seront persistées puis réinjectées lors du redémarrage de l'instance ou de la revalidation.

Ces propriétés peuvent être injectées soit via des méthodes, soit dans des membres de la classe. Lorsque la propriété de configuration est injectée via une méthode, les changements sur cette propriété ne sont pas supervisés. Au prochain redémarrage la même valeur sera injectée (sauf si cette valeur a été modifiée dans le gestionnaire de configuration). Lors d'une injection dans un membre, iPOJO peut superviser les changements et donc stocker la dernière valeur afin de la réinjecter après une reconfiguration.

Par défaut, iPOJO utilise le gestionnaire de configuration spécifié dans OSGi™ afin de sauvegarder et restaurer les valeurs des propriétés.

4. Principes et concepts des types de composants composites pour supporter le dynamisme

Bien que créer des composants atomiques soit nécessaire pour développer des applications dynamiques, il est devenu également crucial de proposer un modèle de composition permettant d'assembler des applications tout en masquant le dynamisme. Ce modèle de composition doit se rapprocher de la composition proposée par les langages de description d'architecture. Des caractéristiques telles que le support de la hiérarchie sont également importantes.

iPOJO propose donc un nouveau style de composition appelé composition de service structurelle permettant de concevoir des applications dynamiques. La principale différence par rapport aux langages de compositions traditionnels est que ce modèle utilise la notion de service comme une entité de premier ordre en plus de posséder la notion d'instance traditionnelle. De plus, chaque composition possède son propre contexte de service. Ainsi les services exposés dans le composite sont isolés. Une application créée à l'aide de ce modèle pourra être géré de manière dynamique lors de l'exécution. Une telle composition (Figure 82) est décrite en fonction de trois différentes entités :

- Des sous-services (dépendance de service) qui sont des services disponibles à l'intérieur de la composition (et donc publiés dans le contexte de service attachés à la composition)
- Les services fournis qui seront des services *exportés* dans le contexte de service parent (ce sont alors des services fournis)
- Des instances qui sont instanciées directement tel que le propose les langages de description d'architecture traditionnelle

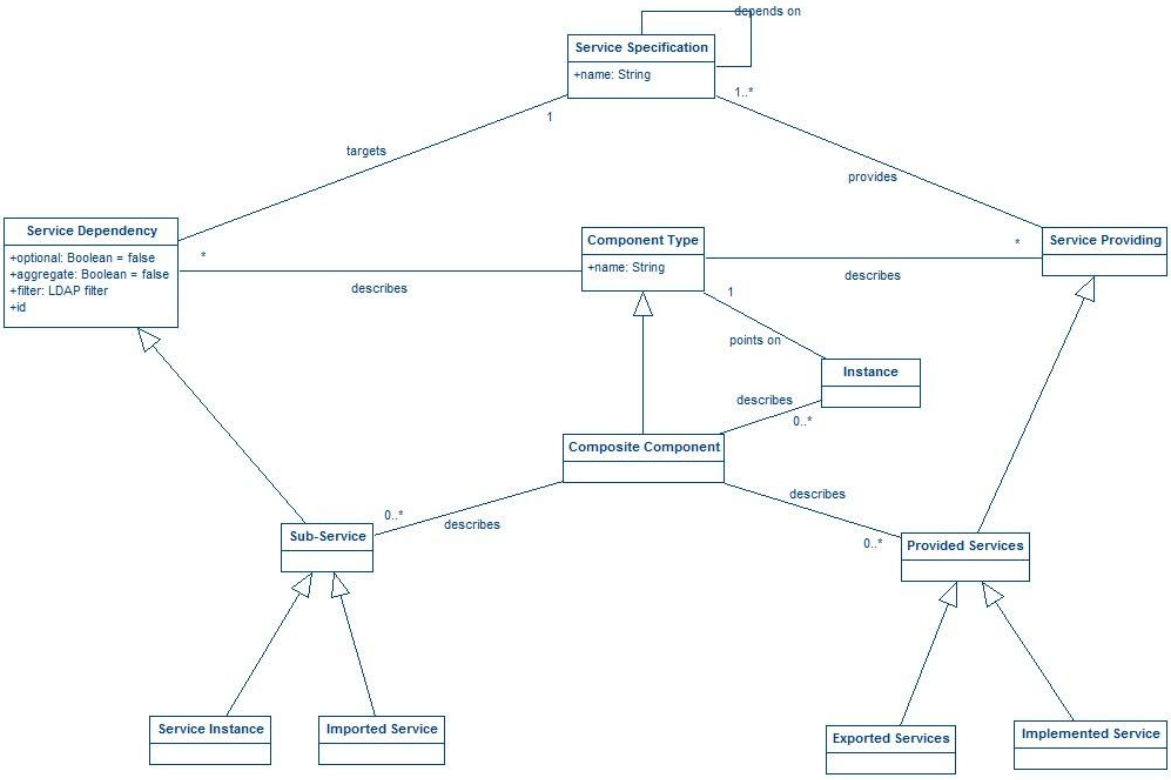


Figure 82. Concepts du modèle de composition

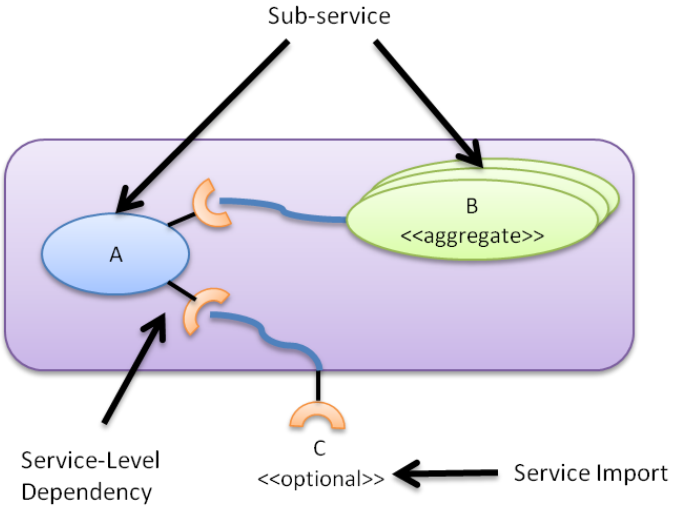
Les types de composant composite sont des types de composants. Une instance d'une composition peut suivre les mêmes règles qu'une instance d'un composant atomique. Cette section décrira plus en détails ces différents éléments d'une composition et comment ils réagissent face au dynamisme lors de l'exécution.

4.1. Sous-services

La notion de sous-service est l'implantation des dépendances de service pour les composites. Un composite peut ainsi utiliser des services. Cependant, deux types d'action sont envisageables afin de consommer ce service (Figure 83):

- Instancier le service au sein du composite
- Importer ce service depuis le contexte de service supérieur

Lors d'une instanciation, le composite recherche une implémentation de service. Si une implémentation compatible est disponible, une instance est créée. Cette instance vit au sein du composite. L'instance ainsi créée n'est disponible que dans le composite lui-même. Le dynamisme est supporté via le remplacement de l'instance si l'implémentation utilisée disparaît. En effet, cette dépendance étant une dépendance de service elle ne cible pas d'implémentation particulière. Ceci permet donc de substituer l'instance par une instance provenant d'une autre implémentation disponible. L'état peut être sauvegardé et réinjecté si la spécification de service définit son modèle d'état. Lorsque cette dépendance est agrégée, une instance par implémentation disponible est créée.




```
<composite name="Composite">
  <subservice action="instantiate" specification="A"/>
  <subservice action="instantiate" specification="B" aggregate="true"/>
  <subservice action="import" specification="C" optional="true"/>
</composite>
```

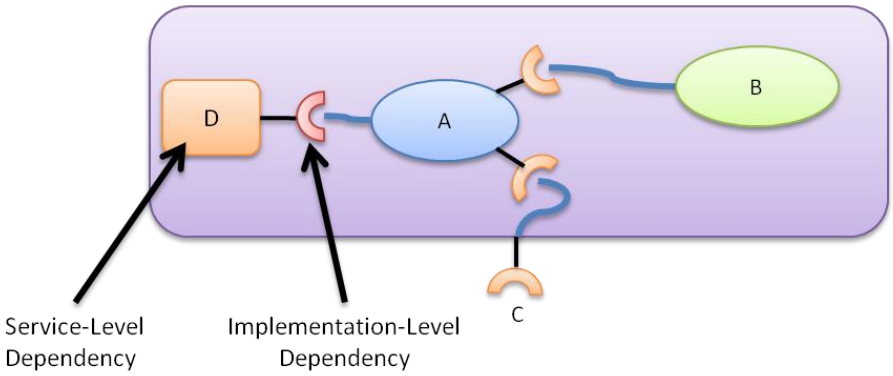
Figure 83. Exemple de sous-service et d'import de service

Un import de service publie un service disponible dans le contexte de service supérieur dans le contexte de service de la composition composite. Ce style de dépendance réagit de manière similaire aux dépendances de service des composants atomiques. Ainsi, dès lors qu'un fournisseur est disponible dans le contexte du parent, celui-ci est publié dans le contexte de service du composite. Ceci est très différent des services instanciés. En effet, le fournisseur importé n'est pas isolé dans le composite et donc peut éventuellement être utilisé par d'autres consommateurs vivant dans le contexte de service supérieur.

Ces deux dépendances suivent le modèle de dépendance de service d'iPOJO. Ainsi elles peuvent spécifier une politique de liaison, un comparateur, être agrégées, optionnelles ou filtrées. Les dépendances instanciées peuvent également fournir une configuration aux services instanciés.

4.2. Instances internes

La notion d'instances internes provient des langages de description d'architecture traditionnels. Grâce à ces instances, une composition peut demander la création d'une instance d'un type de composant particulier avec une configuration particulière. Cependant, le dynamisme lors de la gestion de ces instances est limité. En effet, vu qu'elle cible une implémentation particulière, ces instances ne peuvent être créées que lorsque la fabrique associée à cette implémentation est disponible.




```
<composite name="Composite">
  <subservice action="instantiate" specification="A"/>
  <subservice action="instantiate" specification="B"/>
  <subservice action="import" specification="C"/>
  <instance component="D"/>
</composite>
```

Figure 84. Exemple de composition contenant une instance interne

Une instance créée ainsi vit dans le contexte de service de la composition et donc n'a accès qu'au service publié dans ce contexte de service (Figure 84). Ces instances peuvent être utilisées lorsqu'une instance ne fournissant pas de service est nécessaire comme par exemple pour une interface graphique. Elles peuvent être également utilisées comme *glue code* afin de personnaliser une implémentation de service tel que cela sera défini dans la prochaine section. Dans ce cas, cette notion d'instance se rapproche de la notion de module présente dans Koala [107].

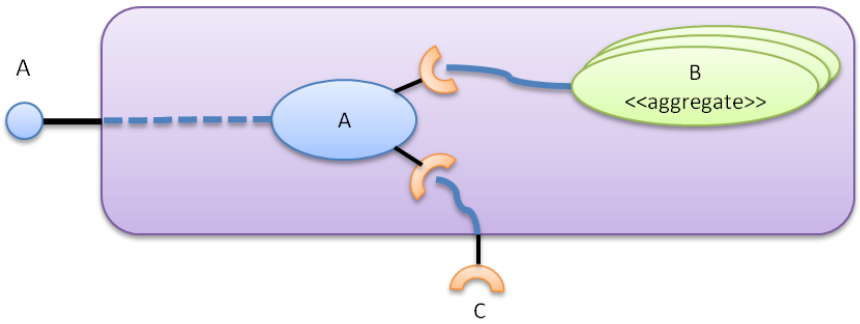
4.3. Fourniture de service et export de service

Une composition peut également fournir des services. Deux mécanismes sont disponibles pour cela :

- L'exportation de service
- L'implémentation de service

Dans les deux cas, le composite est alors considéré comme une implémentation du service et ainsi être instancié au sein d'une autre composition. Le modèle obtenu est donc un modèle hiérarchique identique au modèle de composition hiérarchique des modèles à composant. De plus, en tant qu'implémentation de service, la composition doit respecter les contraintes imposées lorsque les spécifications implémentées possèdent des dépendances de service.

Lorsqu'un service est exporté, le composite publie un service existant, c'est-à-dire publié dans son contexte de service, dans le contexte de service parent. Cette méthode requiert que le composite possède au moins un fournisseur de ce service. Cette exportation est très proche des liens *promotes* proposé par SCA.



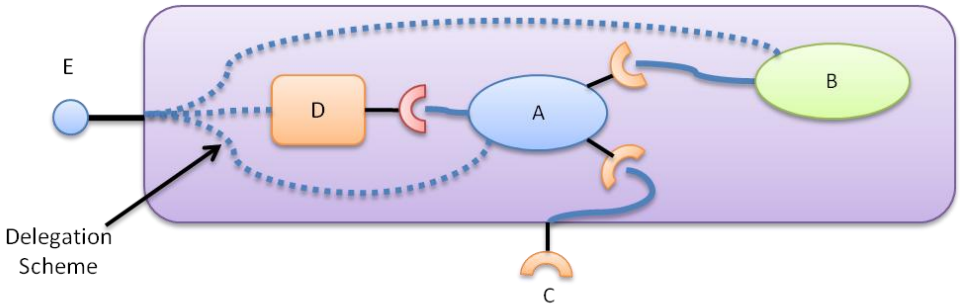

```
<composite name="Composite">
  <subservice action="instantiate" specification="A"/>
  <subservice action="instantiate" specification="B"/>
  <subservice action="import" specification="C"/>
  <provides action="export" specification="A" />
</composite>
```

Figure 85. Exemple de composition exportant un service

La deuxième méthode pour fournir un service est de l'implémenter. Cependant, ce service est implémenté par délégation. Ainsi chaque méthode est alors déléguée sur une entité vivante dans le composite possédant la même méthode (même nom et même argument).

Cette délégation peut être spécialisée pour chaque méthode pour indiquer le service à employer ainsi que la politique de délégation. Lorsque cette délégation n'est pas indiquée, iPOJO recherche automatiquement le schéma de délégation. iPOJO vérifie en permanence le schéma de délégation. Si un service ne peut plus être fourni (c'est-à-dire qu'une méthode ne peut plus être déléguée), l'instance devient invalide.

Deux politiques de délégation sont disponibles. La politique de délégation « One » délègue la méthode sur un seul fournisseur du service. La politique de délégation « All » délègue la méthode sur l'ensemble des candidats disponibles. Cette deuxième politique n'a de sens que pour les méthodes ne retournant pas de résultat.




```
<composite name="Composite">
  <subservice action="instantiate" specification="A"/>
  <subservice action="instantiate" specification="B"/>
  <subservice action="import" specification="C"/>
  <provides action="implement" specification="D" />
</composite>
```

Figure 86. Exemple de composition implémentant un service

Lorsqu'une méthode est déléguée sur un sous-service optionnel, celui-ci est potentiellement absent. Dans ce cas, la méthode lance une *UnsupportedOperationException*[30] pour indiquer l'impossibilité d'exécuter cette opération actuellement. Les instances internes sont également utilisées lors de la délégation. Ainsi une instance peut contenir un schéma de délégation pour une ou plusieurs méthodes d'un service fourni. Si une instance interne possède des méthodes d'un service fourni, elle sera systématiquement choisie pour exécuter ces méthodes (Figure 86). Il est ainsi possible de créer des composites fournissant des services à partir de sous-service tout rajoutant de la logique de composition au sein de ces instances internes.

4.4. Cohérence des compositions

La composition structurelle telle qu'illustrée dans cette section repose particulièrement sur les dépendances de niveau service introduits dans les spécifications de service. Grâce à ces dépendances, un assembleur peut décrire sa composition et vérifier que celle-ci est cohérente. En effet, une composition est cohérente si tous les services requis par les services et les instances internes introduits dans la composition seront disponibles lors de l'exécution.

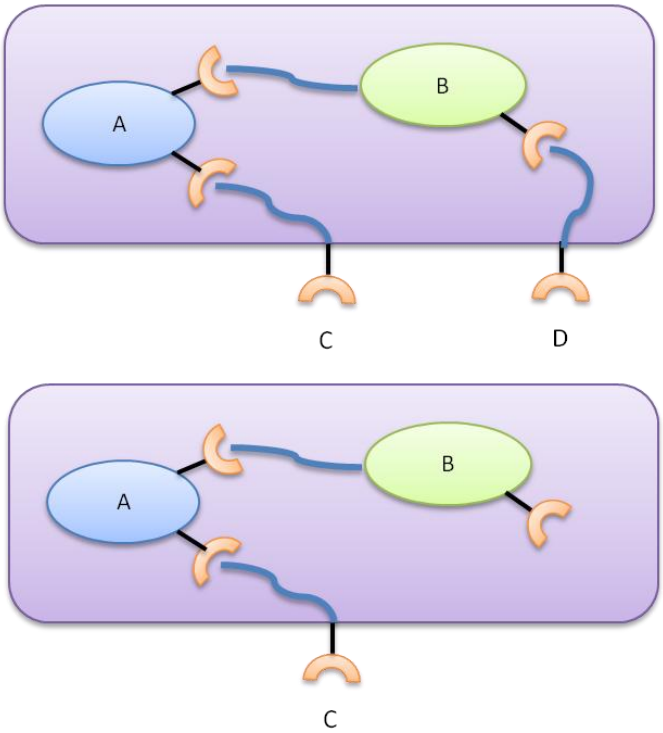


Figure 87. Une composition cohérente (en haut) et une composition incohérente en bas

Ainsi, iPOJO vérifie la cohérence de la composition lors de son déploiement. En plus de vérifier que le graphe de dépendances de service est cohérent, iPOJO vérifie également que toutes les méthodes des services fournis par implémentation peuvent être déléguées.

Il est important de comprendre que cette vérification ne concerne que les dépendances de service de niveau spécification. Ainsi lorsqu'une implémentation possède des dépendances de service d'implémentation celles-ci sont résolues différemment tel qu'expliqué page 111. La disponibilité de tel service ne peut être vérifiée à l'avance.

[30] http://java.sun.com/j2se/1.3/docs/api/java/lang/UnsupportedOperationException.html

4.5. Source de contexte et gestion du dynamisme contextuel

Bien que les concepts introduits dans la composition permettent de gérer le dynamisme provenant de l'évolution de l'application (via la substitution d'instance) ainsi que le dynamisme d'environnement (via les importations de service), le dynamisme provenant d'un changement de contexte n'est pas géré ainsi. En effet, il est parfois nécessaire de réagir pro-activement au dynamisme lors d'un changement de contexte [188].

Afin de supporter ce type de dynamisme, les sous-services peuvent déclarer des filtres contextuels. Ces filtres seront alors reconfigurés en fonction des sources de contexte disponibles. Une source de contexte est simplement un moyen pour obtenir des informations sur le contexte et être notifié de changement tel que défini dans [189]. Les sources de contextes (accessible sous la forme d'un service) peuvent être choisies ou non, locales au composite, globales ou importés du parent.

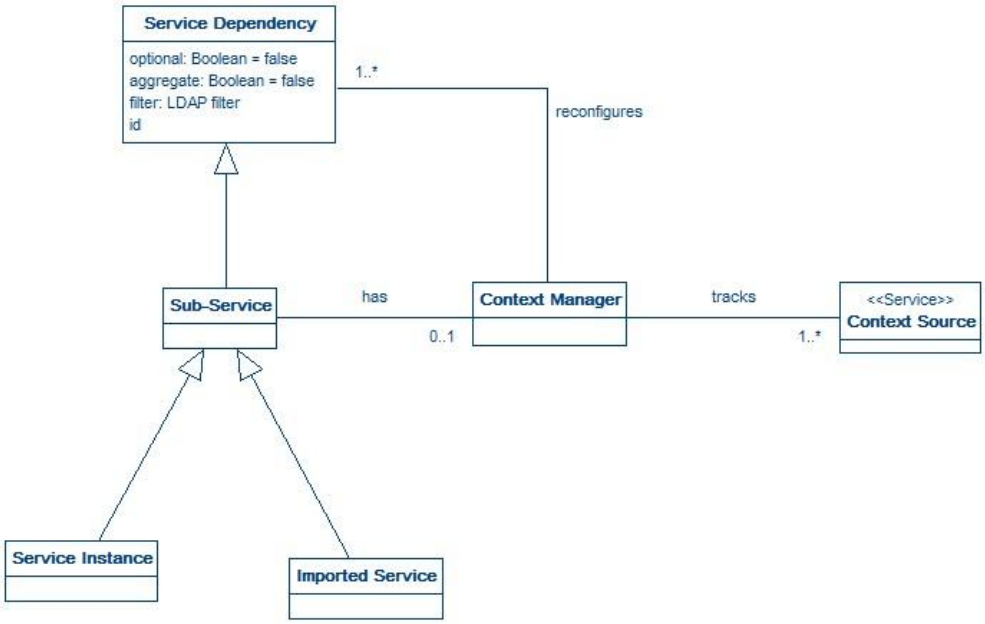
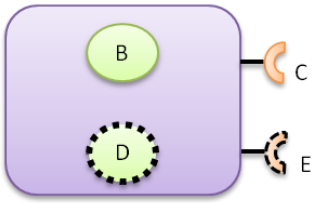


Figure 88. Reconfiguration des dépendances de service en fonction du contexte




```
<composite name="Composite">
  <subservice action="instantiate" specification="B"/>
  <subservice action="import" specification="C"/>
  <subservice action="instantiate" specification="D"
  filter="(setting=${my.setting})"
  context-source="parent:context-service"/>
  <subservice action="import" specification="E"
  filter="(location=${user.location})"
  context-source="local:context-service"/>
</composite>
```

Figure 89. Exemple de composition avec des filtres contextuels

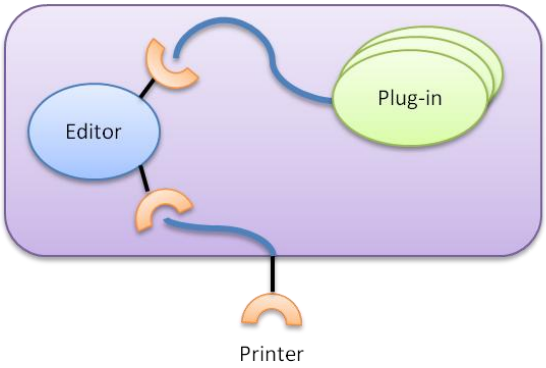
Le gestionnaire de contexte s'enregistre aux différentes sources de contexte (Figure 88). En fonction des informations disponibles, ce gestionnaire reconfigure les dépendances de service (à la fois les services instanciés et importés). Cette reconfiguration porte sur l'injection de valeur dans les filtres contextuels (Figure 89).

Grâce à ce filtrage contextuel, il est possible de créer des compositions réagissant au contexte. Le contexte peut donc influencer à la fois les services importés mais également les implémentations de service choisies rendant le modèle de composition très flexible.

5. Gestion du dynamisme dans les instances de composants composites

Afin de montrer les bénéfices du modèle de composition proposée, cette section présente une application simple conçue avec de modèle de composition. Cette application est un éditeur de texte. Celui-ci est une composition contenant un sous-service central (éditeur) ainsi que d'un ensemble de plugin. La spécification de service de l'éditeur définit les fonctionnalités basiques de l'édition de texte ainsi que des fonctionnalités de manipulation de fichier (sauvegarde, restauration). Cette spécification définit également des points d'extension afin que les plug-ins puissent rajouter des icones, des menus ou manipuler le texte ou la gouttière.

Cette spécification définit deux dépendances de service. La première cible des plug-ins est agrégée. La deuxième dépendance est une dépendance optionnelle, simple sur service d'impression. Dans la composition, la dépendance de service sur les plug-ins est réalisée via l'utilisation d'un sous service (Figure 90). Ainsi, pour chaque implémentation de plug-ins, une instance sera créée à l'intérieur du composite. Dès qu'une nouvelle implémentation devient disponible, une instance sera créée. Si une implémentation disparaît, l'instance créée avec cette implémentation sera détruite. L'absence d'implémentation invalidera la composition. La dépendance sur le service d'impression est réalisée par un import de service. Ce service sera donc importé du contexte de service parent. Dès que celui-ci est disponible, le service d'impression sera publié dans le contexte de service de la composition. Si ce service disparaît, alors la composition en cherchera un autre. Lorsqu' aucun service d'impression est disponible, le composite restera valide mais ne pourra pas utiliser les fonctionnalités d'impression.

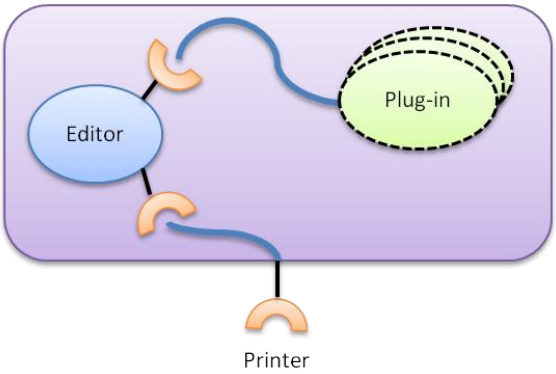



```
<composite name="Editor1">
  <subservice action="instantiate" specification="... Plugin" aggregate="true" />
  <subservice action="instantiate"  specification="...Editor"/>
  <subservice action="import"  specification="...Printer" optional="true"/>
</composite>
```

Figure 90. Editeur de texte simple

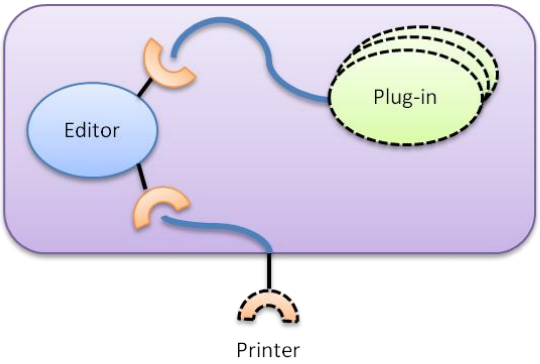
Malgré la simplicité de cette éditeur, celui illustre parfaitement comment une application à plug-in peut être conçue avec le langage de composition proposé et comment le dynamisme est géré. Il est également

possible d'aller plus loin et de filtrer les plug-ins afin d'instancier seulement les plug-ins utiles. Dans cette première version, tous les plug-ins disponibles étaient instanciés sans regarder leur utilité par rapport à l'édition en cours. Si l'on modifie l'éditeur afin d'également proposé un service de contexte (une source de contexte), il peut alors annoncer le type de fichier (par exemple le type MIME[31] du fichier) actuellement édité et notifié lorsque celui-ci change. Grâce à cette information, il est possible de filtrer dynamiquement les plug-ins utilisés. Ceci évite alors de surcharger l'éditeur avec des plug-ins inutiles (Figure 91).




```
<composite name="Editor2">
 <subservice action="instantiate" specification="... Plugin" aggregate="true"
  filter="(type=${my.type}) "  context-source= " local:editor" />
 <subservice action="instantiate"  specification="...Editor"/>
 <subservice action="import"   specification="...Printer" optional="true"/>
</composite>
```

Figure 91. Editeur de texte avec une dépendance contextuelle




```
<composite name="Editor3">
 <subservice action="instantiate" specification="... Plugin" aggregate="true"
  filter="(type=${my.type}) "  context-source= " local:editor" />
 <subservice action="instantiate"  specification="...Editor"/>
 <subservice action="import"  specification="...Printer" optional="true"
 context-source="global:location-source"
 filter="(&(printer.location=${current.location})(duplex=true))" />
</composite>
```

Figure 92. Editeur de texte réagissant au contexte

Dans cette seconde version, le sous -service représentant l'éditeur participe à la composition afin que celle-ci supporte les changements contextuels. En effet, grâce à la publication du type de fichier courant, seuls les plug-ins compatibles seront instanciés. Lorsque ce type change, l'ensemble des plug-ins à utiliser est

[31] Multipurpose Internet Mail Extension

réévalué. Ceci ne nuit pas au support du dynamisme : l'arrivée et le départ d'implémentation de plug-ins restent également gérés. Cependant, une implémentation apparaissant est utilisée seulement si elle est compatible avec le type de fichier actuellement édité. Cette séparation du support du contexte permet aux développeurs du service d'édition de se concentrer seulement sur le code métier de ce service. Il n'a donc pas à gérer ces changements de contexte.

Il est également possible d'aller encore plus loin si nous attachons un filtre contextuel à l'importation du service d'impression (Figure 92). Par exemple, la composition pourrait importer un service d'impression seulement si celui-ci est dans la même pièce que l'utilisateur. Ce filtre dynamique utilisera alors une source de contexte indiquant la position courante de l'utilisateur. Ainsi, les fonctionnalités d'impression seront activées et désactivées en fonction de la localisation de l'utilisateur et des services d'impression disponibles. En comparaison de la seconde approche ou la composition elle-même participer à la mise à jour du contexte, cette réaction à la localisation est totalement externalisée et transparente.

6. Synthèse

Ce chapitre a présenté les deux catégories de types de composant supportées dans iPOJO et plus exactement :

- Le modèle de développement des composants atomiques
- Le modèle de composition supportant le dynamisme

Ce chapitre a donc décrit comment les composants atomiques pouvaient être implémentés et quelles étaient les propriétés importantes que devait supporter le modèle de développement utilisé pour cette implémentation. Ainsi, ce modèle de développement doit être à la fois simple et flexible. La simplicité doit tendre vers un modèle de développement *POJO* ne contenant que la logique-métier du composant mais également à un descripteur de type de composant concis et simple. Ainsi le dynamisme, les problèmes de gestion d'état, de synchronisation et les primitives de l'approche à service ne doivent pas apparaître dans le code du développeur. Le descripteur contient toutes les informations nécessaires afin de gérer le dynamisme. Cependant, cette simplicité ne doit pas limiter les possibilités.

Afin de proposer ce modèle, les types de composants atomiques décrivent les services fournis et requis mais également les méthodes d'activation et de désactivation, les variables d'état … Grâce à cette description, iPOJO peut gérer tous les types de dynamisme lors de l'exécution sans que l'implémentation du composant n'ait à le traiter. Ce chapitre a également montré comment ce modèle peut être projeté sur Java et le modèle de développement résultant. Celui-ci remplit les contraintes énoncées précédemment. En effet, le code du développeur peut ne contenir que la logique-métier. L'intégralité du dynamisme est gérée de manière transparente. Le descripteur est rendu très simple en supprimant toutes informations superflues et redondantes.

De plus, ce chapitre a présenté le modèle de composition proposé par iPOJO. Ce modèle de composition utilise la notion de service comme entité de premier ordre. Ainsi une telle composition peut ne pas avoir de lien avec des implémentations spécifiques ce qui permet alors la substitution d'une implémentation par une autre. Ce langage de composition reste tout de même proche des langages de description d'architecture afin d'être intuitif. Les compositions proposent également une notion d'isolation similaire à la composition verticale de l'approche par composant. Il a également montré comment une composition pouvait importer des services provenant du contexte de service supérieur.

Afin de supporter le dynamisme provenant de changement contextuel, la notion de filtre contextuel reconfigurant les dépendances de service a été présentée dans ce chapitre. Ainsi une application conçue avec

ce langage de composition devient automatiquement dynamique. Elle supporte toutes les sources de dynamisme, tels que l'évolution, le dynamisme d'environnement et les changements contextuels.

Bien que ces fonctionnalités décrites dans ce chapitre permettent de créer et d'exécuter des applications dynamiques, elles ne sont pas suffisantes. En effet, un SOA étendu dynamique doit également proposer des fonctionnalités d'introspection et de reconfiguration. De plus, l'ajout de nouvelles préoccupations non fonctionnelles est également un besoin important. Le chapitre suivant présentera les capacités d'introspection et de reconfiguration proposées par iPOJO. De plus, le mécanisme d'extensibilité mis en place sera également présenté afin de montrer comment des nouveaux besoins non-fonctionnels peuvent être gérés tout en garantissant le dynamisme.

Chapitre 8
Introspection, reconfiguration & extensibilité

iPOJO, le modèle à composant à service proposé dans cette thèse permet le développement d'application dynamique en combinant les principes de l'approche à service, la notion de langage de description d'architecture et les capacités des modèles à composant. Au-dessus d'un SOA dynamique et hiérarchique, les composants iPOJO peuvent être implémentés concrètement (type de composant atomique) ou de manière abstraite (composition). Dans les deux cas, le dynamisme est masqué et est géré de manière transparente lors de l'exécution.

Cependant, ces deux fonctionnalités ne suffisent pas à implanter toutes les capacités requises d'un SOA dynamique étendu. Bien que la base de cette pyramide soit implantée par le SOA dynamique hiérarchique d'iPOJO et que le langage de composition permet la mise en place d'application dynamique et leur vérification à l'exécution, il est nécessaire d'offrir d'autres fonctionnalités permettant la gestion, la vérification et la reconfiguration du système dynamique exécuté. De plus, cette thèse s'est focalisée sur le dynamisme. Cependant, d'autres besoins non-fonctionnels sont généralement requis. L'intégration de ces besoins dans un milieu dynamique doit également être gérée.

Introspecter des applications dynamiques est un besoin essentiel. En effet, l'architecture réelle (c'est-à-dire les composants et connecteurs réellement utilisés) est dynamique. La validité d'une application est donc tributaire de la disponibilité des composants nécessaires. Afin de comprendre et de corriger des problèmes, il est crucial d'être capable d'introspecter et de visualiser l'architecture actuelle de l'application ainsi que les services et implémentations de service disponibles.

Une fois un problème détecté et compris, il est également nécessaire de reconfigurer l'application. Cependant, vu que cette application est décrite de manière abstraite et est assemblée dynamiquement à l'exécution, il n'est pas possible de manipuler directement les éléments formant cette application telle que le font les modèles à composant traditionnels. Reconfigurer de telles applications consiste donc à modifier les contraintes des dépendances de service afin d'agir sur l'application. La modification de ces contraintes a un effet direct sur l'architecture de l'application. De plus, la reconfiguration des instances (modification de paramètres) et des publications de service (c'est-à-dire le changement des propriétés de service publiées) sont également nécessaires.

Enfin, il est également nécessaire de pouvoir supporter de nouveaux besoins non-fonctionnels. L'introduction de ces besoins dans un milieu dynamique peut s'avérer complexe. Il est donc nécessaire de proposer une infrastructure permettant l'extension du modèle. Développer ces extensions doit être le plus simple possible et doit également masquer le dynamisme pouvant intervenir dans ces extensions.

Ce chapitre décrit les fonctionnalités que propose iPOJO afin d'introspecter un système ainsi que les capacités de reconfiguration mises en place. De plus, l'infrastructure permettant l'extensibilité du modèle sera présentée et plus particulièrement comment de nouveaux besoins non-fonctionnels peuvent être gérés tout en garantissant le dynamisme de l'application.

1. Introspection d'applications à service dynamiques

Les applications dynamiques sont des applications dont l'architecture évolue à l'exécution. De ce fait, il est extrêmement important de pouvoir introspecter un système en cours d'exécution afin de comprendre l'architecture actuelle et si besoin est de la corriger (c'est-à-dire reconfigurer l'application).

Dans le cadre d'application à service dynamique telle que celle composée avec iPOJO plusieurs informations sont nécessaires afin de connaitre l'état du système :

- Les services actuellement disponibles dans le contexte global
- Les types de composants et les implémentations de services disponibles
- La structure des instances vivant dans le contexte global et plus particulièrement la structure des compositions

Tout d'abord, la liste des services disponibles dans le contexte global est importante pour comprendre comment sont résolues les dépendances de service (pointant vers des fournisseurs concrets) utilisant ce contexte de service. Ainsi, la liste des services publiés ainsi que les propriétés de service attachés sont nécessaires. Ces informations peuvent facilement être obtenues en questionnant le courtier de service du contexte global.

Ensuite, les types de composants disponibles sont une information importante à connaitre. En effet, avec ces types il est possible de comprendre pourquoi certaines instances ne sont pas créées (types associés absents). Plus particulièrement, cette information permet de connaitre les implémentations de service disponibles et donc de comprendre pourquoi certaines dépendances de service (résolue par instanciation) restent insatisfaites.

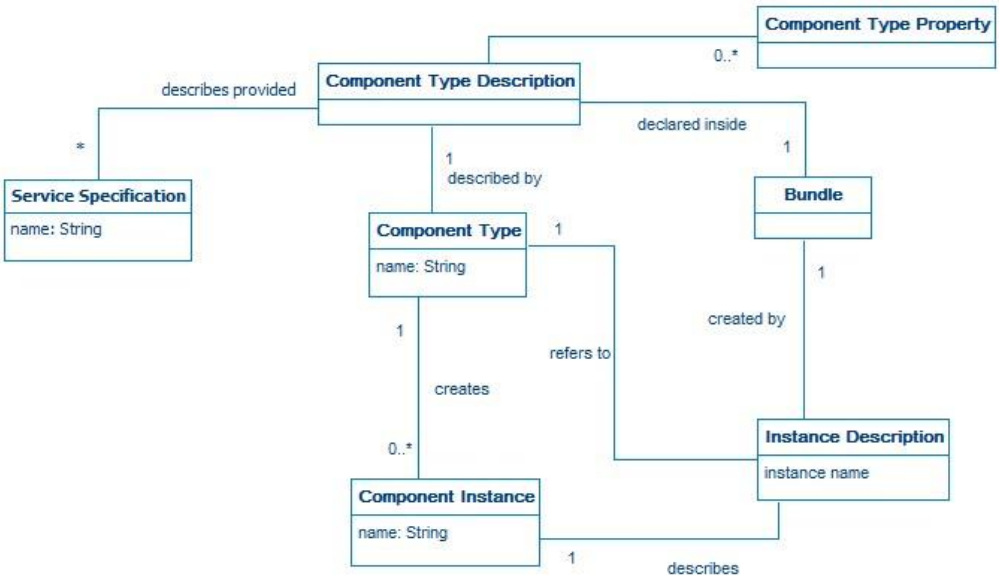


Figure 93. Description des types de composant

Afin d'obtenir la description des types de composant, iPOJO fournit un outil collectant et traitant les informations sur les types de composants disponibles[32]. Cette description suit le modèle ci-dessus. La description d'un type de composant est relativement simple. Elle est composée principalement des

[32] Cette description est également disponible dans le courtier de service car publiée avec le service de fabrique associé aux types

spécifications de service implémentées (et donc publiées par les instances de ces types) et de propriétés. À partir de ces propriétés, un développeur peut connaitre les propriétés nécessaires afin de créer des instances.

Enfin, la dernière catégorie d'information permet de comprendre la structure des applications et donc donne l'architecture actuelle des applications exécutées. Chaque instance, atomique ou composite, est décrite par une description d'instance (Figure 94). Cette description donne toutes les informations nécessaires afin d'inférer l'architecture de l'application.

Tout d'abord chaque description d'instance donne l'état actuel de l'instance et permet d'obtenir le type de composant (et donc sa description). La description d'une composition contiendra la description des instances internes créées (ainsi que leurs interconnections). Chaque description donne également des informations sur les services fournis par l'instance (provided service description) et sur les services requis par l'instance (service dependency description). Cependant, en fonction de l'instance, ces informations pourront être complétées avec des informations spécifiques telles que l'implémentation par défaut (et si elle est actuellement utilisée), le schéma de délégation …

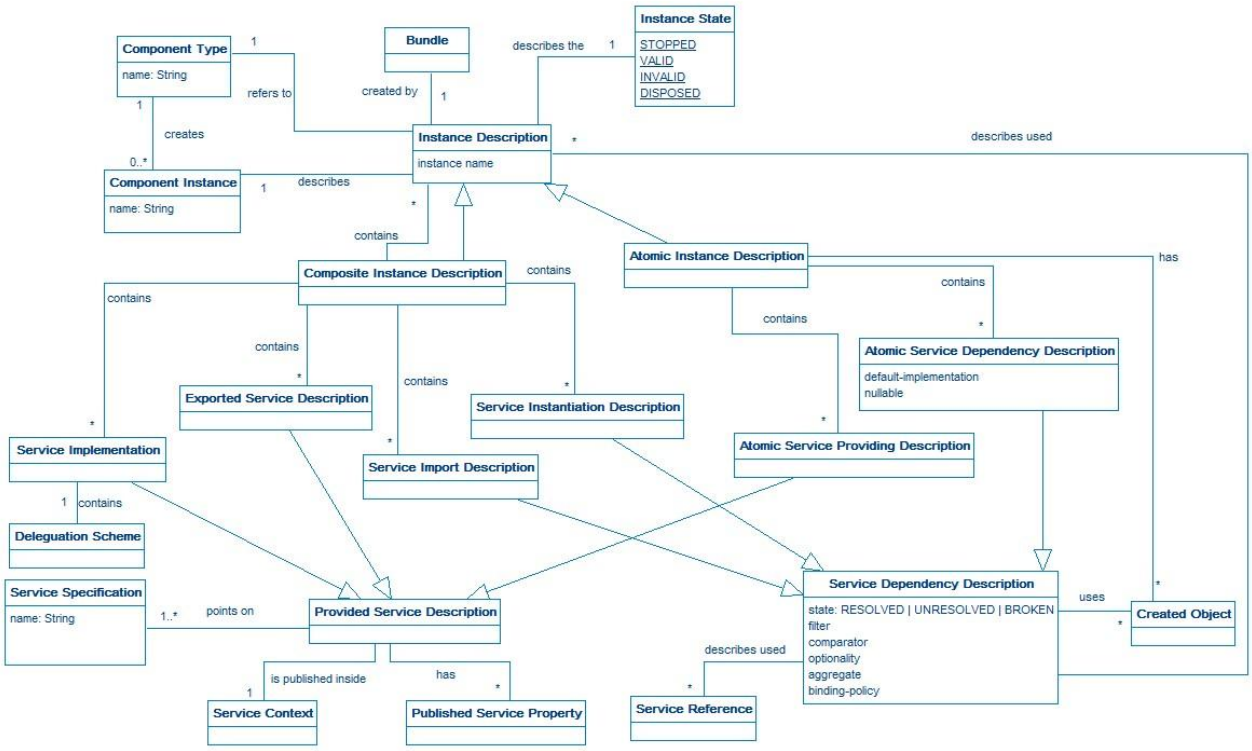


Figure 94. Modèle de description d'instance[33]

Chaque description de service fourni indique le contexte de service dans lequel le service est fourni ainsi que les propriétés de service publiées ainsi que leur valeur actuelle. En fonction du type de publication certaines informations sont ajoutées à cette description telle que :

- Le nom, le type et la description de l'instance interne utilisée pour exporter un service dans une composition
- Le schéma de délégation, c'est-à-dire les instances ou services choisis pour déléguer les méthodes d'un service fourni, dans le cadre d'un service implémenté par un composite

La description des dépendances de service indique l'état actuel de la dépendance, comment celle-ci est configurée actuellement (filtre, comparateur, politique de liaison, optionalité, agrégation …) et donne des

[33] Certains détails du modèle ont été masqués afin de le rendre plus lisible

informations sur les fournisseurs (ou implémentation) utilisés. En effet, iPOJO permet de découvrir qu'elles sont les instances utilisées, ou dans le cas de service non fourni par une instance iPOJO, des informations sur le fournisseur[34]. Ainsi, il est possible de reconstituer le graphe d'instances et donc d'avoir la structure de l'application.

Ainsi, grâce à sa capacité d'introspection proposée par iPOJO, il est possible de connaitre la structure actuelle des applications, les instances utilisées, les services et types de composant disponibles … Ces informations s'avèrent extrêmement utiles afin de comprendre pourquoi une application ne fonctionne pas correctement.

2. Reconfiguration dynamique d'applications à service

La section précédente a présenté les capacités d'introspection fournies par iPOJO. Cependant, cette introspection se limite à la remontée de l'architecture actuelle du système. Il est donc parfois nécessaire de reconfigurer ce système et les applications exécutées [190].

Comme l'a expliqué le chapitre 3, la reconfiguration dynamique est le processus de modification de l'architecture d'une application durant son exécution. Plusieurs actions sont définies afin de manipuler ces architectures telles que l'ajout, la suppression ou le remplacement d'instance, ainsi que l'ajout ou la suppression de connecteurs. Cependant, ces actions n'ont pas de sens dans les applications dynamiques conçues à l'aide du langage de composition d'iPOJO, car la notion d'instance y est limitée et les connecteurs suivent le paradigme de l'approche à service. Ainsi la manipulation de l'architecture d'une application décrite par une composition à service structurelle est basée sur deux actions :

- La reconfiguration des dépendances de service
- La reconfiguration des instances « concrètes », comme les instances internes des composites.

Cette section présente ces deux actions et comment elles sont utilisées pour reconfigurer dynamiquement des applications.

2.1. Reconfiguration des dépendances de service

Les compositions structurelles de service telles que celle proposée par iPOJO sont décrites principalement en terme de dépendances de service. Ces dépendances sont résolues de deux façons différentes :

- Par l'instanciation d'un sous-service à l'intérieur du composite
- Par l'importation d'un fournisseur du contexte de service Parent dans celui du composite

La manipulation de ces deux dépendances influence directement la structure de l'application. En effet, en contraignant ou en paramétrant ces dépendances il est possible de forcer le changement d'implémentation de service utilisé, le changement de configuration des instances créées ou de sélectionner plus finement les fournisseurs de service importé. Ainsi à la place de manipuler directement l'architecture de l'application, ce sont les dépendances de service qui sont manipulées. De cette façon, il n'est pas injecté de dépendances fortes

[34] Les instances iPOJO peuvent utiliser des services fournis par des *bundles* OSGi™ standards. Cependant, certaines informations ne seront pas disponibles dans ce cas là. Seule la référence de service utilisée ainsi que le *bundle* publiant ce service seront indiqués.

sur des implémentations de service spécifiques, sur des fournisseurs de service particuliers[35] et donc le dynamisme pourra toujours être géré.

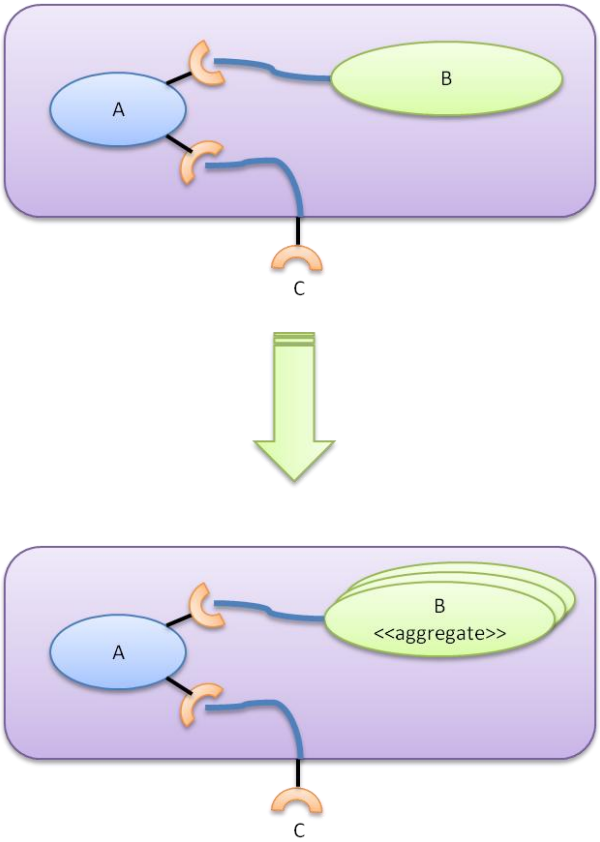


Figure 95. Exemple de reconfiguration de dépendance de service

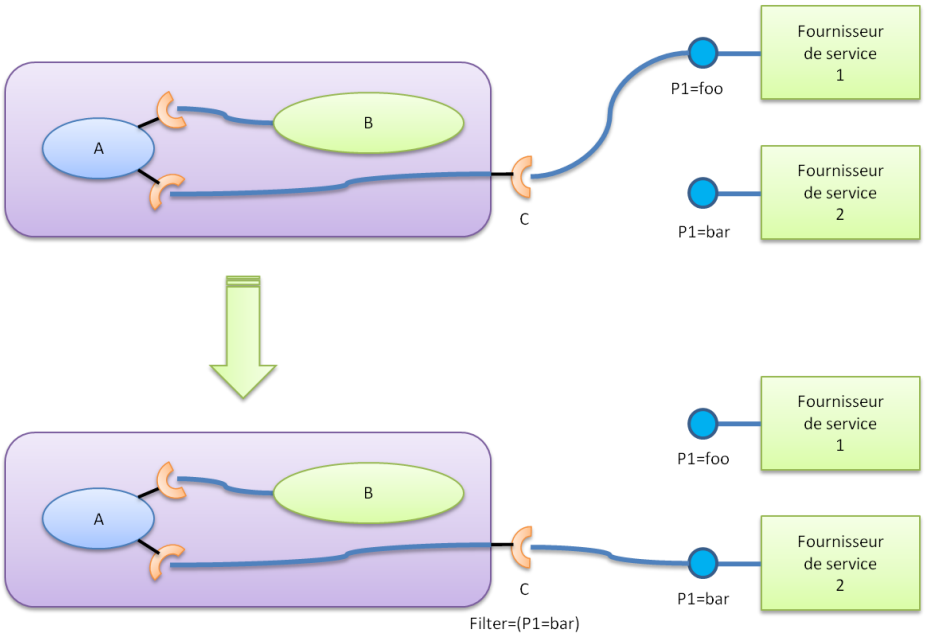


Figure 96. Exemple de reconfiguration de dépendance de service

[35] Il est néanmoins possible de reconfigurer les dépendances pour les forcer à choisir une implémentation particulière ou à un fournisseur particulier.

Lors d'une reconfiguration, la spécification de service requise ne peut être changée. Cependant tous les autres attributs du modèle de dépendance de service peuvent être modifiés. Ainsi, une dépendance optionnelle peut devenir obligatoire ou vice-versa, une dépendance simple peut devenir agrégée (Figure 95), les filtres de dépendances peuvent être modifiés (Figure 96), la politique de liaison peut être changée ainsi que le comparateur utilisé. Dans le cas de sous-service instancié, la configuration de l'instance peut également être modifiée.

La reconfiguration de dépendance de service possède des propriétés très intéressantes afin de conserver le dynamisme et permet donc d'influencer directement l'architecture actuelle d'une application. De plus, ces reconfigurations n'interrompent pas l'exécution de l'application.

2.2. Reconfiguration des instances « concrètes »

Bien que la reconfiguration des dépendances de services permet de manipuler l'architecture des compositions, il est parfois nécessaire de reconfigurer les fournisseurs de service et les instances internes créées dans les composites. Ainsi iPOJO permet de reconfigurer la configuration interne d'instance ainsi que de modifier les propriétés de service publiées.

La reconfiguration d'instance vise à changer les paramètres d'une instance créée. Celle-ci doit être créée de manière statique, c'est-à-dire en décrivant spécifiquement le type de composant. Plus exactement, ce type de reconfiguration permet de modifier les instances internes créées dans des compositions qui ne sont pas exprimées sous la forme de dépendance de service, ou des instances créées dans le contexte de service global (Figure 97).

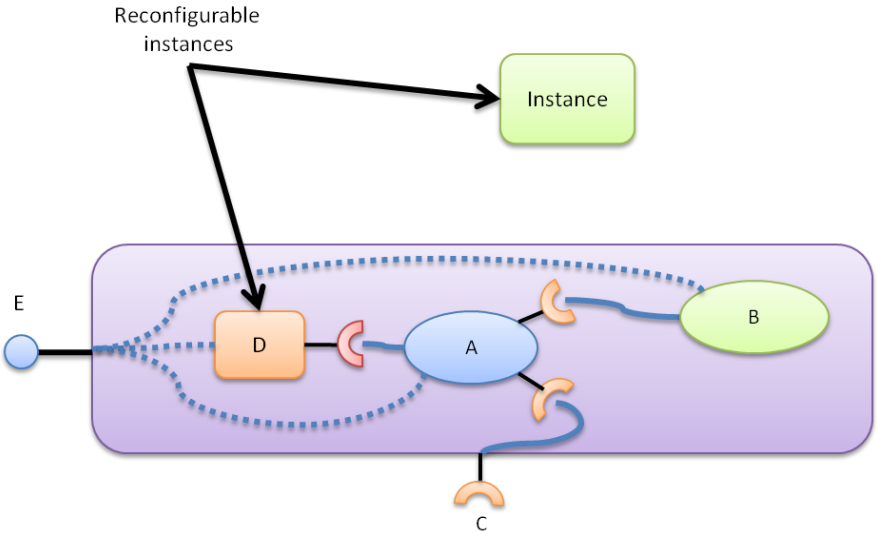


Figure 97. Instances reconfigurables directement

Ces instances peuvent être reconfigurées dynamiquement en leur réinjectant une configuration. Cette configuration peut influencer à la fois le comportement interne de l'instance, mais également les propriétés de service si cette instance publie des services. Ainsi il est possible d'ajouter, de supprimer ou de modifier les propriétés de service publiées par l'instance.

3. Un modèle extensible

Cette thèse s'est particulièrement focalisée sur le dynamisme. Cependant, le dynamisme n'est pas le seul besoin non-fonctionnel. Il est donc également important de supporter d'autres besoins tels que la persistance, la sécurité ou d'autres types de connecteurs comme l'envoi et la réception évènements.

Cependant l'ajout de ces nouveaux besoins peut s'avérer délicat lorsque l'environnement est dynamique. Par exemple, l'envoi et la réception d'évènement peuvent dépendre d'un intergiciel à message accessible sous la forme de service pouvant lui-même évoluer ou devenir inaccessible. Ainsi, ces besoins peuvent être soumis au dynamisme et donc doivent le gérer.

Cette section décrit les mécanismes mis en place dans iPOJO permettant d'étendre le modèle et de supporter de nouveaux besoins non-fonctionnels. L'ajout de nouveau besoin non-fonctionnel résulte en l'extension du modèle de composant iPOJO.

3.1. Composition des conteneurs et *handlers*

Afin d'être extensible, iPOJO propose d'utiliser le principe de conteneur ouvert [191, 192]. Ainsi, les conteneurs d'instance d'iPOJO ne sont pas monolithiques, mais sont eux-mêmes composés de traitants appelés *handler* (Figure 98). Chaque handler traite un ou plusieurs besoins non fonctionnels. Toutes les capacités du modèle d'iPOJO sont implémentées sous la forme de handler, y compris la gestion du dynamisme et la gestion des primitives de l'approche à service dynamique. Les handlers peuvent être délivrés dans des unités de déploiement (*bundle*) différentes de celle contenant le cœur d'iPOJO. Ceci donne une grande flexibilité. En effet, chaque plate-forme iPOJO peut contenir un ensemble d'extension différent sans pour autant faire grossir le modèle cœur (Figure 99).

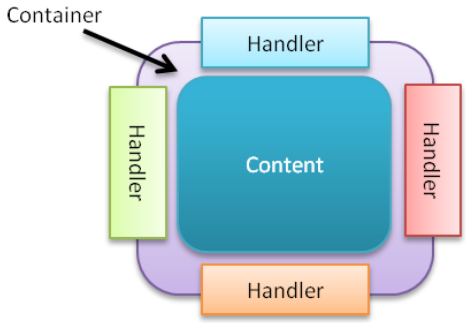


Figure 98. Composition des conteneurs des instances de composant iPOJO

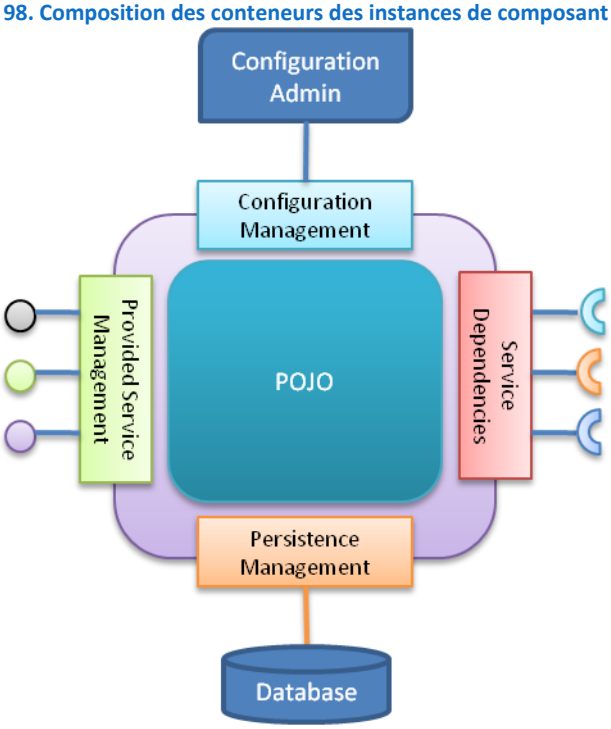


Figure 99. Exemple de composant utilisant des handlers fournis par iPOJO et des handlers externes

La notion de handler revisite l'automate du cycle de vie de l'instance. Auparavant, une instance était valide si toutes les dépendances de service étaient satisfaites. En fait, une instance est valide si tous les handlers attachés à son conteneur sont valides (Figure 100). Comme la gestion des dépendances de service est effectuée dans un handler, ce handler est valide si les dépendances gérées sont satisfaites.

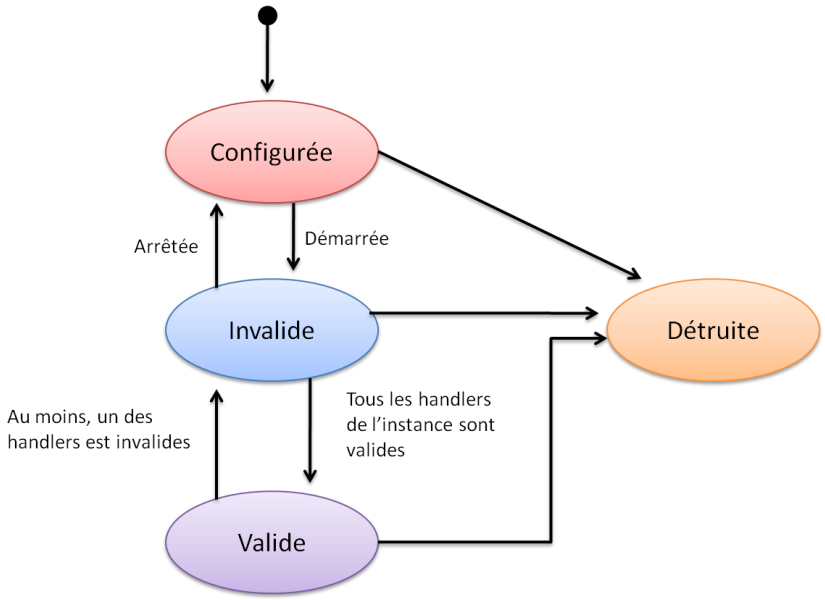


Figure 100. Cycle de vie des instances en fonction des handlers

Chaque handler est un type de composant iPOJO primitif. Chaque handler est implémenté sous la forme d'un POJO *spécial* (étendant la classe *Handler*) et peut utiliser tous les autres handlers disponibles. De cette manière, les handlers peuvent utiliser les handlers définis par iPOJO gérant le dynamisme et les primitives de l'approche à service. Ainsi l'implémentation d'un handler peut supporter le dynamisme très simplement, et son code contient seulement la logique de configuration, de traitement du besoin non-fonctionnel associé et les liens avec le contenu de l'instance.

La description d'un handler est très proche de la description d'un type de composant primitif Java. La seule différence réside dans la description d'un espace de nom et d'un nom indiquant l'élément géré par le handler (Figure 101).

```
<handler  classname="o.a.f.i.h.eventadmin.EventAdminSubscriberHandler"
          name="subscriber"
          namespace="http://.../subscriber">
          <provides>
                    <property
                              field="m_topics"
                              name="event.topics"
                              value=""
                    />
          </provides>
</handler>
```

Figure 101. Type de composant définissant un handler, utilisant lui-même le handler de publication de service

Comme les handlers sont des instances de composant iPOJO, ceux-ci sont créés à l'aide de fabrique comme les autres instances. Cependant, ceci influe sur les fabriques des composants dépendantes de ce handler, car cette fabrique peut apparaître et disparaître. En effet, la dépendance entre un type et les handlers utilisés est une relation fortement couplée. La fabrique d'un type de composant utilisant un handler particulier ne peut créer d'instance que si tous les handlers requis sont disponibles (sinon le conteneur de l'instance ne serait pas complet). De plus, lorsque la fabrique d'un handler disparaît, toutes les instances requérant ce

handler sont détruites. Aucune autre instance d'un type dépendant de ce handler ne peut être créée tant que la fabrique associée à ce handler ne réapparaît pas.

L'extensibilité a un impact direct sur l'introspection. En effet, chaque handler peut participer à la description de l'instance. Il peut donc indiquer sa validité ainsi que des informations sur son état. Ainsi, la description d'une instance contient l'ensemble des descriptions proposées par les handlers composant le conteneur de l'instance. Les handlers peuvent également être reconfigurables et profiter des mêmes capacités de reconfiguration que les autres instances de composition d'iPOJO.

Les mécanismes d'extensibilité d'iPOJO s'appliquent à la fois au niveau des composants atomiques et composites. Les deux prochaines sections présentent les caractéristiques et des exemples d'extension.

3.2. Extensibilité des composants atomiques

Étendre les composants atomiques consiste à créer de nouveaux handlers gérant de nouvelles propriétés non-fonctionnelles non gérées par défaut par iPOJO. Ces handlers profitent également de la machine d'injection fournie par iPOJO ce qui permet donc de conserver un modèle de développement très simple. Cette section présente deux exemples de handlers pour composant primitifs.

De nombreux travaux proposent des solutions pour administrer des passerelles OSGi à distance. Cependant ces travaux proposent généralement des fonctionnalités de déploiement ou de niveau passerelle, mais ne permettent pas de reconfigurer finement les instances exécutées actuellement sur cette passerelle. Lorsqu'un développeur a besoin de créer un composant reconfigurable à distance il doit gérer dans son code ce code de reconfiguration. Ceci est souvent délicat et manipule des technologies, telles que JMX, pas forcément dans le domaine de compétence du développeur. Le handler d'administration permet de séparer ce code en proposant les fonctionnalités de reconfiguration à distance. Celles-ci seront accessibles en utilisant JMX. L'implémentation du composant n'a pas connaissance de ces reconfigurations : les reconfigurations sont effectuées par le handler avec des appels réflectifs ou de l'injection de valeurs dans des membres (Figure 102). Ainsi, un administrateur se connectant sur le serveur JMX pourra reconfigurer ou effectuer des actions sur l'instance.

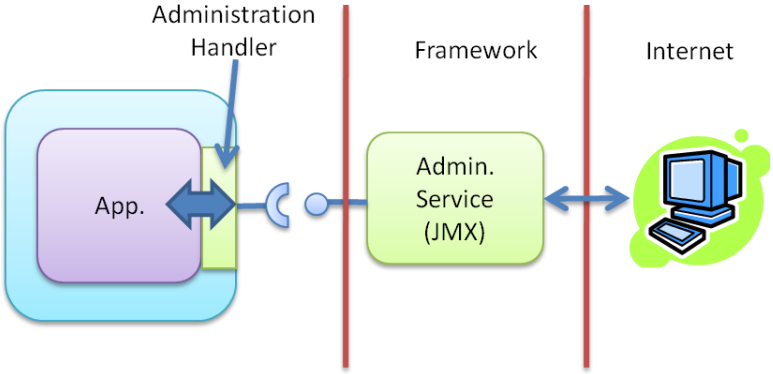


Figure 102. Schéma de principe du handler d'administration

Ce premier handler externe a montré comment on pouvait rajouter des propriétés non-fonctionnelles. Les handlers permettent également de définir de nouveaux types de connecteurs. Un connecteur est alors généralement un ensemble de deux handlers, chacun représentant les extrémités du connecteur. Par exemple, un connecteur dans lequel transitent des évènements est implanté par un handler d'émission, un handler de réception et un intergiciel à message (Figure 103). L'utilisation de ces handlers permet de masquer la complexité des interactions avec l'intergiciel à message utilisé. De plus celui-ci peut être dynamique et donc apparaître ou disparaître dynamiquement.

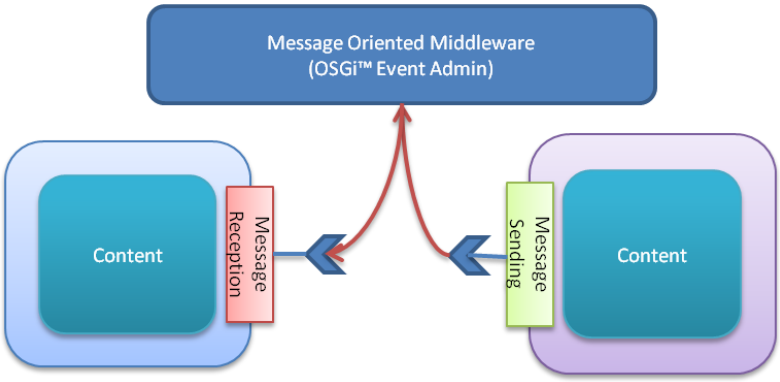


Figure 103. Schéma de principe du handler de communication par évènement

Le modèle de développement proposé pour la réception est très simple. L'implémentation est notifiée de la réception d'un message par l'invocation d'une méthode spécifiée dans la configuration du handler. Le handler gère alors la souscription aux différents sujets et comment ces évènements sont réceptionnés. L'envoi est effectué en appelant une méthode *send* sur un objet injecté dans l'implémentation. Ainsi, en appelant cette méthode, l'implémentation envoie un message sur les sujets et en utilisant le modèle d'interaction (synchrone ou asynchrone) décrit dans la description du handler. Bien que ce modèle de développement n'est pas POJO (car impose une dépendance entre l'implémentation du composant et le handler), ce modèle est relativement simple. Actuellement, il n'est pas possible de proposer un modèle de développement POJO, lorsque celui-ci manipule des « fluent interfaces » [193].

3.3. Extensibilité des composants composites

Le mécanisme d'extensibilité est également disponible au niveau des composites. Ainsi il est possible de modifier le langage de composition afin de rajouter d'autres concepts. Un handler composite peut accéder au contexte de service interne et celui du parent. Il peut alors gérer des importations, des exportations ou des instanciations.

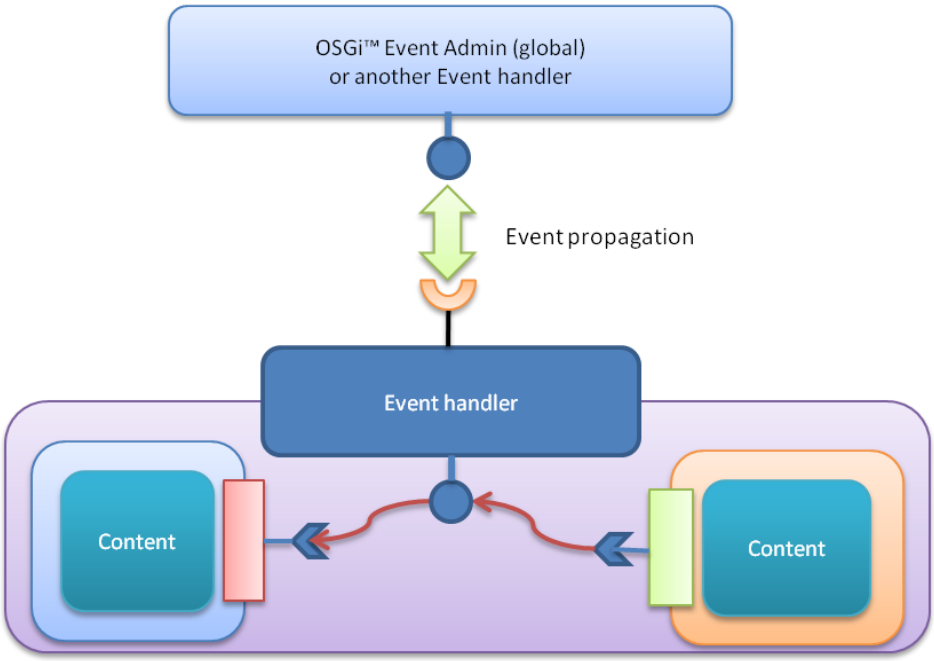


Figure 104. Support de la communication par évènement locale au composite

L'un des gros avantages des handlers et des composites est qu'ils peuvent publier des services dans le contexte de service du composite. Par exemple, dans le contexte d'une communication par évènement, un handler de composition dédié donne accès à un intergiciel à message à l'intérieur du composite. Ainsi les

instances vivant dans ce composite utiliseront cet intergiciel ce qui permet de garantir l'isolation des évènements. Cependant, ce handler peut être configuré pour importer ou exporter des évènements provenant de l'intergiciel à message présent dans le contexte de service parent (Figure 104). Comme le handler publie le même service que l'intergiciel à message définit par OSGi™, les instances internes utiliseront celui-ci de manière transparente.

Les handlers de composite peuvent également fournir des mécanismes ou des informations agrégées. Ainsi, un handler de composite gère la configuration des instances internes. Ce handler peut être reconfiguré à l'exécution, reconfigurant ainsi les instances locales. De la même façon, un handler de composite peut publier un service de journalisation dans le contexte de service du composite utilisé par les instances pour laisser une trace de leurs activités. Cette journalisation locale permet d'éviter que toutes les applications écrivent dans le même service de journalisation (global).

4. Synthèse

Ce chapitre a présenté les capacités d'introspection, de reconfiguration et d'extensibilité proposées par iPOJO. En effet, afin d'implanter correctement un SOA étendu dynamique, ces fonctionnalités sont nécessaires.

L'introspection est un requis particulièrement important dans le contexte des applications dynamiques. En effet, l'architecture de ces applications évolue à l'exécution. Il est donc primordial d'être capable de remonter et de visualiser l'architecture courante d'un système. Cette représentation contient la liste des services disponibles dans le contexte de service global, la liste des implémentations de service et des types de composant disponibles, l'architecture des différentes applications et instances créées. Grâce au modèle de dépendances utilisé par iPOJO, il est possible d'inférer les connecteurs et donc la structure du système. De plus, les informations remontées permettent à l'administrateur d'avoir une vision précise de l'état des instances et donc de comprendre pourquoi certaines applications ne fonctionnent pas correctement.

Cependant, une fois introspecté, il est parfois nécessaire d'agir sur le système. Cependant, les principes de reconfigurations dynamiques ne s'appliquent pas sur les applications composées de service dynamiques. En effet, celles-ci ne sont pas décrites comme un ensemble d'instances, mais comme un ensemble de services importés, instanciés, d'instances de composant … Il n'est donc pas possible de manipuler directement la structure de l'application. Cependant, il est possible de l'adapter en reconfigurant les dépendances de service. Ainsi, iPOJO propose des fonctionnalités afin de reconfigurer dynamiquement les dépendances de service. Ces reconfigurations ont un effet immédiat sur la structure de l'application. Le principal avantage de cette méthode est qu'elle évite de tisser des liens forts entre l'application et des types de composant ou fournisseurs de service spécifiques. Lorsque ce lien existe (parce qu'il est décrit dans l'application) alors il est possible de modifier la reconfiguration de l'instance. Cette reconfiguration peut influencer les propriétés des services publiés par l'instance. Ces deux opérations de reconfiguration dynamique offrent un très large éventail de possibilités tout en ne compromettant pas le dynamisme de l'application.

Enfin, ce chapitre a présenté les mécanismes d'extension proposés par iPOJO. iPOJO se focalise principalement sur le dynamisme qui n'est pas le seul besoin non-fonctionnel. Afin de permettre l'ajout d'autres besoins et donc d'étendre le modèle, iPOJO fournit un mécanisme d'extension. En fait, toutes les fonctionnalités d'iPOJO sont fournies de cette façon. Ces extensions peuvent être délivrées séparément du cœur d'iPOJO et sont développées avec le même modèle de développement que les composants atomiques (en Java). Ainsi, ces handlers peuvent profiter des fonctionnalités déjà fournies telles que la gestion du dynamisme ou la reconfiguration. Grâce à ce mécanisme, un large spectre de besoins non-fonctionnels peut être rajouté. Ceux-ci peuvent également influencer le modèle de composition et y rajouter des contraintes spécifiques à un domaine ou à une application en particulier.

Ce chapitre conclut la présentation du modèle à composant à service proposé dans cette thèse. Celui-ci fournit toutes les fonctionnalités définies dans le SOA dynamique étendu. La suite de cette thèse présentera l'implémentation d'iPOJO ainsi que son utilisation dans trois domaines d'application différents.

Troisième partie

Implémentation & Résultats

Chapitre 9
Implémentation

La précédente partie de cette thèse a décrit une approche afin de concevoir, implémenter et gérer les applications dynamiques. Cette thèse propose un nouveau modèle à composant à service fournissant un modèle de composition et de développement supportant et masquant le dynamisme, ainsi que des fonctionnalités d'introspection, de reconfiguration et d'extension. Cette partie de ce manuscrit décrit l'implémentation et valide cette approche dans différents contextes d'utilisation.

L'implémentation actuelle d'iPOJO[36] est hébergée dans le projet Apache Felix[37], une implémentation open-source d'OSGi™, et s'exécute au-dessus d'une plate-forme OSGi™ R4. Elle est disponible en open source sous licence Apache 2.0[38]. Bien qu'hébergée dans le projet Felix, l'implémentation d'iPOJO fonctionne également sous les autres implémentations d'OSGi™ telles qu'Eclipse Equinox ou Gatespace Telematics Knopflerfish. De plus, cette implémentation supporte l'intégralité des fonctionnalités décrites auparavant.

Ce chapitre décrit l'implémentation d'iPOJO et comment sont implantées les différentes fonctionnalités proposées dans cette thèse. Après une présentation d'OSGi et l'explication de ce choix comme plate-forme sous-jacente, ce chapitre décrit le principe de fonctionnement d'iPOJO. Ensuite, ce chapitre se focalise sur le support des composants atomiques implémentés en Java et le support des compositions. Cette description sera suivie d'une présentation des mécanismes d'extension. Ce chapitre se conclura en donnant quelques chiffres sur l'implémentation actuelle.

[36] http://felix.apache.org/site/apache-felix-ipojo.html
[37] http://felix.apache.org/site/index.html
[38] http://felix.apache.org/site/license.html

1. La plate-forme à service OSGi™

La plate-forme de service OSGi™ est spécifiée par le consortium OSGi™ Alliance. À l'origine (mai 2000), cette spécification ciblait les plates-formes résidentielles et industrielles. Cette spécification ne définissait que les mécanismes de déploiement et quelques services de base (tels que le service de journalisation ou d'accès par HTTP). C'est avec la version 4, parue en octobre 2005, qu'OSGi™ est devenu très populaire. En effet, dans cette version, OSGi™ définit comment créer, déployer et administrer des applications modulaires. Aujourd'hui, OSGi™ est utilisé dans de très nombreux domaines tels que dans les téléphones portables, les véhicules, la gestion de l'énergie, les PDA, les réseaux en grille, la gestion de flottes, ainsi que les plates-formes de divertissement telles qu'iPronto.

OSGi™ version 4.1 (parue en mai 2007) définit actuellement une plate-forme de déploiement dynamique (et continu). Cette particularité n'est pas supportée par défaut par la plate-forme Java (sur laquelle repose OSGi™). Ainsi, les applications sont conditionnées en « *bundles* » (unité de conditionnement et de déploiement utilisé par OSGi™) et sont ensuite installées, démarrées, mises à jour, arrêtées et désinstallées sans redémarrer la plate-forme OSGi™. Les *bundles* ont la capacité d'importer et d'exporter des paquetages. Ceci permet au code d'une application d'être divisé en plusieurs *bundles* important et exportant les paquetages nécessaires et mis à disposition. Ces *bundles* peuvent ensuite évoluer indépendamment. Les liaisons d'import et d'export de paquetages entre *bundles* sont gérées par la plate-forme.

Au-dessus de cette plate-forme de déploiement est défini un environnement à service permettant de faire communiquer les différentes applications et « morceaux » d'application entre eux de manière flexible et dynamique. Cependant, ces interactions doivent être gérées par le développeur. Ce code est malheureusement délicat, superflu, et demande une grande connaissance d'OSGi™ afin d'éviter les erreurs.

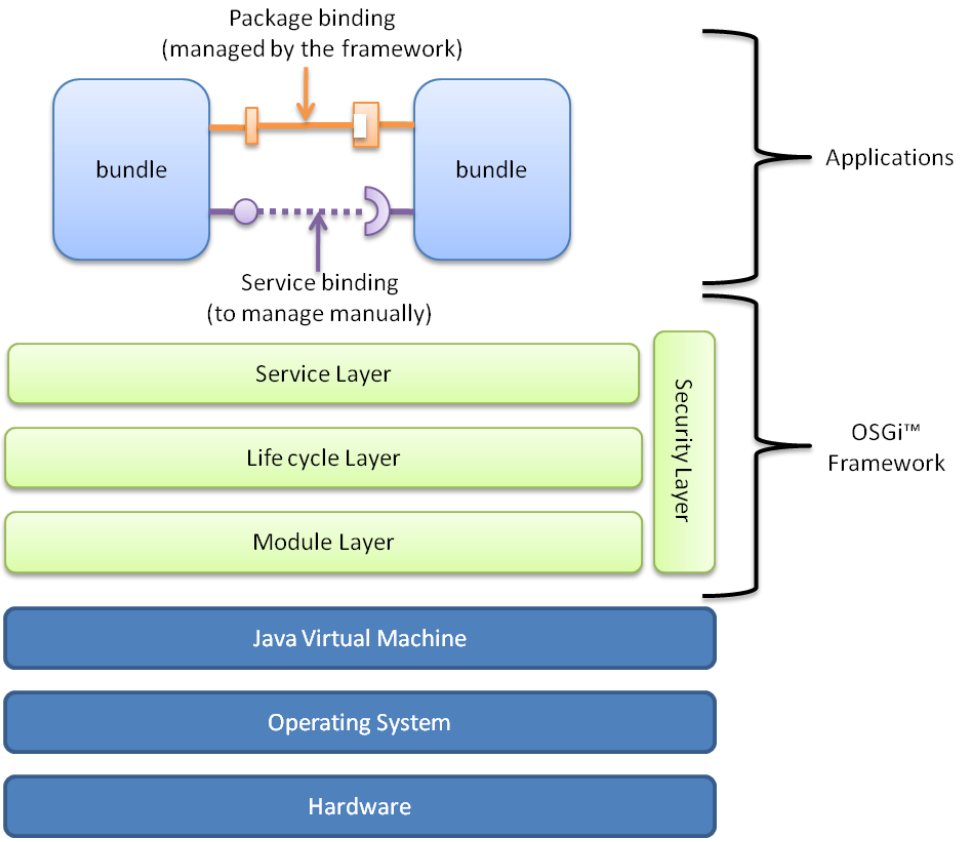


Figure 105. Architecture d'une plate-forme OSGi™

Une plate-forme OSGi™ est découpée en plusieurs couches (Figure 105) :

- La couche de modularité définissant les mécanismes de base permettant la mise en place d'applications modulaires
- La couche gérant le cycle de vie des modules et définissant comment sont gérés les modules et comment ils sont installés, démarrés, arrêtés, mis à jour et désinstallés.
- La couche service définissant l'architecture à service présenté dans le chapitre 4
- La couche de sécurité fournissant les mécanismes afin de protéger et de sécuriser les applications s'exécutant sur la plate-forme.

Plusieurs critères nous ont poussés à choisir OSGi™ comme la plate-forme sous-jacente à iPOJO. Tout d'abord, le large spectre d'utilisation ainsi que l'engouement actuel pour OSGi™ nous ont semblés intéressants. Il s'avérait particulièrement attirant de montrer comment le dynamisme intervenait dans des environnements habituels tels que les plates-formes résidentielles, dans des environnements mobiles comme les téléphones portables, mais également dans des environnements plus surprenants tels que les serveurs d'applications.

Ensuite, OSGi™ définit une plate-forme de déploiement supportant la mise à jour de *bundles*. Cette propriété est très importante, car elle permet de faire évoluer dynamiquement des applications.

Enfin, OSGi™ définit une architecture à service dynamique permettant de créer des applications supportant le dynamisme. En effet, tel que présenté dans le chapitre 4, OSGi™ propose toutes les capacités nécessaires à la mise en place d'applications à service dynamiques. Cependant, les mécanismes définis par OSGi™ sont très délicats à utiliser. Ainsi, le modèle de développement, promu par iPOJO, a un véritable intérêt sur cette plate-forme.

2. Principes de fonctionnement

L'implémentation actuelle d'iPOJO repose sur une plate-forme OSGi™ version 4.1. Ainsi, le canevas d'exécution contenant les mécanismes d'iPOJO est déployé au-dessus d'OSGi™ (Figure 106).

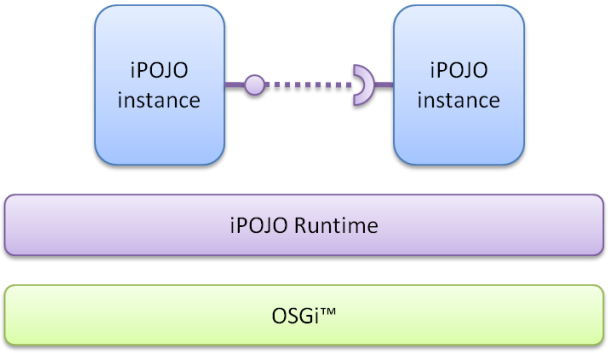


Figure 106. La plate-forme d'exécution d'iPOJO repose sur une plate-forme OSGi™

Le canevas d'exécution d'iPOJO est lui-même divisé en plusieurs *bundles*. Un *bundle* cœur contient tous les mécanismes de base et la gestion des composants atomiques Java. Le modèle de composition est délivré dans un second *bundle*. Les handlers externes sont également distribués dans des *bundles* différents. Cette division a pour but de facilement construire l'environnement iPOJO désiré, tout en limitant la taille de l'infrastructure d'exécution.

Les types de composants et les instances sont également conditionnés dans des *bundles*. Cependant, lors de leur installation, ceux-ci seront détectés par iPOJO qui prendra en charge leur exécution. Ainsi, iPOJO pourra analyser le contenu de ces *bundles*, créer les fabriques et gérer les contextes de service. La suite de cette section décrit comment ces concepts de base ont été implémentés.

2.1. Analyse des *bundles*

Afin de découvrir les types de composant et les instances, iPOJO analyse chaque *bundle* déjà installé et activé ainsi que tous les *bundles* arrivant dynamiquement. Pour cela, iPOJO regarde si le manifeste de ces *bundles* contient une entrée spécifique contenant les métadonnées d'iPOJO. Les différents types de composant et les instances à créer sont détectés de cette manière.

Afin de ne pas bloquer le fil d'exécution de l'installation des *bundles* (qui appartient à la plate-forme OSGi™), les *bundles* détectés sont ensuite mis dans une file d'attente. Cette file d'attente est traitée par un autre fil d'exécution (géré par iPOJO).

2.2. Création et Gestion des fabriques

Lorsqu'un type de composant est déclaré dans un *bundle*, iPOJO crée la fabrique associée à ce type de composant. Cette création analyse le contenu de la description afin de détecter les erreurs éventuelles. Aucune instance n'est créée par défaut.

```
public interface Factory {

        ComponentInstancecreateComponentInstance(Dictionary configuration);

        ComponentInstancecreateComponentInstance(Dictionary configuration, ServiceContext serviceContext);

        ComponentTypeDescription getComponentDescription();

        void reconfigure(Dictionary conf);

        void addFactoryStateListener(FactoryStateListener listener);

        void removeFactoryStateListener(FactoryStateListener listener);

        int getState();
}
```

Figure 107. Extrait de l'interface Factory

Une fabrique peut être valide ou invalide suivant la disponibilité des handlers décrits dans le type de composant. En effet, un lien est fait entre la description du type et les handlers requis. Tant que tous les handlers ne sont pas disponibles, la fabrique n'est pas utilisable et aucune instance ne peut être créée. Lorsqu'un handler requis disparaît, toutes les instances créées précédemment sont détruites et ne seront recréées que lorsque la fabrique redeviendra valide.

Si le type est déclaré comme « public », alors un service est exposé dans le service de registre d'OSGi™ avec l'interface de service ci-dessus. Ce service servira entre autres à créer des instances de ce type à partir d'autres *bundles*. Ce service permet à la fois la création d'instance, mais également d'obtenir des informations sur le type de composant et sur l'état actuel de la fabrique.

Il est important de noter que l'interface de service possède deux méthodes permettant de créer des instances. Une des méthodes permet de spécifier le contexte de service dans lequel s'exécutera l'instance.

2.3. Contexte de service

Le concept de contexte de service est très important dans iPOJO. Ces contextes permettent d'isoler des espaces de service. Deux des contraintes fortes dans l'implémentation d'iPOJO furent les suivantes :

- Un contexte global d'OSGi doit pouvoir être considéré comme un contexte de service (celui-ci est généralement appelé contexte de service global).

- Une instance de service ne doit pas s'apercevoir du fait qu'elle est exécutée dans un contexte de service ou dans le contexte de service global.

Afin de remplir ces contraintes, iPOJO ré-implémente les mêmes mécanismes qu'OSGi™ (Figure 108). De ce fait, iPOJO manipulera tout le temps des références de service et des enregistrements de service (service registration) provenant soit du contexte global soit d'un contexte particulier. De plus, iPOJO fournit un annuaire de service, ayant le même comportement que celui d'OSGi™, mais pouvant être instancié plusieurs fois.

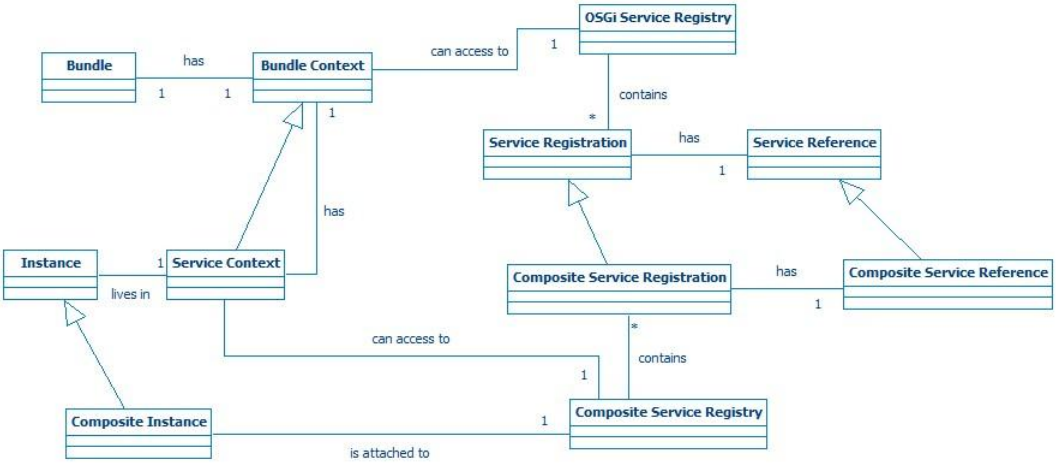


Figure 108. Relation entre contexte de service (*service context*) et *bundle context*

Lorsqu'un nouveau contexte de service est nécessaire, iPOJO instancie un nouvel annuaire de service (Composite Service Registry). Le contexte de service créé aura accès à cet annuaire. Cet annuaire suit le même comportement que l'annuaire d'OSGi™ (OSGi Service Registry), c'est-à-dire contient des enregistrements de service (Composite Service Registration) et permet la découverte de référence de service (Composite Service Reference). Cependant, les implémentations de ces classes sont liées à iPOJO et contiennent des informations cruciales au maintien de la cohérence de ces annuaires.

2.4. Instances

Les instances de composants sont également décrites dans les métadonnées contenues dans le *bundle*. Lorsqu'une déclaration d'instance est effectuée, iPOJO ajoute cette demande d'instanciation dans le créateur d'instance. Celui-ci a pour but de chercher la fabrique requise. Cette recherche se fait dans le *bundle* d'où provient l'instance (et a ainsi accès aux fabriques privées) puis interroge les différentes fabriques exposées dans l'annuaire de service d'OSGi™. Dès lors que la fabrique est disponible et valide, l'instance est créée.

Si, une instance créée est détruite suite à l'invalidation de la fabrique, le créateur d'instance attend la revalidation de la fabrique afin de recréer une nouvelle instance. Les instances sont détruites et retirées du créateur d'instance lorsque le *bundle* déclarant l'instance est arrêté ou désinstallé.

3. Support des composants atomiques

La section précédente a décrit globalement comment étaient implémentés les concepts de base d'iPOJO. Cette section s'attarde sur les composants atomiques implémentés en Java. Plusieurs points seront abordés tels que la machine d'injection et d'introspection (dont les fonctionnalités ont été décrites auparavant dans cette thèse), la gestion du dynamisme et de la synchronisation ainsi que les autres fonctionnalités fournies.

3.1. Implémentation de la machine d'injection

Afin de proposer un modèle de développement à la fois simple et puissant, cette thèse a décrit les fonctionnalités que devait fournir la chaine d'injection et d'introspection sous-jacente. Cette machine a été implémentée pour Java sous la forme d'une manipulation de bytecode.

En effet, le manipulateur inclus dans iPOJO transforme une classe en une classe manipulée permettant l'interception des méthodes et des accès aux membres (champs de la classe) . De plus cette manipulation a la particularité de lier la classe manipulée au conteneur d'iPOJO en y ajoutant des méthodes d'initialisation. Ainsi, le conteneur pourra injecter des valeurs dans des membres, en connaitre la valeur actuelle, mais pourra également invoquer des méthodes. Plusieurs contraintes sont apparues durant le développement de cette machine : l'extensibilité et la capture de référence.

Tout d'abord, afin de supporter l'extensibilité, cette manipulation est générique. Ainsi pour une classe donnée, la classe obtenue après la manipulation sera la même. Les métadonnées ne sont pas examinées durant la manipulation. Bien que ce type de manipulation est considéré comme moins performant qu'une manipulation spécifique, elle a pour avantage d'abstraire la manipulation (les développeurs de handler n'ayant pas à la spécifier). De plus, le dynamisme ayant un impact non négligeable sur le code, le surcoût dû à cette manipulation est négligeable.

Enfin, il est crucial que la classe manipulée ne détienne pas de référence directe sur un objet de service. En effet, si cette classe se retrouvait avec une référence sur un objet de service, le fournisseur ne pourrait pas être déchargé intégralement lors de son départ. Ce problème est dû à la politique de déchargement de classe de la machine virtuelle Java. Ainsi, lorsqu'un membre de la classe est géré par le conteneur, ce champ ne reçoit jamais la référence sur l'objet injecté. Celui-ci n'est jamais assigné.

```
public class ClientImpl {

    private Service m_service;

    public void doSomething() {
                System.out.println(m_service.getMessage());
    }

}
```

```
public class ClientImpl implements Pojo {

    private Service __getm_service() {
                if(!__Fm_service) return m_service;
                else return (Service) __IM.onGet(this, "m_service");
    }
    private void __setm_service(Service service) {
                if(!__Fm_service) { m_service= service; }
                else {  __IM.onSet(this, "m_service", service); }
    }

    public ClientImpl() { this(null); }
    private ClientImpl(InstanceManager _manager) {  _setInstanceManager(_manager); }

    public void doSomething() {
                if(!__MdoSomething) {
                            __doSomething(); return;
                }
                try{
                            __IM.onEntry(this, "doSomething", new Object[0]);
                            __doSomething();
                            __IM.onExit(this, "doSomething", null);
                } catch (Throwable throwable) {
                            __IM.onError(this, "doSomething", throwable);
                            throw throwable;
                }
    }
    private void __doSomething() { System.out.println(__getm_service().getMessage());
     }

    private void  _setInstanceManager(InstanceManagerinstancemanager) { … }
    public ComponentInstance getComponentInstance() { return__IM; }

    private InstanceManager__IM;
    private boolean__Fm_service;
    private Service m_service;
    private boolean__MdoSomething;
}
```

Figure 109. Illustration de la manipulation. La classe originale (à gauche) devient la classe manipulée (à droite)

La figure ci-dessus montre un exemple de classe manipulée par rapport à une classe d'origine. Remarquons que les accès aux membres de la classe sont remplacés par des appels de méthodes (générés). Le

nom des méthodes d'origines est préfixé par « __ ». De plus, les méthodes sont encapsulées afin de permettre l'interception de l'entrée, de la sortie ainsi que des erreurs potentielles. La manipulation des méthodes prend soin de ne pas modifier les numéros de lignes (si la classe d'origine les contient) afin de ne pas influer sur les informations de débogage. La manipulation des membres permet au conteneur de toujours retourner une valeur à jour. Un mécanisme de drapeau est mis en place pour réduire le coût de délégation lorsque le conteneur ne surveille pas un membre ou une méthode (membres commençants par _F et _M). Il faut également noter que les membres, méthodes et blocs d'initialisation statiques ne sont pas manipulés.

La machine d'interception et d'introspection décrite dans cette thèse doit également fournir des fonctionnalités permettant d'appeler des méthodes. Ces appels sont effectués grâce à l'API de réflexion fournie par Java. Cependant, afin de masquer certaines complexités dues à l'utilisation de cette API dans un environnement dynamique, ces appels sont effectués grâce à une classe utilitaire (fournie par iPOJO) permettant de configurer des appels de méthodes et d'appeler ces méthodes sur les objets sous-jacents. Ainsi, les problèmes de synchronisation, d'appel sur plusieurs objets sont gérés par cette classe.

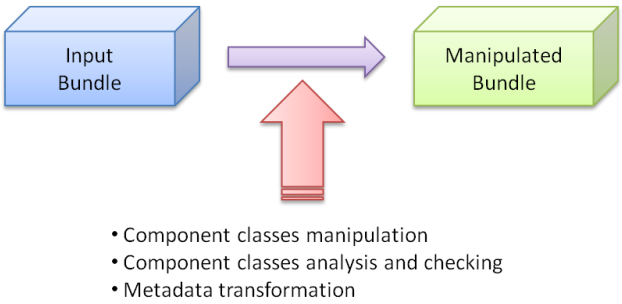


Figure 110. Conditionnement de composants iPOJO

Le manipulateur est implémenté à l'aide de la bibliothèque OW2 ASM [194, 195] qui permet de lire et de manipuler des classes Java. Cette manipulation est effectuée durant la phase de conditionnement du composant (Figure 110). Il aurait été possible d'effectuer cette manipulation lors du chargement. Cependant, OSGi™ ne définit pas de mécanismes permettant d'effectuer cette manipulation sans enfreindre les règles de sécurité. Ce type de mécanisme sera très probablement fourni dans les prochaines versions d'OSGi™. De plus, lors du conditionnement, la manipulation permet d'anticiper certains problèmes ainsi que de remonter des informations sur l'implémentation des composants. Ces informations servent principalement à retarder le chargement de classe à l'exécution (en effet, ceci évite généralement d'utiliser la réflexion simplement pour obtenir des informations sur la classe). Afin d'éviter l'utilisation d'une bibliothèque XML, les métadonnées sont transformées dans une syntaxe interne et sont ajoutées au manifeste du *bundle*.

Il existe aujourd'hui trois interfaces permettant de manipuler les composants Java. Un plug-in Eclipse permet de conditionner des *bundles* contenant des composants iPOJO directement à partir de l'IDE Eclipse. Une tâche *Ant* et un plug-in *Maven* permettent également d'intégrer des *bundles* iPOJO dans des processus de construction de projets plus complexes.

3.2. Gestion du dynamisme

Au-dessus de la machine d'injection décrite dans la section précédente, plusieurs handlers ont pour but de gérer le dynamisme, et particulièrement les interactions avec l'annuaire de service. Deux handlers sont particulièrement impliqués dans cette tâche : le handler de dépendance de service et le handler de fourniture de service (Figure 111).

Le handler de dépendance de service gère les services requis, c'est-à-dire la recherche et la sélection de fournisseurs de service, le maintien de la liaison (en fonction des politiques de liaison définies), ainsi que l'injection des objets de services (par l'intermédiaire de méthodes ou de membres de la classe d'implémentation du composant). Ce handler implémente l'intégralité du modèle de dépendance proposé dans

cette thèse. Ainsi, une fois configuré et démarré, il traque les fournisseurs de service et sélectionne les fournisseurs. Il gère également la liaison. Lorsque cela est nécessaire, ce handler invalide l'instance (absence de fournisseurs, liaison cassée). Ce handler permet également de récupérer des informations sur les fournisseurs utilisés. Cette fonctionnalité est nécessaire durant l'introspection d'un système.

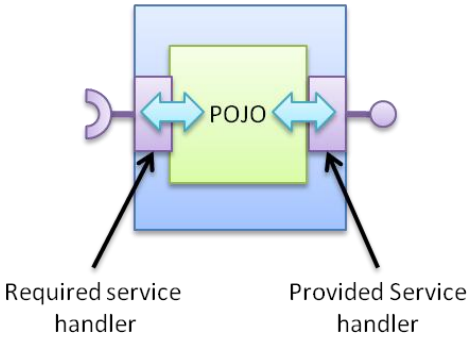


Figure 111. Les fonctionnalités d'iPOJO sont également développées sous la forme de handler

Le handler de fourniture de service a pour but de publier et de fournir des services. Celui-ci gère alors l'intégralité des interactions avec l'annuaire de service afin de gérer l'enregistrement de service. De plus, ce handler supervise des membres de la classe attachés à des propriétés de service afin de répercuter les éventuels changements de valeur.

Un aspect délicat dans l'implémentation de ces handlers est le support de la synchronisation et des reconfigurations. En effet, sachant que plusieurs fils d'exécution peuvent s'exécuter à l'intérieur d'une instance, et que les arrivées et départs de service utilisent également d'autres fils, il est nécessaire de gérer correctement tous ces paramètres. Ceci est crucial afin de reconfigurer les instances proprement. L'implémentation d'iPOJO n'attend pas la quiescence afin d'exécuter une reconfiguration. En effet, la politique d'iPOJO effectue les reconfigurations immédiatement. L'utilisation des caches associés aux différents fils d'exécution permet de garantir la cohérence du système. Ainsi, l'implémentation d'iPOJO utilise les concepts de confinement sur pile et de confinement par thread [184] afin d'effectuer les reconfigurations.

3.3. Autres handlers fournis

L'implémentation actuelle d'iPOJO fournit d'autres handlers. Le cœur d'iPOJO est livré avec un handler de cycle de vie (permettant de créer et de notifier des objets contenus lorsque l'instance devient valide ou invalide), un handler d'architecture utilisé lors de l'introspection du système, une handler de configuration permettant de configurer et de reconfigurer des instances dynamiquement, et un handler de contrôle de cycle de vie permettant au contenu d'une instance de participer au cycle de vie de l'instance.

De plus, plusieurs handlers externes ont également été développés. Ainsi, un handler d'administration par JMX permet d'administrer à distance des instances. Des handlers permettent également d'automatiser l'émission et la réception d'évènements. Un handler de persistance a également été développé et permet de stocker et de recharger l'état d'instance sur un support persistant. D'autres handlers permettent l'automatisation d'interactions complexes utilisant les motifs de conceptions de tableau blanc [196], ou d'extension.

D'autres handlers plus spécifiques ont également été développés afin d'ajouter de la qualité de service, ou des fonctionnalités autonomiques [188].

4. Support des composants composites

La gestion des composants composites est basée sur la notion de contexte de service. Lors de la création d'une instance de composite, un nouveau contexte de service (c'est-à-dire un nouveau registre de service) est créé et attaché au composite. Cependant, le conteneur du composite doit alors gérer à la fois le contexte de service interne (celui venant d'être crée) et le contexte de service parent dans lequel l'instance vit. De plus, le conteneur d'une composition est également composé de handlers. Ces handlers reprennent les concepts des handlers des composants atomiques, mais ont des spécificités liées à leur statut de handler de composite.

Cette section décrit les principes de l'implémentation des compositions structurelles de services et plus exactement comment sont gérés les sous-services et les services publiés.

4.1. Gestion des sous-services

Comme décrit auparavant dans cette thèse, iPOJO supporte deux types de sous-services, mais sont gérés par le même handler (Figure 112):

- Les services importés
- Les services instanciés

Les services importés sont résolus dans le contexte de service « parent ». Ainsi, un fournisseur existant est traqué. Lorsque celui-ci est trouvé alors, le composite republie ce service dans le contexte de service interne. Les propriétés de service attaché à la publication sont également republiées. Seules les propriétés liées au contexte de service (comme l'identificateur d'enregistrement) sont modifiées. Bien qu'il s'agisse d'une republication, les instances internes auront un accès direct (et non pas derrière un proxy) au fournisseur de service. Pour les instances vivant dans le composite, l'accès à ce service n'a aucune différence avec l'accès à un service réellement existant dans le composite. De la même manière, le fournisseur ne perçoit aucune différence lorsqu'il est utilisé par une instance interne au composite. Cependant, lors de l'introspection du système, le fournisseur déclarera l'instance du composite comme consommatrice alors qu'en fait il s'agit d'une instance interne à cette composition.

Les services importés subissent les arrivées et départs des fournisseurs ciblés. Ainsi, le composite doit réagir à ces évènements. De plus, ces dépendances peuvent être reconfigurées ou être contextuelles et donc subir l'influence de sources de contexte. À chaque nouveau contexte ainsi qu'à chaque reconfiguration, l'ensemble des services importés est réévalué afin de calculer les fournisseurs qui sont toujours compatibles, ceux qu'ils ne le sont plus et ceux qu'ils le deviennent.

La gestion des services instanciés repose principalement sur les concepts de fabriques. En effet, les fabriques ont la particularité d'exposer en propriété les services fournis par les instances de cette fabrique. Le composite peut choisir les fabriques à utiliser afin d'instancier un service particulier dans le contexte de service interne. Cette instanciation s'effectue comme une instanciation normale. Cependant, le contexte de service interne à la composition est donné en paramètre à la création ce qui permet d'attacher l'instance à ce contexte de service. Une fois instanciée, l'instance réagit comme une instance « normale » et n'est pas au courant qu'elle s'exécute au sein d'un composite. Cependant, les services requis (sauf les dépendances d'implémentation qui sont résolues différemment) sont résolus dans le contexte de service du composite.

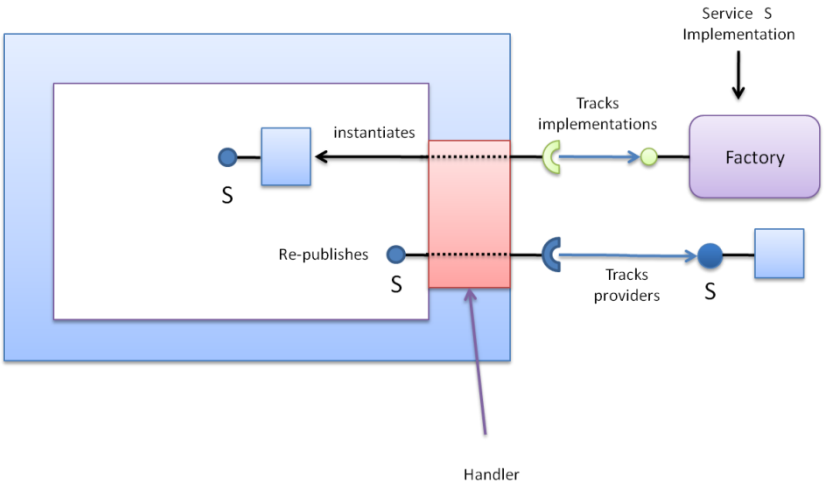


Figure 112. Fonctionnement des sous-services dans les compositions

Comme les instances de services sont créées à l'aide de fabriques, le composite doit traquer la disponibilité de ces fabriques. En effet, celles-ci peuvent apparaître, disparaître ou devenir valides ou invalides. Remarquons que finalement, les instances de service sont transformées en dépendance de service. Cependant au lieu de cibler un fournisseur (comme pour les services importés), ces dépendances ciblent des implémentations, c'est-à-dire des fabriques. À chaque changement, le composite doit réagir et si nécessaire changer de fabriques. Ce changement s'effectue de la manière suivante :

- L'état de l'instance est stocké si la spécification de service définit un état
- L'instance est détruite
- Une nouvelle instance est créée avec une autre fabrique (si une autre fabrique compatible est disponible)
- L'état est réinjecté (s'il a été stocké)

Ce processus peut également avoir lieu lorsque la dépendance de service est reconfigurée (lors d'un changement de contexte pour les dépendances contextuelles). Lorsqu'une instanciation est déclarée agrégée, toutes les fabriques disponibles sont utilisées. De plus, une instance de service peut elle-même être une composition, ce qui rend le modèle hiérarchique. Cependant, l'implémentation d'iPOJO prend soin d'éviter les cycles éventuels.

4.2. Gestion des services fournis

L'implémentation actuelle d'iPOJO supporte les deux types exportés présentés dans cette thèse. Ces deux types sont gérés par le même handler. Dans les deux cas, le type résultant doit être une implémentation correcte des services publiés. Le premier type de service exporté est une simple exportation. Il s'agit d'une importation inversée. Au lieu de chercher un fournisseur dans le contexte de service parent pour le publier dans le contexte de service interne, un fournisseur est recherché dans le contexte de service interne pour l'exporter dans le contexte de service parent (Figure 113). Tout comme pour l'importation, il donne accès au fournisseur et ne publie pas un proxy. Comme l'exportation cible un service présent dans le composite, celui-ci doit gérer l'apparition ou la disparition de ce service au sein du composite.

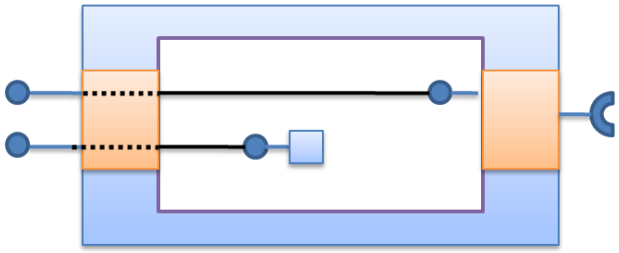


Figure 113. Principes de l'exportation de service

L'implémentation de service est plus complexe (Figure 114). En effet, iPOJO doit découvrir la politique de délégation permettant de fournir le service. Pour cela, iPOJO choisit, sur quelle entité (service ou instance) chaque méthode doit être déléguée. Une fois la politique de délégation décidée, une classe est générée à la volée. En fait, un composant atomique complet (classe d'implémentation et métadonnées) est généré. Une fois le type et la fabrique associés créés, une instance est créée grâce à la fabrique. Cette instance sera utilisée pour fournir les services implémentés. Les services exportés par cette instance seront exportés dans le contexte de service parent.

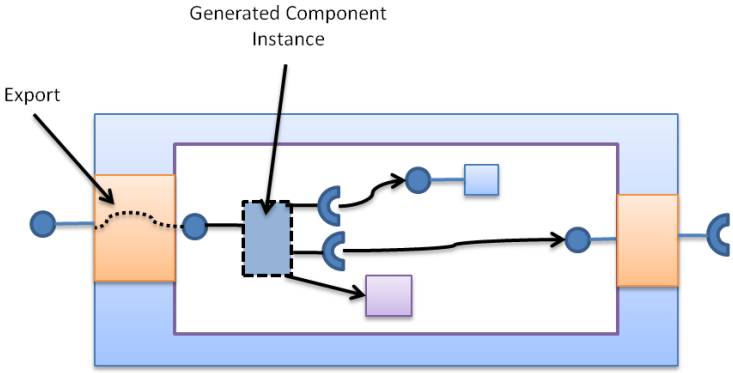


Figure 114. Implémentation de service par un composite

Comme l'implémentation de service repose sur un composant atomique, le dynamisme est géré par cette instance. Ainsi, les services et les instances sur lesquels les méthodes doivent être déléguées sont associés à des dépendances de service ou à des propriétés de configuration. Les propriétés de configuration permettent d'injecter des instances internes afin qu'elles puissent participer à l'implémentation de service. Ainsi, il est possible de créer des implémentations de service sophistiquées. Dès lors que cette instance est invalide, le service n'est plus exporté, et la composition devient invalide.

Lorsque cette instance dépend d'un service optionnel, certaines méthodes peuvent ne pas s'exécuter correctement. Dans ce cas, il a été choisi de lancer une UnsupportedOperationException[39] lorsque la méthode est appelée tant que ce service n'est pas présent.

Lorsqu'une méthode est déléguée sur une dépendance de service agrégée, deux politiques de délégation sont supportées. La méthode peut être déléguée sur chacun des fournisseurs (seulement pour les méthodes ne retournant pas de résultat) ou peut-être délégué sur un seul (politique par défaut).

4.3. Reconfiguration de l'assemblage

Comme présentés précédemment dans cette section, de nombreux évènements peuvent déclencher une reconfiguration d'une composition. Afin de gérer correctement l'état interne, iPOJO gèle l'accès aux instances et aux services impliqués dans une reconfiguration.

[39] http://java.sun.com/j2se/1.3/docs/api/java/lang/UnsupportedOperationException.html

De plus, dès qu'une instance ou un service permet la gestion de l'état, celui-ci est systématiquement sauvegardé et restauré durant une reconfiguration. Cette sauvegarde s'effectue soit en mémoire, soit dans un service de persistance si un de ces services est disponible. Actuellement, ce service est recherché dans le contexte global.

5. Extensibilité et handlers externes

Tel que présenté dans le chapitre précédent, il est devenu très important pour une plate-forme d'exécution de permettre son adaptation facile à de nouveaux domaines et de permettre l'ajout de gestionnaires traitants de nouveaux besoins non fonctionnels. L'implémentation d'iPOJO supporte cette extensibilité à la fois pour les composants atomiques et composites. Cette section décrit comment sont implémentés les mécanismes permettant cette extensibilité.

5.1. Handler pour composants atomiques et composites

Les conteneurs des instances de composants iPOJO sont composés de handlers qui créent une « barrière » entre le contenu de l'instance et l'extérieur. Cependant, ces handlers sont cependant différents s'ils s'adressent aux composants atomiques ou composites. Ces handlers ont tout de même des points communs et ont la même classe parente (Figure 115). Cette classe définit les principes généraux des handlers et les traitements communs à tous les handlers (processus d'attachement à une instance…). Les méthodes liées au cycle de vie des handlers sont également définies dans cette classe (dont une méthode appelée lors d'une reconfiguration dynamique).

```
public abstract class Handler {

    public abstract Handler getHandler(String name);

    public final boolean isValid() { /* … */}

    public final void setValidity(boolean isValid) { /* ..*/ }

    public final boolean getValidity() {  return m_isValid;  }

    public void initializeComponentFactory(ComponentTypeDescription typeDesc, Element metadata) throws ConfigurationException {
        // The default implementation does nothing.
    }

    public abstract void configure(Element metadata, Dictionary configuration) throws ConfigurationException;

   public abstract void stop();

   public abstract void start();

   public void stateChanged(int state) {
        // The default implementation does nothing.
    }

   public HandlerDescription getDescription() {
        return new HandlerDescription(this);
    }

   public void reconfigure(Dictionary configuration) {
        // The default implementation does nothing.
    }
}
```

Figure 115. Extrait de la classe Handler

Les handlers suivent tous le même cycle de vie, ont accès au conteneur et au contenu de l'instance, peuvent demander son invalidation, et peuvent participer à l'introspection. Un handler peut également collaborer avec un autre handler attaché à sur la même instance. L'implémentation actuelle ne permet pas la suppression et l'ajout de handler lorsqu'une instance est déjà créée. Ces ajouts doivent être faits avant le démarrage de l'instance.

La classe Handler est raffinée pour les handlers de composants atomiques et pour les handlers de composants composites. En effet, ces deux types de handlers ont des fonctionnalités différentes :

- Un handler de composants atomiques a la possibilité d'intercepter les accès aux membres de la classe d'implémentation du composant ainsi qu'aux méthodes de cette classe.
- Un handler de composant primitif a accès au contexte de service interne à la composition. Cependant, il n'a pas accès aux membres et aux méthodes vu qu'il ne contient pas d'implémentation « concrète ».

5.2. Développement de handler

Afin d'étendre le modèle fourni par iPOJO, il est possible de développer son propre handler. Un handler est principalement constitué de la classe implémentant une des deux sous-classes d'Handler. Cette classe contiendra l'analyse de la configuration, la vérification de cette configuration, le cycle de vie du handler et la logique-métier. Cependant, afin de simplifier ce développement, un handler est lui-même un composant iPOJO (atomique) et peut donc réutiliser n'importe quel autre handler déjà développé.

Une fois les classes du handler développées, celui-ci est décrit comme un composant atomique. Cependant afin de caractériser le handler, cette description s'effectue grâce au mot-clé « handler » et possède les attributs « name » et « namespace ». Ces attributs définissent l'élément géré par le handler dans une description de type de composant. Ils permettent également à iPOJO d'associer les éléments d'une description de composant à un handler particulier. Ainsi, iPOJO analyse la description d'un type de composant et peut traquer les handlers nécessaires. Les liens entre une instance et les handlers sont donc fortement couplés. Ce choix a été fait afin d'éviter les incohérences. Un troisième attribut « type » définit si le handler est un handler de composant atomique ou composite. Le reste de la description d'un handler est identique à la description d'un type de composant atomique (Figure 116).

```
<handler
  classname="org.apache.felix.ipojo.handlers.architecture.ArchitectureHandler"
  name="architecture"
  namespace= "org.apache.felix.ipojo">
  <provides interface="org.apache.felix.ipojo.architecture.Architecture">
    <property field="m_name" name="architecture.instance" value="" />
  </provides>
</handler>
```

Figure 116. Exemple de description de handler:
le handler d'architecture pour composant atomique Java

5.3. Déploiement et Exécution

Un handler est conditionné dans un *bundle* iPOJO. En effet, comme il s'agit d'un composant iPOJO « spécial », il suit le même procédé de conditionnement. Son déploiement n'est en rien différent de celui d'un *bundle* classique. Ce système propose une grande flexibilité. En effet, chaque plate-forme peut choisir les handlers à installer et si nécessaire les désinstaller dynamiquement. Ainsi, il est possible d'obtenir des serveurs d'applications dynamiques spécialisés simplement (Figure 117) [197].

Une fois déployé, le handler est découvert par iPOJO qui va créer une fabrique particulière pour ce handler. Cette fabrique sera utilisée dès qu'une instance a besoin du handler. Si cette instance est associée à un contexte de service, le handler est également associé à ce contexte.

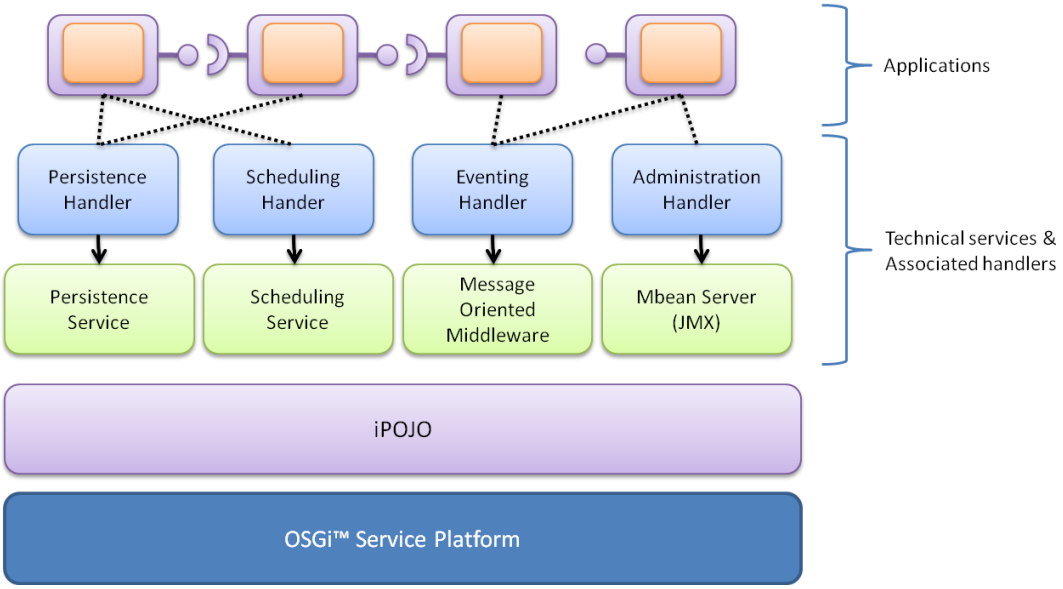


Figure 117. Exemple de personnalisation de la plate-forme iPOJO

Les instances de handlers sont détruites lorsque l'instance est détruite. Si un handler disparaît (désinstallation du *bundle* contenant le handler, invalidation de la fabrique du handler), l'instance est immédiatement détruite. Celle-ci sera recréée lorsque le handler est de nouveau disponible.

6. Quelques chiffres et Synthèse

Ce chapitre a présenté quelques points importants de l'implémentation d'iPOJO. Cette dernière section donne quelques chiffres concernant cette implémentation.

Tout d'abord, le projet iPOJO est divisé en 12 sous-projets (Tableau 30). Ces projets peuvent être classés en quatre catégories. Le tableau ci-dessous présente les différents projets par catégorie, le nombre de lignes de code de chacun des projets et le nombre de lignes de code des tests du projet.

Catégories	Projets	Nombre de lignes de code (environ)	Nombre de lignes de code des tests (environ)
Cœur	Plate-forme d'exécution	7500	30000
	Manipulateur	2350	
	Métadonnées	242	
	Annotations	105	
Modèle de composition	Composite	2900	8000
Outils	« Arch »	130	8500
	Plug-in *Maven*	70	
	Tache *Ant*	80	
	Support du déploiement	2400	
Handlers externes	Handler d'évènements	300	9500
	Dépendances temporelles	250	
	Handler de tableaux blancs	130	
	Handler d'Extensions	200	
	Handler d'administration (JMX)	670	

Tableau 30. Nombre de lignes de code projet par projet

Le cœur d'iPOJO est composé du projet contenant les classes manipulant les métadonnées internes à iPOJO, du manipulateur à la base de la machine d'injection et d'interception, et du projet principal contenant à la fois les principes de base d'iPOJO et les composants atomiques Java. Cet ensemble de projets implante le

canevas d'exécution d'iPOJO. Le *bundle* résultant de ces projets pèse 180 ko. Un des objectifs a été de minimiser cette taille afin de pouvoir l'utiliser dans des environnements contraints.

Le modèle de composition est implanté dans un projet séparé. Ce choix a été fait afin de diminuer la taille de la plate-forme d'exécution. En effet, le *bundle* contenant le modèle de composition à une taille d'environ 230 ko. Cette taille est principalement due à l'utilisation de la bibliothèque ASM qui est embarquée dans le *bundle*.

Plusieurs handlers externes ont également été développés afin d'étendre le modèle proposé par iPOJO. Ainsi, sont disponibles un handler permettant l'émission et la réception d'évènement, un handler d'administration utilisant JMX, un handler gérant des dépendances de service temporelles (bloquant le fil d'exécution lorsque le service n'est pas disponible), un handler gérant certains problèmes de chargement de classes (en fixant le chargeur de classe du fil d'exécution) ainsi que deux handlers automatisant la mise en place des motifs de conceptions d'extension et de tableau blanc.

Des outils ont été également développés. Deux sont particulièrement importants. Tout d'abord, « arch » est une commande permettant d'afficher l'état actuel du système. Cette commande est très utilisée lors du débogage d'un système dynamique, car donne un moyen très simple pour accéder à l'architecture actuelle du système. Ensuite, Junit4OSGi est un projet portant la plate-forme de test unitaire Junit sur OSGi™. Cette plate-forme permet alors de tester le fonctionnement de *bundles* au dessus d'OSGi™ en étendant les mécanismes de Junit afin de permettre l'accès aux contextes de service. De nombreuses méthodes utilitaires rendent l'écriture de tests plus aisés. Junit4OSGi utilise iPOJO afin de fonctionner, mais est également utilisé afin de tester iPOJO.

L'implémentation actuelle d'iPOJO fonctionne sur toutes les implémentations d'OSGi compatible avec la version 4). Apache Felix[40], Eclipse Equinox[41], et Gatespace Telematics Knopflerfish[42] sont testés. iPOJO est implémenté avec le profil J2ME Foundation [198]. De nombreuses machines virtuelles Java sont donc supportées telles que BEA JRockit, Mika, IBM J9, Google Dalvik (utilisé dans Android), et Apache Harmony. Les machines virtuelles Java de Sun sont bien entendu supportées.

[40] http://felix.apache.org/site/index.html
[41] http://www.eclipse.org/equinox/
[42] http://www.knopflerfish.org/kf2.html

Chapitre 10
Validation

Le chapitre précédent a décrit les principes d'implémentation d'iPOJO. Ce chapitre se focalise sur la validation de l'approche et de l'implémentation. Cette validation est effectuée en évaluant l'implémentation sur deux bancs d'essai mais également en décrivant comment iPOJO est employé dans trois contextes d'applications différents.

Le premier banc d'essai teste la performance de l'injection dans iPOJO. iPOJO sera comparé à différentes technologies d'injection et d'accès à une ressource. Ce test a pour but de mesurer et de positionner le surcoût en temps d'invocation impliqué par l'utilisation d'iPOJO.

Le deuxième banc d'essai teste le temps de démarrage d'une application à service de grande taille. Il s'agit d'un problème connu dans OSGi. En effet, OSGi tend à favoriser les mécanismes d'extension pour les applications de grandes tailles afin de diminuer le temps de démarrage. Ainsi, ce test vise à mesurer l'impact d'iPOJO lors du démarrage d'application à service de grande taille.

iPOJO est utilisé dans différents contextes. L'approche a été validée dans trois domaines d'applications :

- la plate-forme résidentielle,
- les serveurs d'applications,
- et les plates-formes pour téléphone mobile

La plate-forme résidentielle implémentée à l'aide d'iPOJO tend à simplifier le développement d'applications domotiques dynamiques. De plus, les contraintes du domaine ont poussé à étendre le modèle d'iPOJO afin de proposer un ensemble de services techniques et de handlers spécifique.

Le serveur JEE Jonas utilise OSGi et iPOJO afin de modulariser le cœur du serveur et de lui permettre la mise à jour dynamique de services techniques. Cette problématique est devenue cruciale dans le développement de serveur d'application afin de devenir plus flexible.

Enfin, le dernier domaine d'application concerne la téléphonie mobile. En effet, il devient important de proposer des intergiciels évolués afin de faciliter le développement d'applications pour téléphone mobile. iPOJO est utilisé dans OW2 uGasp. Ce projet propose un intergiciel pour le développement et l'exécution de jeux massivement multi-joueurs pour téléphone portable.

Ce chapitre sera organisé de la façon suivante. Tout d'abord, les performances d'iPOJO seront testées sur les deux bancs d'essai. Les résultats seront comparés par rapport aux autres plates-formes populaires. Ensuite, les différentes utilisations d'iPOJO seront présentées, en mentionnant les avantages et les inconvénients dans chacun des cas.

1. Evaluation de la machine d'injection et d'interception

L'implémentation d'iPOJO est basée sur une machine d'injection et d'interception. Afin de quantifier le coût de cette infrastructure, il nous a paru intéressant de comparer les capacités d'injection d'iPOJO par rapport à d'autres technologies d'injection, et plus particulièrement le coût de l'injection d'iPOJO par rapport à d'autres technologies. Cette section décrit un banc d'essai mis au point pour l'évaluation et la comparaison des surcoûts engendrés par l'injection.

1.1. Description du banc d'essai

Ce banc d'essai est en fait une extension d'un banc d'essai existant (Facbench) qui s'intéressait au même problème. Les extensions faites concernent principalement les types de données échangées afin d'effectuer un banc d'essai testant de nombreux mécanismes internes.

L'idée globale du banc d'essai est simple. Deux composants (un serveur et un client) sont interconnectés (Figure 118). Le client a accès au serveur en utilisant la technologie testée. Lorsque le banc d'essai démarre, le client interagit un grand nombre de fois avec le serveur. Ces interactions utilisent différents types d'arguments et de retour, mais le serveur n'effectue aucune action réelle.

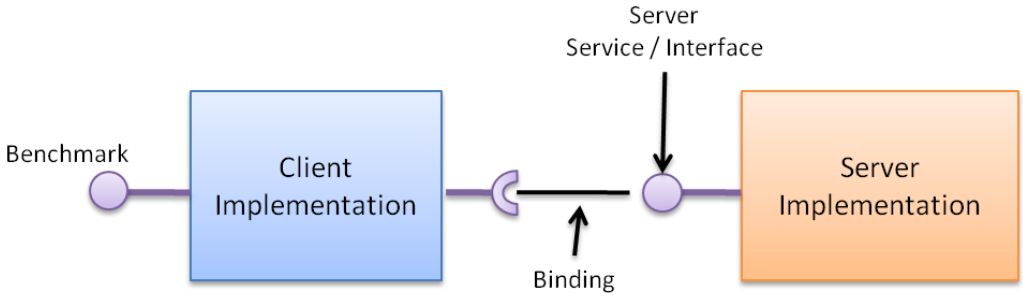


Figure 118. Acteurs du banc d'essai

1.2. Protocole de test

Le déroulement du banc d'essai est identique pour chaque technologie testée (Figure 119). Tout d'abord, les composants serveur et client sont créés. Ensuite, le serveur est mis à dispositif pour le client. Ceci s'effectue soit par injection dans le code du client (méthode *set,* passage dans le constructeur), soit par découverte lorsque celle-ci est supportée.

Une fois les composants initialisés, le banc d'essai effectue un « échauffement » afin d'éviter tous ralentissements dus à l'initialisation de la machine virtuelle (chargement de classe…). Lorsque ce préchauffage est effectué, le test est réellement effectué cinq fois. Une mesure est prise à chaque exécution.

Un test s'effectue en demandant au client d'appeler une méthode sur le serveur un grand nombre de fois (1 million). Quatorze méthodes sont appelées ainsi en variant les paramètres échangés (sans arguments, objet de petite taille, tableau, tableau de grande taille, collection de grande taille…). Les types de retour varient également.

La prise de mesure s'effectue avant et après la demande à client d'effectuer les appels au serveur (méthode doBench). Seront retenus pour l'analyse les temps minimaux sur l'ensemble des cinq tests. Plus ce temps est petit, plus la technologie d'injection est performante.

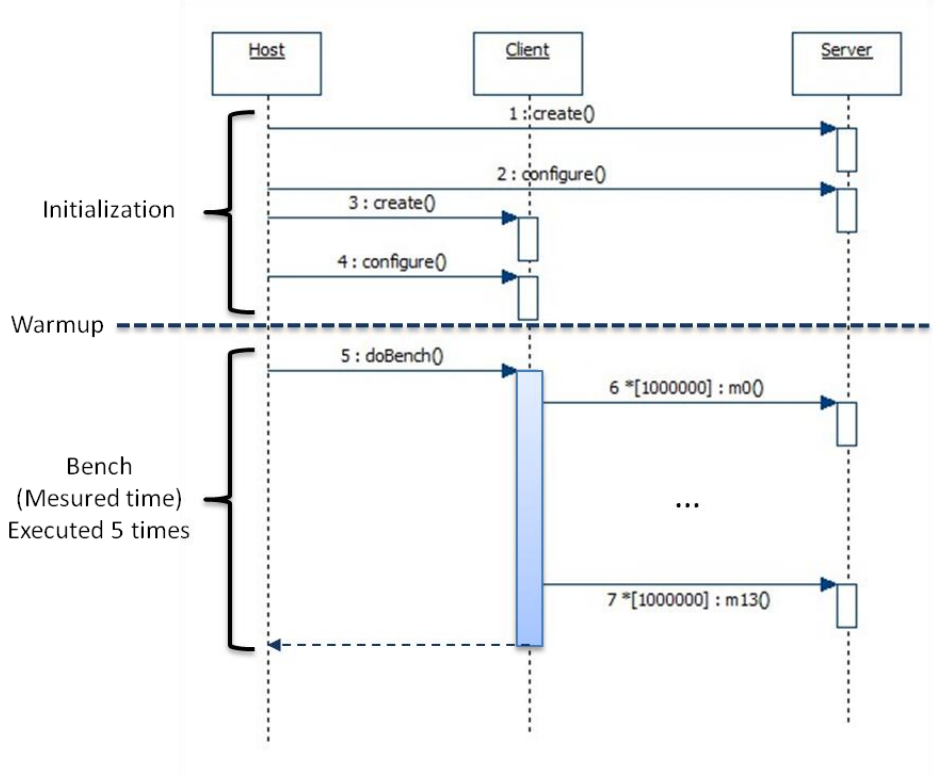


Figure 119. Fonctionnement du banc d'essai

Le banc d'essai est exécuté sur un ordinateur disposant d'un processeur 2,6 GHz Dual Core de marque Intel, fonctionnant sur Mac OS 10.5.4 (Leopard). La machine dispose également de 4 Go de mémoire à 667 MHz. Le banc d'essai est exécuté en utilisant Java 6.

1.3. Ordre de grandeur

Afin de comparer et d'évaluer le coût des technologies d'injection et d'interception, le banc d'essai a été également effectué sans infrastructure d'injection et d'interception sous-jacente, et avec des technologies de base tels que les proxies… Les mesures prises lors de ces tests nous permettent de positionner les différentes technologies d'injection et d'interception par rapport à des cas connus.

Tout d'abord, le banc d'essai est effectué en utilisant simplement Java avec des appels directs. Le client communique directement avec le serveur. Ensuite, le test est effectué avec des appels utilisant un protocole de synchronisation. En effet, la synchronisation (prise de verrou) est relativement couteuse en Java. Dans ce test, chaque méthode du serveur est synchronisée. Trois configurations testent différentes sortes de proxies. Dans ces cas le client n'accède pas directement au serveur, mais les appels sont relayés par un objet intermédiaire. Trois types de proxies ont été utilisés :

- Un proxy statique développé manuellement déléguant simplement les méthodes sur le serveur
- Un proxy statique développé manuellement, mais utilisant la réflexion afin d'invoquer les méthodes sur le serveur
- Un proxy dynamique généré par la machine virtuelle Java, dont l'intercepteur (InvocationHandler) délègue les appels sur le serveur.

Enfin a également été mesuré le coût des appels utilisant JMX (le serveur est exposé sous la forme d'un *MBean* statique que le client invoque en utilisant JMX et un connecteur RMI). En effet, JMX est une technologie régulièrement utilisée pour la reconfiguration dynamique, car elle propose une infrastructure distribuée, et sécurisée permettant la supervision d'applications Java.

Le tableau ci-dessous donne les mesures effectuées dans tous les cas cités auparavant. Nous remarquerons plus particulièrement le coût de la synchronisation par rapport à des appels directs et le coût des proxies dynamiques.

Type d'invocation	Temps mesurés
Invocation directe	
Invocation de méthodes synchronisées	
Serveur derrière un proxy statique	
Serveur derrière un proxy statique avec appels réflectifs	
Serveur derrière un proxy dynamique	
Serveur atteint avec JMX	

Tableau 31. Ordre de grandeur pour le banc d'essai

1.4. Plates-formes et technologies testées

Afin de comparer iPOJO par rapport aux autres technologies d'injection et d'interception, plusieurs technologies ont été testées. Une implantation utilisant OSGi™ a été tout d'abord effectuée, suivit d'une implantation utilisant l'implémentation de Declarative Services disponible sur Apache Felix. Ensuite Spring Dynamic Modules a été évalué de deux manières différentes. Apache Tuscany, une implémentation libre de SCA a également été étudiée. Cette section décrit les configurations testées pour chacune de ces technologies ainsi que pour iPOJO.

1.4.1. iPOJO

Quatre implantations différentes du protocole de tests utilisant iPOJO seront décrites et analysées dans ce chapitre. Toutes testent l'injection d'objet de service et l'accès au service. La première implantation utilise l'injection dans des membres de la classe d'implémentation du client afin d'accéder au serveur (Figure 120).

```
public class ClientImpl extends org.apache.felix.ipojo.bench.commons.DefaultClient {

  private Server server; // Service injection

  private int numberOfLoops;

  public void init(int numberOfLoops, Server bench) {
    this.numberOfLoops = numberOfLoops;
    // The second argument is ignored as
    // the server is discovered at runtime
  }

  protected void sbench1(int nb) {
    for(int i=0; i<nb; i++) {
      server.m1();
    }
  }

  ...
}
```

Figure 120. Accès au serveur grâce à un membre injecté

La deuxième implantation optimise la première en diminuant le nombre d'accès au Thread Local (garantissant l'état du composant). En effet, et comme l'illustreront les résultats, la création et l'accès au Thread Local sont coûteux (Figure 121). Ce style de développement reste néanmoins protégé face au dynamisme. En effet, le serveur utilisé sera mis en cache jusqu'à la sortie de la méthode.

```
public class ClientImpl extends org.apache.felix.ipojo.bench.commons.DefaultClient {

  private Server server; // Service injection

  private int numberOfLoops;

  public void init(int numberOfLoops, Server bench) {
    this.numberOfLoops = numberOfLoops;
    // The second argument is ignored as
    // the server is discovered at runtime
  }

   protected void sbench1(int nb) {
     Server s = server; // Use a local variable to avoid ThreadLocal accesses
     for(int i=0; i<nb; i++) {
       s.m1();
     }
   }

  ...
}
```

Figure 121. Utilisation d'une variable locale afin d'optimiser les temps d'injection

La troisième configuration effectue l'injection des objets de service (serveur) en utilisant des méthodes de liaisons. Il faut noter que ce style de modèle de développement demande nécessairement des blocs de synchronisation autour de l'accès au serveur. En effet, les méthodes de liaison injectant et retirant le serveur sont appelées dans un fil d'exécution différent de celui exécutant les tests (Figure 122).

```
public class ClientImpl extends DefaultClient {

  private Server server; // Service object

  private int numberOfLoops;

  public synchronized void bindServer(Server s) { // Bind method
    server = s;
  }

  public synchronized  void unbindServer(Server s) { // Unbind method
    server = null;
  }


  protected void sbench1(int numberOfLoops) {
    for (int i = 0; i < numberOfLoops; i++) {
      synchronized (this) { // Lock the server access that can
                            // be accessed from a different thread.
        server.m1();
      }
    }
  }

  ...

}
```

Figure 122. Utilisation de méthode de liaison pour accéder au serveur

Enfin, la dernière implantation utilise des compositions. Le client est exécuté dans une composition qui importe le serveur (grâce à une importation de service). Ce teste vise à montrer le temps d'accès à un service

importé au sein d'une composition par rapport à l'accès *direct* à ce service. Ce test utilise la première implémentation du client (injection dans des membres de la classe non optimisée).

1.4.2. OSGi et Declarative Services

Il nous a paru intéressant de quantifier le temps d'accès à un service en utilisant OSGi™ seulement. En effet, les services OSGi™ étant dynamiques, il est nécessaire de mettre en place un protocole de synchronisation complexe. Ce protocole a un coût non négligeable. Le code ci-dessous montre un aperçu du protocole de synchronisation nécessaire.

```
public class OSGiDynamicClient extends DefaultClient implements ServiceListener {
    private BundleContext context;
    public OSGiDynamicClient(BundleContext bc) {
      context = bc;
      context.addServiceListener(this);
    }

    public void doBench() {
      // Service lookup
      ServiceReference ref = context.getServiceReference(Server.class.getName());
      if (ref == null) {
        System.err.println("Cannot find the server");
        return;
      } else {
        synchronized (this) { // Lock the server access that can be accessed
                              // by the serviceChanged thread
          server = (Server) context.getService(ref);
        }
        super.doBench();
      }
    }
    protected void sbench1() {
      for (int i = 0; i < numberOfLoops; i++) {
        synchronized (this) { // Lock to ensure the server access.
          if (server != null) { // Check availability
            server.m1();
          }
        }
      }
    }

   ...

    public void dispose() {
      context.removeServiceListener(this);
      System.gc();
    }
    public void serviceChanged(ServiceEvent arg0) {
      synchronized (this) {
        // NOTHING TO DO, it just simulates the lock holding ...
      }
    }
  }
```

Figure 123. Implantation du client avec OSGi

Une deuxième implantation utilisant Declarative Service a été réalisée. Le modèle de développement utilisé est sensiblement équivalent à l'implantation utilisant iPOJO et des méthodes de liaison. Nous avons utilisé l'implémentation de Declarative Services disponible sur Apache Felix.

1.4.3. Spring Dynamic Modules

Le banc d'essai décrit ci-dessus a été implanté de deux manières différentes avec Spring dynamic modules (version 1.1.1). La première implantation s'exécute au sein d'un seul *bundle* (Figure 124). En effet, Spring Dynamic Modules associe le contexte d'application (*Application Context*) est associé à un *bundle* OSGi™. Ainsi une application Spring sera conditionnée à l'intérieur d'un seul *bundle*. L'exécution intra-*bundle* ne relève pas des mêmes mécanismes qu'une implantation utilisant plusieurs *bundles* (qui aura alors différents contextes d'application).

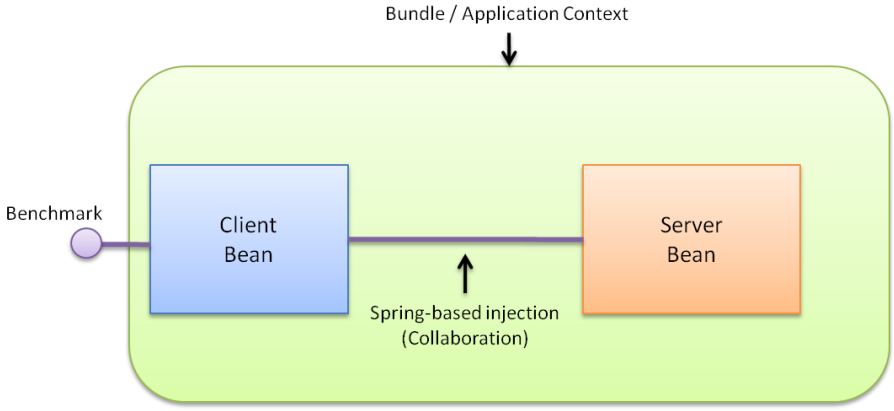


Figure 124. Banc d'essai utilisant l'injection intra-*bundle* de Spring-DM

La deuxième implantation utilise justement deux *bundles* (Figure 125). Un *bundle* contient le serveur et exporte le service associé. Le second *bundle* contient le client et importe le service donnant accès au serveur.

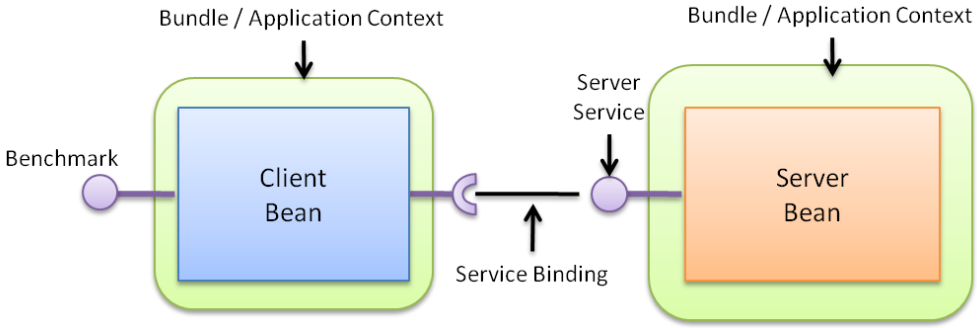


Figure 125. Banc d'essai utilisant l'injection inter-*bundle* de Spring-DM

1.4.4. Apache Tuscany

Apache Tuscany est une implémentation libre de SCA. Bien qu'il ne supporte pas le dynamisme et n'est pas lié à une technologie d'implémentation, SCA est devenu une technologie incontournable. Nous avons choisi d'implanter les tests en Java. Deux tests ont été réalisés.

Le premier test s'exécute au sein d'un seul composite (Figure 126). Ceci permet à la plate-forme SCA d'injecter directement le serveur dans le client (grâce à une méthode de liaison).

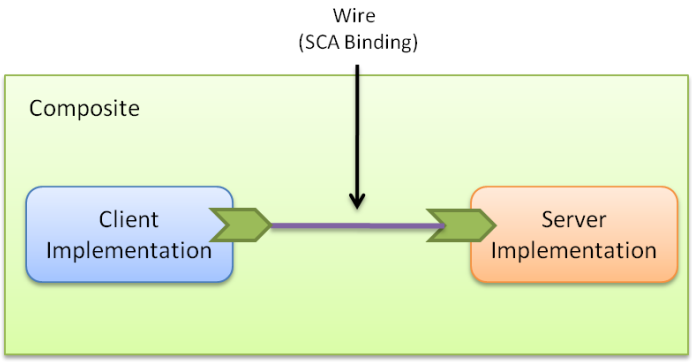


Figure 126. Banc d'essai dans un seul composite SCA

Le second test utilise deux composites distincts pour l'exécution du client et du serveur (Figure 127). Cependant, afin de donner accès au serveur, un troisième composite décrit les liens entre les deux composites sous-jacents. L'accès au serveur est alors décrit dans la description du composite global.

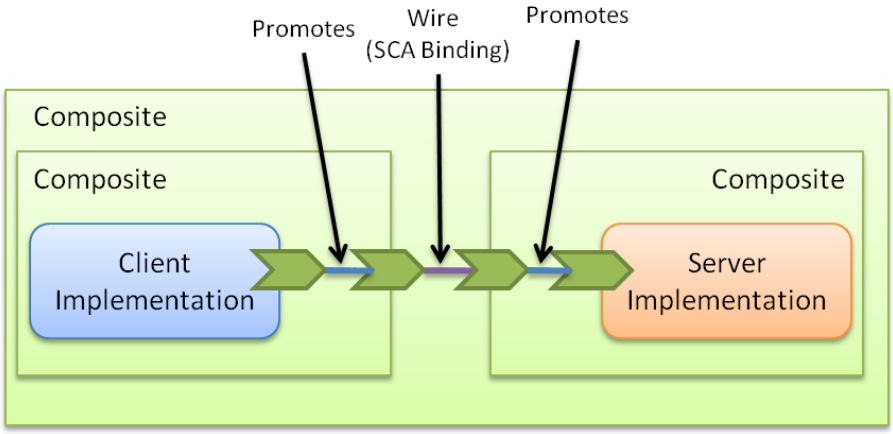


Figure 127. Banc d'essai utilisant plusieurs composites SCA

1.5. Résultats et analyses

Les résultats de ce banc d'essai sont visibles dans le tableau et le graphique ci-dessous. En vert sont représentées les valeurs des tests d'étalonnage, en bleu les résultats des différentes exécutions utilisant iPOJO et en rouge les autres technologies testées.

Technology	Metric
Direct Invocation	132
iPOJO Optimized Service field injection	152
Static Proxy	235
Synchronized Invocation	633
OSGi Dynamic Service access	659
iPOJO Service Method Injection	670
Declarative Services	690
Spring Intra-bundle injection	831
iPOJO Service field injection	1389
iPOJO Imported service access	1397
Static Proxy with dynamic invocations	1723
Dynamic Proxy	1889
Invocation throught JMX	9380
Spring inter-bundle injection	12050
Tuscany intra-composite injection	12643
Tuscany inter-composite injection	12806

Tableau 32. Résultat du banc d'essai

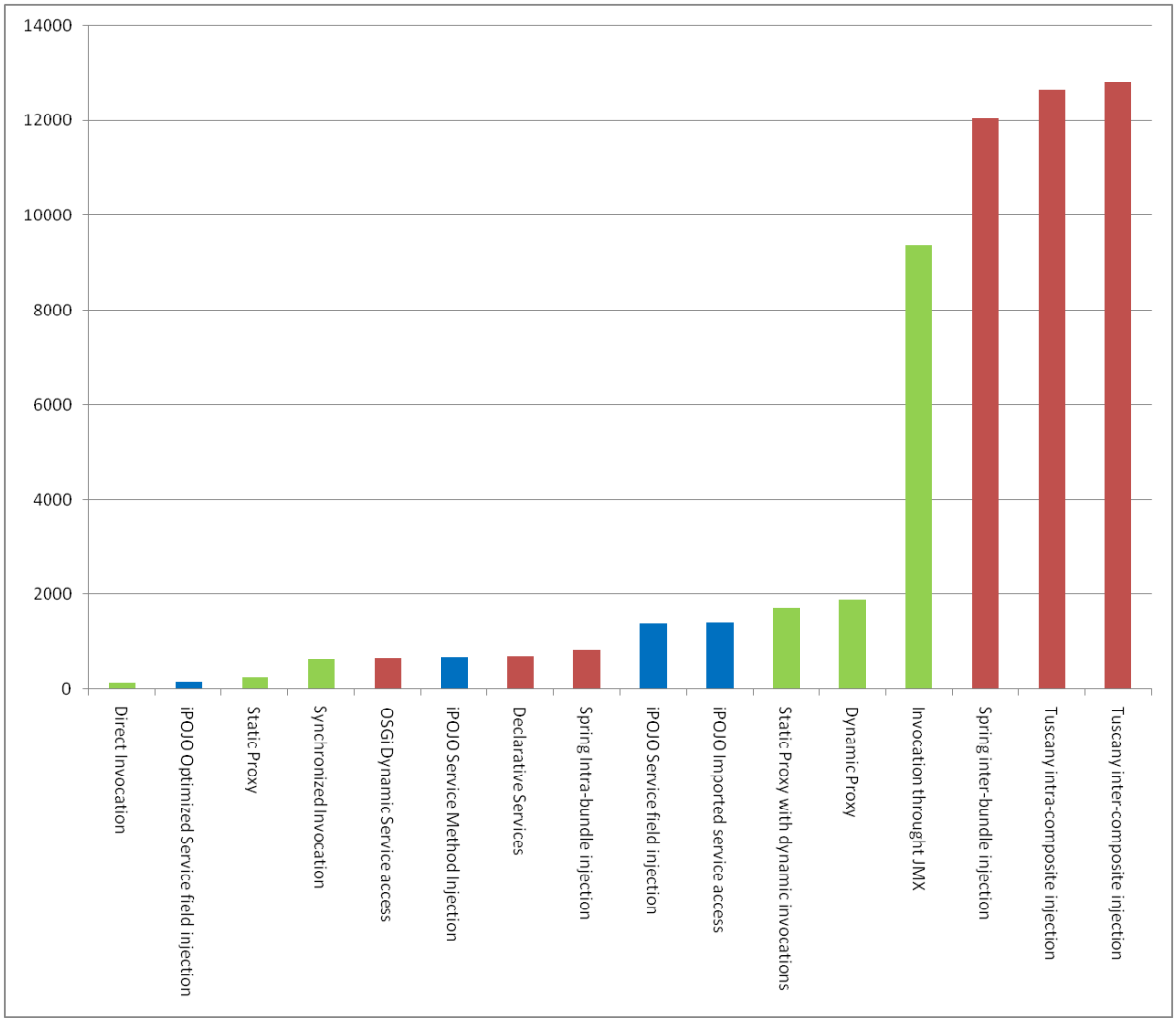


Figure 128. Comparaison des différentes technologies sur le banc d'essai

De ces résultats nous pouvons remarquer plusieurs détails intéressants. Tout d'abord, la première implémentation « naïve » utilisant iPOJO (injection de l'objet de service dans un membre de la classe) obtient de meilleures performances qu'un proxy dynamique. Ce résultat prouve l'intérêt d'une manipulation de bytecode par rapport aux mécanismes de réflexion. L'injection optimisée des objets de service obtient d'excellentes performances, car évite un grand nombre d'indirections. Ceci est possible, car l'objet utilisé reste capturé le temps de l'exécution de la méthode et des méthodes appelées (grâce au confinement par fil d'exécution). Les mécanismes de cache sous-jacents évitant les blocs de synchronisation sont plus rapides que les blocs synchronisés. Nous pouvons également remarquer que l'injection par méthode de liaison a des performances équivalentes par rapport à OSGi™ « pur » et à Declarative Services. Enfin, le temps d'accès d'un service importé à l'intérieur d'un composite est pratiquement équivalent à l'accès à ce même service en dehors d'un composite. Ce fait est dû à la republication du même objet de service dans le composite et non pas à un proxy.

Finalement, nous remarquerons que l'utilisation de technologies lourdes pénalise grandement les infrastructures telles que Tuscany ou Spring. Concernant ce dernier, Il faut noter que l'injection intra-*bundle* (c'est-à-dire à l'intérieur du même contexte d'application) n'est pas dynamique et obtient donc de bonnes performances.

2. Temps de démarrage de la plate-forme

La plate-forme OSGi™ a un problème connu lors du démarrage d'applications à service de grande taille. En effet, lors du démarrage, l'application enregistre et requiert de nombreux services. Ces interactions impliquent que la plate-forme propage un grand nombre d'évènements (appelée tempête d'évènements). Cependant, cette propagation est effectuée séquentiellement, par un seul flux d'exécution. De plus, la généricité d'OSGi™ force les différentes implémentations à réévaluer l'ensemble des filtres des consommateurs de services afin de déléguer les évènements (Figure 129 partie A). Ce parcours et cette évaluation sont très coûteux.

Afin de contourner ce problème, la plupart des applications de grande taille, comme l'IDE Eclipse, n'utilisent pas l'approche à service, mais utilisent les « extensions ». Ce modèle a l'avantage d'éviter la tempête d'évènements.

Cependant, il est également possible de contourner la tempête d'évènement avec iPOJO. Un des principaux problèmes lors du démarrage d'une application à service de grande taille est le parcours et l'évaluation systématique des consommateurs de service afin de déléguer les évènements seulement aux consommateurs intéressés par l'évènement. Afin d'éviter ce parcours, iPOJO enregistre un seul consommateur « fictif » sans filtrage et délègue les évènements aux différentes instances (Figure 129 partie B). Comme, ce propagateur interne d'évènements possède plus de connaissances sur les dépendances de service à traiter, la propagation est effectuée beaucoup plus rapidement.

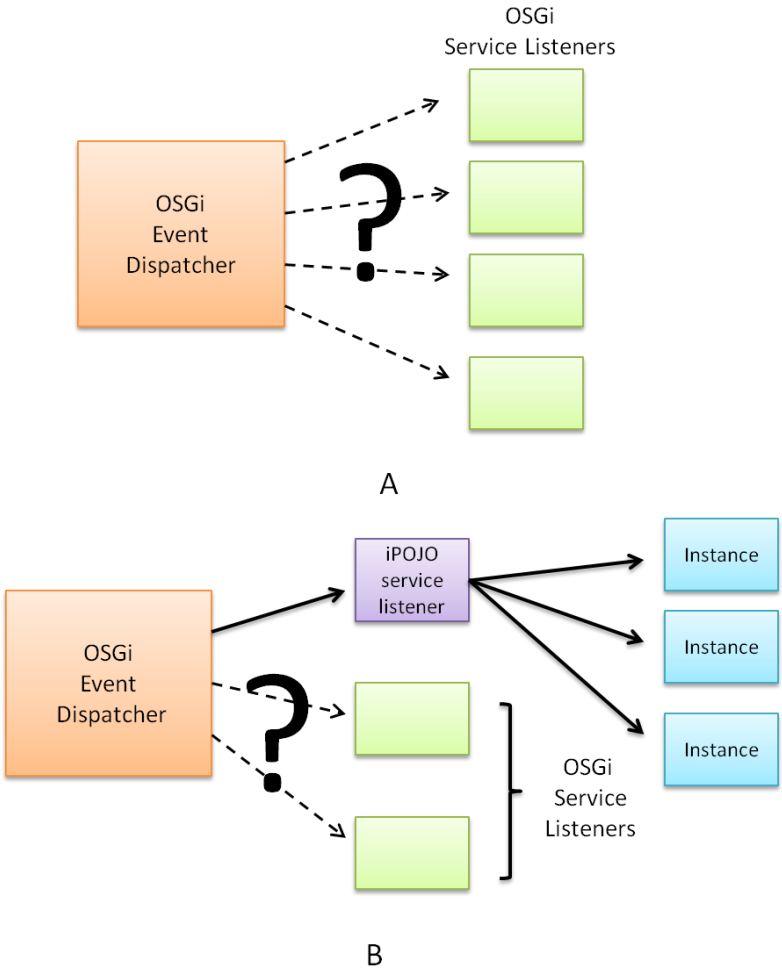


Figure 129. Fonctionnement d'iPOJO pour faire face à la "tempête d'évènements "

Cette section évalue ce gain et analyse le temps de démarrage d'une application de grande taille utilisant différentes technologies tel qu'OSGi™ pur, Declarative Services, Spring-DM et iPOJO.

2.1. Description du banc d'essai

Le banc d'essai est basé sur une application fictive de grande taille. Cette application est composée de 10 000 *bundles*, et utilisant 10 000 services différents. Chaque *bundle* requiert entre 0 et 5 services. Une des particularités de cette application est son organisation pyramidale. Ainsi, un *bundle* du niveau n, ne peut utiliser que des services du niveau n-1. Évidemment, les *bundles* du niveau 0 ne consomment pas de service, mais se contentent de fournir des services. Ainsi, au final tous les composants de l'application sont valides (Figure 130).

L'application est ensuite exécutée sur la plate-forme OSGi™ Apache Felix. Cette plate-forme est configurée pour lancer tous les *bundles* composant l'application lors de son démarrage. Le temps de démarrage est mesuré afin de comparer les différentes technologies. La fin du lancement est détectée lorsque l'ensemble des composants de l'application est démarré.

Le banc d'essai est exécuté sur un ordinateur disposant d'un processeur 2,6 GHz Dual Core de marque Intel, fonctionnant sur Mac OS 10.5.4 (Leopard). La machine dispose également de 4 Go de mémoire à 667 MHz. Le banc d'essai est exécuté en utilisant Java 6.

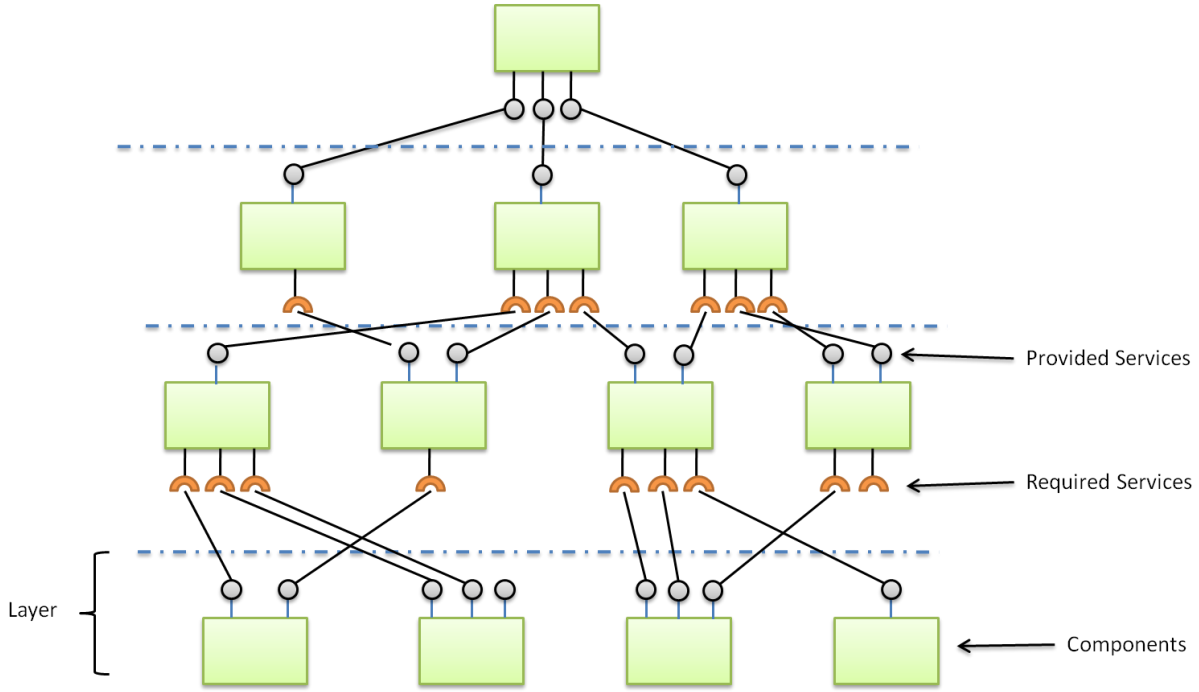


Figure 130. Structure de l'application testée

La plate-forme lancée est configurée avec un cache contenant l'application. Ceci permet d'éviter que la plate-forme copie les *bundles* dans le cache, ce qui est très couteux en temps d'accès disque.

2.2. Plates-formes et technologies testées

Afin de valider l'intérêt de l'approche choisie dans iPOJO, l'application a été implémentée à la fois avec OSGi™ « pur » et avec iPOJO.

La première variante de l'application est réalisée en utilisant l'API d'OSGi™. Ainsi, chaque *bundle* est démarré avec un activateur, enregistrant les différents écouteurs pour service requis et publiant les services fournis dès lorsque tous les services requis sont disponibles.

La deuxième variante utilise iPOJO. Chaque *bundle* contient un type de composant et déclare une instance de ce type. Chaque type suit les préceptes de l'application, et expose son architecture. Celle-ci nous permettra justement de vérifier la structure de l'application. Ce test a été exécuté avec deux versions d'iPOJO. L'une n'utilisant pas le propagateur interne et l'autre l'utilisant.

2.3. Résultats et analyses

L'application a été lancée cinq fois avec chacune des technologies. Le temps minimal a été retenu (Tableau 33). Ces résultats montrent l'intérêt d'utiliser un propagateur d'évènement interne. Cependant, le gain n'est pas exceptionnel et le temps de démarrage reste grand. Néanmoins, la comparaison entre les deux versions d'iPOJO montre que la tempête d'évènement est mieux gérée grâce au propagateur interne. La différence minime avec OSGi™ provient du surcoût entrainé par la lecture des métadonnées et à la création d'objets.

Technologies	Temps de démarrage
OSGi™	512 687 ms
iPOJO sans propagateur interne	794 261 ms
iPOJO avec propagateur interne	491 543 ms

Tableau 33. Résultat du banc d'essai concernant le temps de démarrage

3. Utilisation d'iPOJO sur une passerelle résidentielle

iPOJO est aujourd'hui utilisé dans plusieurs domaines d'applications. Cependant à son origine, iPOJO s'est principalement focalisé sur les plates-formes résidentielles. Plusieurs projets de passerelle ont été développés avec iPOJO comme support d'exécution. Cette section décrit la problématique de ce contexte et montrera comment iPOJO résout une partie de cette problématique.

3.1. Contexte et problématique des passerelles résidentielles

De nos jours, nous vivons dans un environnement contenant de nombreux dispositifs électroniques communicants tels que les téléphones mobiles, les PDAs, les récepteurs satellites… De plus, ces dispositifs deviennent de plus en plus petits et perfectionnés. L'émergence de ces dispositifs a profondément changé la façon d'interagir avec notre environnement. Tout laisse à penser que cette expansion va continuer. Les dispositifs auront des capacités de plus en plus évoluées et coopéreront automatiquement afin de fournir des services de plus haut niveau.

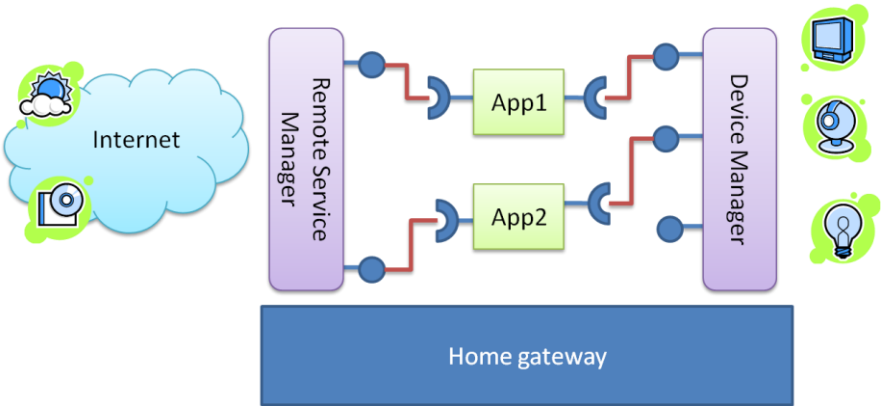


Figure 131. Positionnement d'une passerelle résidentielle

Cependant, avant de réaliser cette vision, de nombreux défis scientifiques et techniques restent à relever. Le développement de services évolués utilisant des dispositifs hétérogènes, distribués est particulièrement complexe. Le dynamisme est un besoin crucial dans ce type d'application. En effet, l'environnement et le contexte de l'utilisateur évoluent en permanence. De plus, la passerelle résidentielle

doit permettre la communication avec des services distants (services web) et les dispositifs présents actuellement dans la maison (Figure 131).

3.2. Vers un serveur d'applications pour les applications résidentielles

Les travaux présentés dans cette section ont été initiés dans le contexte du projet européen ITEA ANSO. Ce projet rassemblait les acteurs européens majeurs du contexte afin de définir et de construire une passerelle résidentielle au dessus une plate-forme de service. Ces travaux ciblent particulièrement les besoins technologiques afin de faciliter le développement et l'exécution d'applications résidentielles. Parmi ces besoins, nous retrouvons la distribution, le dynamisme, la persistance, la sécurité…

La solution proposée vise à fournir un serveur d'applications dédié à l'exécution d'applications résidentielles. Ce serveur fournit à la fois une machine d'exécution supportant le dynamisme (iPOJO) et un ensemble de services techniques facilitant le développement d'applications. Ainsi, le serveur gère les communications distantes avec les services web et les dispositifs de la maison, offre des fonctionnalités d'administration, propose un support persistant permettant de stocker l'état des applications, et permet également l'ordonnancement d'actions ainsi que l'envoi et la réception d'évènements (Figure 132).

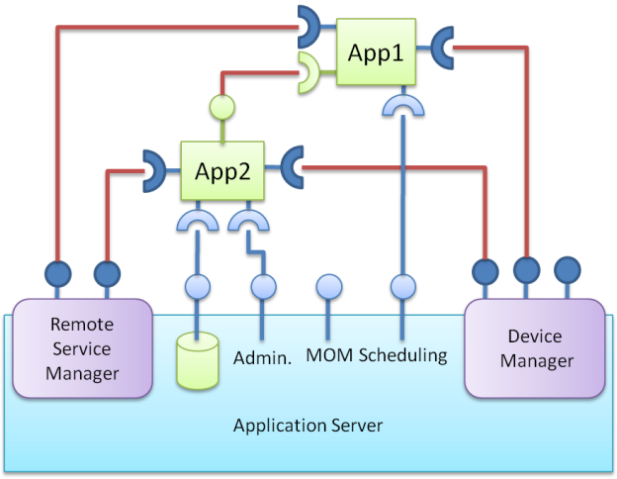


Figure 132. Schéma de principe du serveur d'applications

De plus, un certain nombre de handlers ont été développés afin de masquer les interactions avec les différents services techniques (Figure 133). Ainsi, le développeur peut mieux se focaliser sur le code logique de son application. Ensuite, il configure les différentes propriétés non fonctionnelles. Lors de l'exécution, le serveur prend en charge l'ensemble de ces besoins non fonctionnels ainsi que le dynamisme.

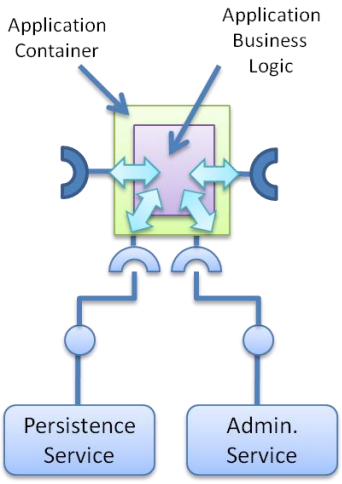


Figure 133. Mécanisme d'extension dans le serveur d'applications

Ce serveur est encore développé aujourd'hui au sein de l'équipe ADELE [199, 200]. De nombreux problèmes du domaine restent encore à travailler. Ainsi, la sécurité ou l'autonomie sont des points cruciaux devant être traités. Cependant, bien qu'il ait beaucoup évolué depuis ces débuts dans le contexte du projet ANSO, de nombreuses briques sont utilisées chez des partenaires industriels. Par exemple, France Telecom utilise un portail web supportant l'apparition et la disparition de « portlet » (morceau d'IHM). Schneider Electric utilise également les mécanismes d'exécution mis en place par iPOJO afin de développer une plate-forme résidentielle innovante. Nous ne pouvons détailler ces utilisations pour des raisons de confidentialité.

4. Utilisation d'iPOJO dans le serveur d'application JEE JOnAS

Le serveur d'applications OW2 JOnAS est un serveur d'applications implémentant les spécifications Java EE (Enterprise Edition). Ce serveur permet d'exécuter des applications d'entreprise composées de Servlet, Enterprise Java Beans... JOnAS a été l'un des premiers projets à utiliser iPOJO. Ce besoin provient que les dernières versions de JOnAS utilisent OSGi™ comme plate-forme d'exécution. Cette section décrit les points importants ayant poussé ce projet à utiliser iPOJO et quelles en sont les perspectives.

4.1. Un serveur d'applications JEE au dessus d'OSGi™

L'objectif de JOnAS est d'implémenter entièrement la spécification JEE, en s'appuyant sur des composants tiers tout en permettant une flexibilité d'utilisation maximale (du développeur à l'administrateur, en passant par le clustering).

JOnAS s'est récemment tourné vers OSGi™ afin d'avoir une grande flexibilité au niveau de la composition du serveur d'applications (Figure 134)[40]. Ce choix a été motivé par la popularité grandissante d'OSGi™, mais également par un besoin de découplage et de dynamisme promu par l'approche à service. De plus, OSGi™ propose également des facilités de déploiement très intéressantes dans le contexte de JOnAS. Ainsi l'intégralité des services techniques est aujourd'hui implémentée en utilisant iPOJO. Ceux-ci sont ensuite fournis en temps que service OSGi™ et peuvent également utiliser d'autres services techniques en recherchant le service associé. De cette manière, ces services peuvent apparaître et disparaître dynamiquement. De plus, il est possible de proposer plusieurs implémentations d'un même service technique, et ainsi de substituer une implémentation par une autre. Enfin, les composants applicatifs s'exécutant sur JOnAS sont également exposés sous la forme de service OSGi™.

Figure 134. Architecture de Jonas 5

4.2. Gestion du dynamisme

OSGi™ a permis au serveur JOnAS d'être très flexible. Cependant, il a rapidement fallu gérer le dynamisme des services. En effet, les services techniques fournis par le serveur peuvent être installés et désinstallés dynamiquement.

Une gestion manuelle de ce dynamisme n'était pas envisageable vus la quantité de code à migrer et le coût en terme de ligne de code dû au dynamisme. iPOJO a donc été choisi pour le modèle de développement simple qu'il propose. De plus, le fait qu'iPOJO ne soit pas lié à une spécification lui permet de réagir plus rapidement aux attentes des utilisateurs. Ainsi, les services techniques ont été implémentés en utilisant iPOJO afin de rendre la gestion du dynamisme plus aisée.

La mise en place du serveur a été « plutôt rapide[43] ». Malgré quelques difficultés initiales lors de la migration vers OSGi™, et quelques mises au point sur iPOJO, le serveur JOnAS 5 fonctionne aujourd'hui sur OSGi™/iPOJO.

4.3. Perspectives et attentes

L'équipe JOnAS continue à contribuer à iPOJO. Cette collaboration a apporté beaucoup de retour et a permit d'améliorer grandement l'implémentation. Les points importants à aborder aujourd'hui pour JOnAS concernent le déploiement. En effet, iPOJO ne propose pas de mécanisme de déploiement, point crucial aujourd'hui. La mise en place d'un tel outil serait très bénéfique. Il permettrait de prendre en charge le déploiement de *bundles* contenant des composants iPOJO de manière automatique. Ainsi, les dépendances de services pourraient être résolues non pas au niveau du registre OSGi™ seulement, mais également grâce à un dépôt de *bundles*. Les liens entre iPOJO sont *OSGi Bundle Repository* (OBR) sont actuellement investigués.

5. Utilisation d'iPOJO sur les téléphones mobiles

iPOJO est également utilisé sur les téléphones portables. En effet, il devient clairement nécessaire de proposer un intergiciel évolué afin de faciliter le développement d'applications sur ces plates-formes. Cette section relate l'expérience d'uGASP qui utilise iPOJO afin de créer une infrastructure pour les jeux massivement-multi-joueurs ubiquitaire.

5.1. Contexte

Le domaine du jeu multi-joueurs ubiquitaire (JMU) pourrait constituer l'avenir du jeu en mobilité. Ce type de jeu offre une nouvelle expérience de jeu permettant aux joueurs d'interagir à deux niveaux simultanément, dans le monde réel (ou physique) et dans un monde virtuel (ou imaginaire). Ainsi les actions d'un joueur dans le réel telles que son déplacement, ses interactions avec des dispositifs technologiques disséminés sur une zone de jeu, ses informations bio-sensorielles ou encore ses mouvements corporels ont un impact direct sur l'état du monde virtuel et donc sur le déroulement du jeu. A l'inverse, l'état du monde virtuel peut aussi impacter les possibilités d'interactions dans le monde réel que ce soit avec les dispositifs disséminés ou avec les autres joueurs.

La mise en place de JMUs implique la gestion d'une grande variété technologique telles que des dispositifs embarqués (téléphones mobiles, PDA, capteurs biomédicaux, récepteur GPS...) portés par les joueurs, des dispositifs disséminés sur la zone de jeu (tags RFID, capteurs environnementaux...), des connectivités réseaux multiples (réseau cellulaire, Wifi, Bluetooth, Wimax...)...

[43] Propos recueilli auprès de l'équipe Jonas.

uGASP (ubiquitous GAming Services Platform) est un intergiciel dédié aux JMUs, il a pour objectif d'offrir une couche d'abstraction de la complexité du développement induite par ce type de jeu.

5.2. Pourquoi iPOJO

uGASP est une évolution majeure du middleware GASP dont le développement à débuter en 2004. Ce dernier offre le support de jeux multi-joueurs sur appareils mobiles et plus particulièrement sur téléphones Java. GASP met en œuvre une architecture monobloc quasi-statique implémentée en Java au dessus d'Apache Tomcat. En effet, l'ajout de nouvelles fonctionnalités dans GASP nécessite une recompilation intégrale du code liée au choix d'une approche orientée objet. L'unique partie dynamique est le déploiement de nouvelles logiques de jeu.

L'expérience issue du développement de GASP a conduit à adopter une architecture dynamique, modulaire, extensible et configurable pour l'intergiciel dédié aux JMUs. C'est pourquoi uGASP est bâti sur la base d'une plate-forme à service.

Le choix s'est alors porté sur OSGi™ dont les caractéristiques intrinsèques sont en adéquation avec les choix architecturaux. uGASP a donc été développé et testé sur l'implémentation OSGi R4 Apache Felix.

Cependant les mécanismes OSGi relatifs à la gestion dynamique de services (publications, souscriptions, résolutions des dépendances de services...) sont complexes. Cette gestion nécessite une réelle expertise d'OSGi™ et des mécanismes de synchronisation de Java.

L'équipe du projet uGasp tenait à ce que l'adoption de cette nouvelle architecture ait un impact minimal sur l'implémentation originale de GASP afin de pouvoir clairement dissocier la logique fonctionnelle de uGASP des mécanismes OSGi sous-jacents. De plus, l'objectif d'uGASP étant de fournir une couche d'abstraction de complexité aux développeurs de JMUs, on ne pouvait imposer à un utilisateur d'uGASP de manipuler des mécanismes OSGi bas niveau afin de développer le code serveur de leur jeu. C'est pourquoi, l'équipe du projet a choisi d'adopter une couche d'abstraction des mécanismes de gestion de services OSGi sur laquelle reposerait uGASP. Il existe plusieurs solutions de ce type telles que les couches Service Binder, Declarative Services, Dependency Manager, et iPOJO...

iPOJO a été choisi pour plusieurs raisons. Tout d'abord, iPOJO facilite le développement de composants de services OSGi grâce à la clarté et la simplicité de la description XML des caractéristiques des types de composant, et ce même pour un novice de la programmation orientée composants. La compilation de *bundles* OSGi est également très simple à prendre en main car basée sur l'outil BND et intégrable à l'outil *Apache Ant* afin d'automatiser le processus de compilation et la création de *bundles*.

Les mécanismes d'injections de services dans les membres de la classe mère du composant sont très intéressants. Tout d'abord, lorsque l'on veut implémenter une version composant d'un code orienté objet celui-ci n'est que très faiblement impacté et le passage d'un modèle à l'autre est très rapide. Ainsi, l'implémentation orientée composant (utilisant iPOJO) d'uGASP à partir du code orienté objet de GASP a été intégralement réalisée en moins de 3 semaines.

D'autre part, cette approche permet une compréhension haut niveau des caractéristiques du modèle dynamique à composants, notamment en ce qui concerne le cycle de vie du composant, de la résolution de liens entre composants... remplissant ainsi parfaitement son rôle d'abstraction de complexité. Cette caractéristique est primordiale dans le cadre de l'élaboration de l'intergiciel puisque permet une prise en main rapide d'uGASP par un développeur non initié aux modèles à composants et à OSGi™.

iPOJO offre également un mécanisme d'usine à composants (Fabrique) largement mis en œuvre dans uGASP notamment dans la gestion de multiples instances de jeu d'un même *bundle* de logique serveur de jeu.

Enfin, iPOJO permet la mise en place de composites, ceux-ci se révélant indispensables dans le cadre de services spécifiques mutualisables entre instances de composants. Par exemple, le service de géo-localisation d'uGASP est accessible par de nombreuses instances à un même instant, celui-ci requiert une initialisation de paramètres de projection spécifiques à chaque transaction (plusieurs requêtes de projection relatives à un même centre de projection), la seule manière d'assurer l'isolation du service pour toute transaction est de créer un composite liant instance de jeu et instances de services requis. Cette caractéristique d'isolation permet de se décharger de mécanismes complexes de synchronisation entre les instances de services.

5.3. Architecture choisie

uGASP est donc construit au dessus d'OSGi R4 et d'iPOJO (voir figure ci dessous). L'adoption d'un modèle à composant a permis de mettre en œuvre dans uGASP une architecture totalement modulaire et divisée en famille de services: services de connectivités réseaux (HTTP, Socket), de connectivités protocolaires (MooDS, OSC...), de connectivités utilisateurs (administrateurs, joueurs, maitre de jeu...), de services externes (dédiés aux utilisateurs comme le lobby pour les joueurs), de services internes (systèmes, ubiquitaires dédiés aux logiques de jeu tels que le service de géo-localisation...), de gestion d'instances de jeux et enfin de logiques de jeux.

Les liens entre composants sont donc résolus par iPOJO. Par exemple, le fait que le service de gestion de la base de données ne soit pas accessible met en état invalide le composant de gestion de session de jeu et par effet de cascade impacte l'accessibilité de la plateforme au niveau du connecteur HTTP. Les logiques de jeu sont implémentées comme des types de composants, ce qui permet de gérer de multiples instances d'un même jeu, grâce aux principes de fabrique.

Les composites iPOJO sont utilisés pour isoler une instance de jeu et les instances de services ubiquitaires qu'elle requiert.

5.4. Exemple d'application

Un JMU reposant sur uGASP/iPojo/Apache Felix a été réalisé en juin 2008 en collaboration avec la société Xilabs pour le festival d'arts numériques « Le cube » à Issy Les Moulineaux, France. Ce jeu intitulé Meet Your Heart Between (MYHT) est un jeu mêlant géo-localisation par GPS et « affective computing ». Le jeu accueille 8 joueurs simultanément. Chaque joueur dispose d'un téléphone mobile Java (J2ME) et d'un capteur biométrique (ECG). Le but du jeu est de retrouver son *jumeau cardiaque* après avoir rejoint un point de départ ayant été attribué au joueur. Sur les téléphones sont affichés la carte *YahooMaps* de la zone de jeu, les projections des positions GPS correspondantes à la position réelle du joueur ainsi que le téléchargement des cartes associées sont effectuées par le service de géo-localisation de uGASP. Le joueur ne voit sur son écran que la position, et donc les déplacements, des joueurs dont les rythmes cardiaques sont similaires au sien.

5.5. Perspectives

Une des principales perspectives du projet uGASP est de supporter une architecture OSGi distribuée. Actuellement OSGi est centralisé, cependant il existe des tentatives de distribution telles que r-OSGi. iPOJO devrait dans un avenir proche offrir la possibilité d'adopter ce type d'architecture. Un travail en collaboration étroite avec l'équipe de développement d'iPOJO va être mené afin de permettre la connexion de clients OSGi embarqués à uGASP. Ceci permettrait de déployer à distance des *bundles* de logiques clientes de jeu aussi bien sur des téléphones mobiles supportant OSGi à l'instar d'Android que sur d'autres dispositifs disséminés sur la zone de jeu. Les problématiques d'administration à distance, de mise à jour, d'auto-configuration des clients ou encore d'adaptation à l'hôte pourraient trouver des réponses grâce à ce type d'architecture.

6. Conclusion

Ce chapitre a évalué l'implémentation actuelle d'iPOJO et a décrit trois utilisations de cette technologie dans différents domaines d'applications.

Les évaluations ont montré que l'implémentation d'iPOJO était utilisable dans de nombreux contextes, car n'implique pas un énorme coût. De plus, les performances d'iPOJO sont globalement meilleures que celle des autres modèles à composant à service tel que Spring-DM ou Tuscany (une implémentation de SCA). Ces performances de l'injection sont dues à l'utilisation d'une manipulation de bytecode. Cette manipulation évite la mise en place de mécanismes lourds tels que des proxies dynamiques. De plus, une attention particulière à été apportée au code afin de gérer correctement les problèmes de synchronisation et d'éviter les pertes de temps inutiles. De plus, un certain nombre d'optimisations telles que l'utilisation d'un propagateur d'évènements interne, ont été mises en place pour améliorer les performances.

iPOJO est aujourd'hui utilisé dans de nombreux contextes. Ce chapitre a choisi de décrire trois contextes d'utilisation différents. Ces contextes ont été sélectionnés, car ils sont très différents et demandent des fonctionnalités différentes. En effet, la plate-forme résidentielle requiert des services techniques et des handlers associés afin de faciliter le développement d'application ; le serveur d'applications JOnAS requiert un modèle de développement simple et une grande flexibilité dans la gestion du dynamisme ; enfin uGASP requiert un modèle de composition facile à utiliser. Cependant, dans tous ces domaines le dynamisme est apparu comme un besoin crucial.

De ces utilisations, nous pouvons remarquer qu'iPOJO est généralement utilisé à la base de serveur d'application dynamique et flexible. En effet, en proposant à la fois une machine d'exécution gérant le dynamisme et des fonctionnalités d'extension, iPOJO est un bon candidat dans ce contexte. Son faible poids en mémoire et ses performances lui permettent d'être utilisé dans de très nombreux domaines.

iPOJO est également utilisé dans d'autres projets tels qu'un portail web dynamique, utilisé chez France Telecom, une suite RFID ainsi que dans des environnements de développement…

Chapitre 11
Conclusion & Perspectives

Nous avons proposé iPOJO, un modèle à composant et une machine d'exécution permettant la conception, le développement et l'exécution d'application supportant le dynamisme. L'idée directrice d'iPOJO est d'infuser les concepts de l'approche à service dans un modèle à composant. Cette infusion s'effectue à la fois au niveau des composants atomiques, mais également dans les compositions. De plus, iPOJO est intégralement implémenté et est actuellement utilisé dans des contextes très différents tels que les plates-formes résidentielles, les serveurs d'applications et les applications pour téléphones mobiles. Cette implémentation est disponible en tant que sous-projet du projet Apache Felix. Dans ce chapitre, nous faisons une synthèse des propositions faites dans cette thèse.

Plusieurs questions restent ouvertes. Ce chapitre détaillera également quelques perspectives au travail proposé. Trois perspectives principales seront présentées. Celles-ci concerneront le déploiement d'applications dynamiques, le développement d'applications autonomiques et sensibles au contexte ainsi que la mise en place de langage de description d'architecture spécialisé et de machine d'exécution associée.

Le support du déploiement est devenu une caractéristique primordiale pour les applications. En effet, les solutions actuelles ne sont pas satisfaisantes. De plus, le déploiement d'applications dynamiques n'a jamais été investigué profondément.

L'informatique autonomique est devenue très populaire depuis le lancement de cette initiative par IBM en 2001. Cependant, les solutions proposées aujourd'hui afin de créer des applications autonomiques ont de nombreuses contraintes. Nous avons remarqué qu'un des principaux requis pour créer des applications autonomiques est le dynamisme. En effet, la structure de l'application supervisée doit pouvoir évoluer. Ces mêmes contraintes apparaissent dans les applications sensibles au contexte, ou la structure est régie par le contexte environnant.

Enfin, tout comme l'ont montré les différentes validations d'iPOJO, l'avenir des serveurs d'applications se dirige vers des serveurs personnalisés et dédiés à un domaine particulier. La dernière perspective présentée dans ce chapitre abordera la possibilité de créer des langages de composition d'applications spécifique à un domaine accompagné de modèles de développement et de machines d'exécution dédiées.

1. Synthèse

Cette section met en avant les apports de ce manuscrit. Elle résumera les différents points abordés dans cette thèse.

1.1. L'émergence d'un nouveau besoin

La prolifération d'objets communicants accompagnés par l'essor d'Internet pousse de nombreuses industries à proposer des applications intégrant monde réel et monde informatique. En effet, la jonction entre le monde physique et les processus opérationnels permet de développer de nouveaux services, mais également d'améliorer des systèmes existants. Cependant, ce nouveau contexte change profondément le développement de nouvelles applications. Cette thèse a montré que le dynamisme était devenu un besoin crucial pour les nouvelles applications. En effet, il leur est nécessaire de pouvoir évoluer à l'exécution. Ces adaptations sont nécessaires pour ajouter de nouvelles fonctionnalités à une application, mais également pour en améliorer la qualité ou pour l'adapter à un nouvel environnement d'exécution ou à un nouveau contexte.

Cette thèse a identifié trois raisons majeures qui poussent les applications à évoluer :

- Une application doit pouvoir évoluer durant son exécution : avec la diminution rapide du temps entre deux versions d'un même logiciel, il devient aujourd'hui nécessaire de pouvoir mettre à jour ou de reconfigurer des applications déjà déployées.
- Une application doit s'adapter à son environnement d'exécution : les applications mobiles ou ubiquitaires par exemple doivent sans cesse vérifier la disponibilité des ressources (réseau, dispositifs mobiles…) qu'elles utilisent et s'adapter à ces fluctuations.
- Finalement, certaines applications requièrent de s'adapter en fonction de leur contexte. Par exemple, une application doit s'adapter en fonction des actions d'un utilisateur afin de toujours fournir un service optimal.

Cependant, la mise en place d'application dynamique reste très complexe. Tel que présenté dans ce manuscrit, une application dynamique doit supporter la modification de son architecture durant son exécution. De nombreux travaux adressent le problème des applications dynamiques. Cependant, la plupart des approches pour mettre en place des applications dynamiques entrainent de nombreuses contraintes et sont donc rarement satisfaisantes.

Récemment, l'approche à service est apparue et ouvre de nouvelles voies pour la gestion du dynamisme. Cette approche propose un nouveau moyen pour développer des applications flexibles. Cette approche utilise des services pour construire des applications plus rapidement et composables. L'approche à service permet de développer des applications modulaires telles que le proposait Parnas, il y a 30 ans [139]. Ceci a évidemment un intérêt crucial pour l'intégration de systèmes. Cependant, cette thèse a également montré que cette propriété est également intéressante afin de mettre en place des applications dynamiques.

Papazoglou a défini le concept d'architecture à service étendue. Une architecture à service est un ensemble de technologies et de protocoles permettant la mise en place d'application suivant le paradigme de l'approche à service. Cependant, une architecture à service ne fournit pas les fonctionnalités nécessaires pour composer des services ainsi que de gérer ces applications. Une architecture à service étendue propose justement les fonctionnalités nécessaires pour créer, exécuter et gérer des applications à service. Cette thèse a introduit une variation de l'architecture à service étendue afin de le personnaliser pour exécuter et gérer des applications dynamiques. C'est sur cette architecture à service dynamique étendue que reposent les contributions de cette thèse.

1.2. Approche & Exigences

La problématique de cette thèse est la construction d'applications dynamiques. En effet, la première partie de cette thèse a montré que les approches actuelles étaient insuffisantes. Cette thèse explore donc une nouvelle solution pour créer des applications dynamiques et les administrer. Nous proposons ainsi de combiner des concepts d'architecture logicielle, de la programmation par composant et de l'approche à service. Le but est de proposer un modèle de composant à service permettant de concevoir des applications dynamiques et de les administrer tout en proposant un modèle de développement simple pour le développeur et un modèle de composition intuitif pour l'assembleur. La gestion du dynamisme est alors automatisée à la fois dans le code du développeur et dans la gestion des compositions. Les applications conçues en suivant la méthode proposée supportent automatiquement le dynamisme.

Ce modèle à composant doit fournir les fonctionnalités définies dans les architectures dynamiques étendues introduites dans cette thèse. Cependant, six exigences doivent être remplies afin de fournir une infrastructure permettant la conception, l'implémentation, l'exécution et l'administration de systèmes dynamiques.

La première exigence est la conception à la fois d'un modèle à composant à service et d'une machine d'exécution. Ainsi, tous les éléments du modèle existeront lors de l'exécution. Ceci est primordial pour comprendre lors de l'exécution la structure de l'application. Ainsi, celle-ci sera décrite avec les concepts introduits dans le modèle.

La deuxième exigence concerne l'isolation. En effet, dans les architectures à service traditionnelles, telles qu'OSGi™, les services sont publiés de manière globale. Ainsi, tous les consommateurs peuvent accéder à l'ensemble des services. Cette structure plate ne permet pas d'isoler certains services privés à application. Or, cette isolation est nécessaire afin de composer des applications telles que le proposent les mécanismes de composition des modèles à composant. L'architecture à service dynamique sous-jacente doit donc permettre d'isoler certains services dans un annuaire de service attachée à une application.

La troisième exigence consiste à proposer un modèle de développement le plus simple possible. En effet, la gestion du dynamisme a un impact important sur le code. Ce dynamisme doit donc être masqué afin de proposer un modèle de développement « POJO » aux développeurs.

La quatrième exigence concerne les compositions de service. En effet, afin de concevoir des applications, le modèle à composant à service doit proposer un langage de composition. Cette thèse propose un langage de composition structurelle qui permet d'architecturer des applications en termes de services. La principale différence par rapport aux langages de composition traditionnels vient du fait que la composition ne cible pas d'implémentation particulièrement, mais est justement exprimée en termes de spécifications de service. Ceci a l'avantage de découpler la composition des implémentations. À l'exécution l'infrastructure d'exécution choisira une implémentation disponible.

Une application dynamique voit son architecture évoluer au cours du temps. Il est donc crucial de proposer des mécanismes afin d'introspecter l'état du système et de le reconfigurer. Il doit être possible de remonter l'architecture du système actuelle afin de comprendre les connexions, les services requis manquants… De plus, il doit être possible de reconfigurer une application ou une partie d'une application. Tout comme pour les modèles à composant supportant la reconfiguration dynamique, ces reconfigurations peuvent être effectuées par un administrateur humain ou par un gestionnaire d'adaptation externe. Dans ce cas, le gestionnaire ne modifie que les dépendances de services ce qui évite de manipuler directement l'architecture de l'application.

La dernière exigence concerne la personnalisation du modèle proposé. En effet, celui-ci cible principalement le dynamisme. Il doit donc être possible de l'étendre facilement afin de prendre en charge

d'autres préoccupations non fonctionnelles. Ce mécanisme d'extensibilité doit être disponible à la fois pour les composants atomiques et pour le langage de composition. Fournir un modèle à composant extensible n'est pas nouveau, cependant ces nouveaux traitants peuvent profiter des propriétés non fonctionnelles déjà traitées et donc supporter le dynamisme.

1.3. iPOJO : un modèle à composant à service

Afin de remplir les exigences définies précédemment, cette thèse propose un nouveau modèle à composant à service appelé iPOJO. iPOJO a plusieurs propriétés importantes telles que :

- L'utilisation de concept de l'approche à service dans le modèle à composant
- La définition de dépendances de service au sein des spécifications de service permettant de composer les spécifications de services entre-elles.
- Un modèle de dépendance de service riche et flexible donnant une grande liberté aux utilisateurs.
- La notion de contexte de service permettant l'isolation de service composition par composition.

À partir de ces propriétés, iPOJO fournit un modèle à composant et une machine d'exécution. Le modèle de développement proposé est le plus simple possible, car masque intégralement le dynamisme et les problèmes liés tels que la synchronisation. De plus, iPOJO fournit un langage de composition permettant d'architecturer des applications en termes de service. Le dynamisme est décrit dans la description du composant ou de la composition grâce au modèle de dépendance de service fourni. Dans les deux cas, ce dynamisme est intégralement géré par la machine d'exécution. En effet, à la fois l'évolution, le dynamisme issu de l'environnement et le changement de contexte sont gérés par iPOJO. La contribution de cette thèse peut être positionnée en utilisant la même caractérisation que celle définit dans le chapitre 3 (Tableau 34).

Critères	
Initialisation des adaptations	Tous types d'évènements (contexte, environnement, administrateur)
Localisation de la logique d'adaptation	Dans la description de l'architecture de l'application, ainsi que dans le modèle à composant
Gestion des politiques d'adaptation	Politique variable (reconfigurable)
Positionnement du gestionnaire d'adaptation	Dans le conteneur des instances
Mécanisme utilisé pour l'adaptation	Capacité du modèle à composant, utilisation de la réflexion Java, manipulation de bytecode
Gestion des interruptions	Détection des régions impliquées (blocage des transactions), Minimisation des interruptions

Tableau 34. Positionnement d'iPOJO par rapport à la caractérisation définit dans cette thèse

De plus, iPOJO fournit des mécanismes d'introspection permettant de remonter la structure actuelle d'un système. Ainsi, il est possible de visualiser les instances de composants et leurs interconnexions. Ceci permet entre autres de déterminer des problèmes de résolution de service, ou d'état incohérent. iPOJO fournit également des mécanismes de reconfiguration. Cependant, ceux-ci agissent principalement sur les dépendances de service et ne permettent pas de manipuler directement l'architecture d'une application. En effet, la reconfiguration des dépendances de service a un impact direct sur l'architecture, sans introduire de dépendance forte avec une implémentation ou un fournisseur de service spécifique.

Enfin, iPOJO définit des mécanismes d'extensibilité. Ceux-ci permettent de personnaliser le modèle afin de rajouter d'autres préoccupations telles que la communication par évènements ou la persistance. Ces nouveaux handlers peuvent étendre à la fois le modèle des composants atomiques et le modèle de composition. De plus, ils sont également gérés par iPOJO, ce qui leur permet d'être dynamiques de manière transparente.

L'intégralité de la proposition a été implémentée. L'implémentation est disponible sous licence Apache et est hébergée en tant que sous projet du projet Apache Felix. De plus, iPOJO est actuellement utilisé dans différents contextes tels que les passerelles résidentielles, les serveurs d'applications, et les téléphones portables. Cette thèse a décrit ces différents cas d'utilisation validant le modèle à composant à service proposé.

2. Perspectives

L'utilisation d'iPOJO dans différents domaines montre la facilité d'utilisation du modèle proposé dans cette thèse. Cependant, certaines questions sont restées ouvertes. Cette section décrit trois perspectives de travail envisagées.

2.1. Support du déploiement

Le déploiement est l'ensemble des activités qui permettent de rendre un système disponible [56, 201]. Le processus de déploiement est généralement constitué de différentes activités s'exécutant soit sur le site du fournisseur du logiciel, soit sur le site de l'utilisateur, soit sur les deux sites. Cependant, comme le déploiement d'une application est généralement un processus spécifique, il est très complexe de définir exactement chacune des activités. Néanmoins, les experts de ce domaine s'accordent sur les activités faisant parti du processus de déploiement [202] : la mise à disposition, l'installation, l'activation, la désactivation, l'adaptation, la mise à jour, la désinstallation et enfin le retrait.

Une des perspectives du travail effectué dans cette thèse est de fournir une infrastructure facilitant le déploiement d'application dynamique. Le déploiement est devenu un problème majeur aujourd'hui. En effet, fournir des solutions fiables et efficaces pour faciliter le déploiement d'application est devenu un enjeu crucial pour les fournisseurs d'applications. Le déploiement d'application dynamique est un domaine faiblement investigué. Mais, le déploiement de ce type d'application ouvrirait de nouvelles voies. Par exemple, la reconfiguration dynamique d'une application ne se ferait plus en fonction des éléments présents dans l'environnement d'exécution, mais en fonction des ressources mises à dispositions dans des dépôts.

Actuellement, iPOJO ne propose pas d'infrastructure pour faciliter le déploiement. Ceci est une limite importante. En effet, iPOJO fournit à la fois un modèle à composant (modèle de développement et langage de composition) et une machine d'exécution. Le processus pour installer et administrer les applications développées doit être fait manuellement.

La plate-forme d'exécution sous-jacente à iPOJO (OSGi™) est une plate-forme de déploiement qui supporte les activités de déploiement qui s'effectue sur le site client. De plus, l'*OSGi Bundle Repository* (OBR) [203] définit un format afin de décrire des ressources. Il est ainsi possible de mettre en place des dépôts mettant à disposition des *bundles*. Ces *bundles* peuvent ensuite être installés sur la plate-forme OSGi™. Le moteur de déploiement accompagnant l'OBR résout les dépendances[44] que décrit une ressource en cours d'installation.

Plusieurs outils s'appuyant sur les capacités d'OSGi™ et sur OBR sont envisageables afin de faciliter le déploiement d'application dynamique développé avec iPOJO. Tout d'abord, il serait possible de compléter les annuaires de service avec les ressources *potentiellement* disponibles (c'est-à-dire mise à disposition dans des dépôts). Ceci permettrait de résoudre les dépendances de service avec les implémentations et les fournisseurs actuellement présents mais également avec ceux qui peuvent être installés. Une des contraintes à prendre en compte dans cette extension est la possibilité d'échec lors du déploiement.

[44] Les dépendances d'une ressource sont décrites de manière générique, mais sont généralement des requis de paquetages Java (dépendance de code).

Une deuxième infrastructure intéressante serait de fournir des outils automatisant la description de composant et d'applications. De cette manière, la création de dépôts OBR, contenant des composants et des applications développées avec iPOJO, serait grandement facilitée. Ces dépôts proposeraient ensuite l'ensemble des implémentations de service et l'ensemble des fournisseurs de services qu'ils contiennent. Lors de l'installation d'une application, le moteur de déploiement de l'OBR peut se reposer sur les descriptions générées afin de déployer l'ensemble des *ressources* requises par l'application.

Cependant, lors du déploiement de composition de service, l'OBR ne peut pas décider s'il est nécessaire d'installer une implémentation de service ou un fournisseur. Fournir un mécanisme reposant sur l'OBR, mais dédié au déploiement de composition serait également une piste intéressante à suivre. Il serait alors possible de déployer toutes les applications développées avec iPOJO de manière automatique.

2.2. Plate-forme d'exécution pour applications autonomiques et sensibles au contexte

Le but de l'informatique autonomique est de produire des applications *autonomes* c'est-à-dire qui s'administrent toute seule. Les applications autonomiques sont des applications gérées par des gestionnaires autonomiques. Ces gestionnaires supervisent l'exécution de l'application afin de lui ajouter des propriétés comme l'auto-optimisation, l'autoprotection ou l'autoréparation.

Les applications sensibles au contexte sont des applications adaptant leur comportement en fonction de leur contexte. Celui-ci peut être le contexte de l'utilisateur. Le but des applications sensibles au contexte est d'optimiser en permanence les applications en fonction de leur contexte. Par exemple, une application devant délivrer un message à un utilisateur choisira le meilleur support (son, texte, flash lumineux) en fonction du contexte de cet utilisateur.

Pour ces deux classes d'application, la machine d'exécution est très importante. En effet, elle doit fournir des capacités d'adaptations dynamiques. De plus, elle doit permettre d'instrumenter l'environnement afin de déterminer l'état du système, permettant ainsi de déterminer le contexte pour les applications sensibles au contexte. Les adaptations à effectuer sont décidées par le gestionnaire d'adaptation et la logique d'adaptation qui peuvent différer en fonction du comportement recherché.

L'infrastructure offerte par iPOJO est très prometteuse pour le développement et l'exécution de ce type d'application. iPOJO fournit les mécanismes basiques pour l'exécution de telles applications. L'une des perspectives à iPOJO est donc d'expérimenter le développement de telles applications, c'est-à-dire mettre en place une boite à outils permettant la mise en place rapide d'applications autonomiques et sensibles au contexte. En effet, ces applications doivent être dynamiques. Or cet aspect est intégralement géré par la machine d'exécution d'iPOJO. De plus, iPOJO permet la *manipulation* de la structure de l'application grâce à la reconfiguration de dépendance de service ainsi que la reconfiguration des instances. Ces deux mécanismes sont justement nécessaires lors de la mise en place d'applications adaptables dynamiquement.

De plus, ces applications doivent pour introspecter l'environnement afin de construire une représentation du système qui servira de base de raisonnement pour détecter les adaptations à effectuer. Pour cela, iPOJO fournit un mécanisme permettant d'introspecter l'état du système. Cependant, les informations remontées ne sont pas forcément suffisantes. En effet, celle-ci ne concerne que la structure des applications et l'état des instances. Elles doivent donc être complétées avec des informations spécifiques à l'application telles que des données sur le contexte, ou des informations spécifiques sur un composant. L'infrastructure fournit par iPOJO, permettant de gérer le contexte (via les sources de contexte), fournit des mécanismes de base, mais ne s'intéresse pas à la récupération de ces données ni à la représentation du contexte. Afin de compléter ces informations, le développement d'handler spécifique est une voie à investiguer. Des travaux préliminaires ont déjà été effectués et sont prometteurs [188, 204].

2.3. Mise en place d'infrastructure centrée domaine

L'ingénierie des domaines consiste à identifier, capturer et structurer l'expérience acquise dans le développement des solutions dans un domaine métier particulier et de fournir les moyens pour la réutilisation de ces informations lors des futurs développements. L'objectif principal est d'utiliser ces informations pour faciliter le développement des nouveaux systèmes dans le même contexte.

Cependant, l'écart entre les requis et les contraintes analysées et l'infrastructure de développement et d'exécution sont grands. Récemment, la combinaison de l'ingénierie par les modèles et de l'approche générative a ouvert la voie à la création d'environnement de développement spécialisé pour un domaine métier [205]. Grâce à ces outils, les architectures et développeurs manipulent des concepts métiers tout au long du développement. Bien que ces outils facilitent le développement, il leur est difficile d'adresser l'environnement d'exécution.

Parallèlement au développement de ces outils, la vision monolithique des serveurs d'applications s'est effritée au profit de serveurs plus flexibles et dédiés à des domaines particuliers. Ainsi, il est fréquent de rencontrer des intergiciels dédiés à un domaine, fournissant les services techniques et les capacités requises dans ce domaine. Cette tendance générale va permettre à terme de proposer une infrastructure complète pour un métier donné. Cette infrastructure sera composée d'un environnement de développement et de composition ainsi que d'une machine d'exécution spécialisée.

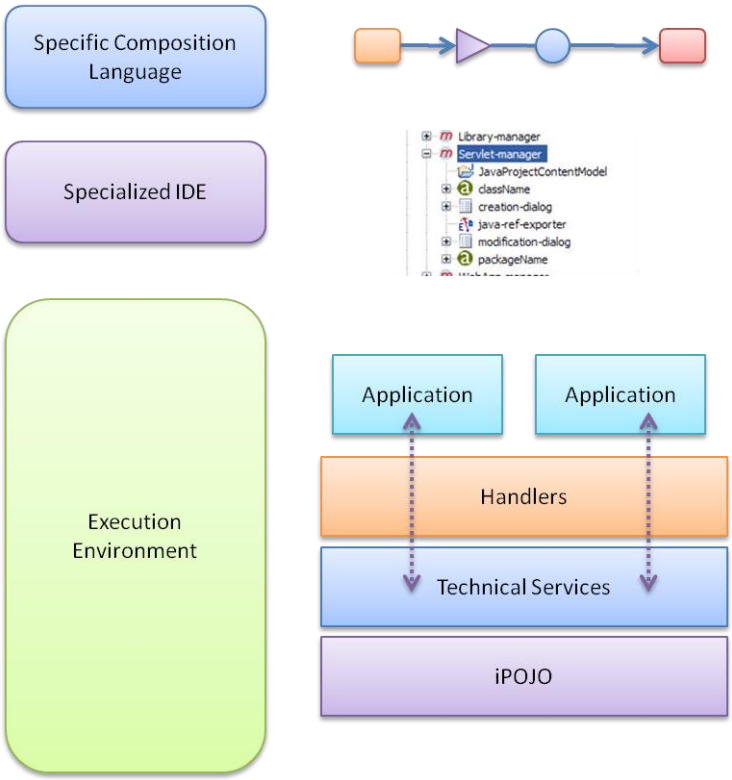


Figure 135. Infrastructure centrée domaine

La mise en place de ce type d'infrastructure est une perspective intéressante. Les mécanismes offerts par iPOJO permettent de facilement mettre en place ce type d'infrastructure. En effet, le modèle de composition métier peut facilement être traduit en composition structurelle. De ce fait, les applications deviennent dynamiques et possède des propriétés d'isolation intéressantes. Ensuite, le développement d'application est grandement facilité par l'utilisation d'handler spécialisé masquant les interactions avec les services techniques. On obtient ainsi l'environnement d'exécution spécialisé (Figure 135).

Chapitre 12
Bibliographie

[1] G. Bierman, et al., “Formalizing Dynamic Software Updating,” *published in the proceedings of the Workshop on Unexpected Software Evolution*, 2003.

[2] B. Myers, “Using Multiple Devices Simultaneously for Display and Control,” *IEEE Personal Communications*, vol. 7, no. 5, 2000, pp. 62-65.

[3] M. Malek, “The NOMADS Republic - a case for ambient service oriented computing,” *published in the proceedings of the Symposium on Service-Oriented System Engineering*, 2005, pp 9-12.

[4] M. Weiser, “The computer for the 21st century,” *Scientific American*, vol. 265, no. 3, 1991, pp. 66-75.

[5] P.H. Salus, *Casting the Net - From ARPANET to Internet and Beyond*, Addison-Wesley, 1995, p. 299.

[6] T. O'Reilly, “What Is Web 2.0 - Design Patterns and Business Models for the Next Generation of Software,” *Oreilly Network*, July 2005; http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

[7] J. Hendler, “Web 3.0: Chicken Farms on the Semantic Web,” *IEEE Computer*, vol. 41, no. 1, 2008, pp. 106-108.

[8] J.-B. Waldner, *Nano-informatique et intelligence ambiante*, 2007.

[9] M. Weiser, “Ubiquitous Computing,” *IEEE Computer*, vol. 26, no. 10, 1993, pp. 71-72.

[10] P.D. Dick, *Ubik*, Garden City, N.Y., Doubleday, 1969.

[11] I. Asimov, *The Rest of the Robots*, Doubleday & Company, 1964.

[12] J. Tati, *Mon Oncle*, 1958.

[13] M. Hawley, et al., “Things that think,” *Personal and Ubiquitous Computing*, vol. 1, no. 1, 1997, pp. 13-20.

[14] MIT, “Project Oxygen - Pervasive, Human-Centered Computing,” 2004; http://oxygen.csail.mit.edu/publications/Oxygen.pdf.

[15] L. Duboc, et al., “A framework for modelling and analysis of software systems scalability,” *published in the proceedings of the International conference on Software Engineering*, 2006, pp 949-952.

[16] M. Hill, “What is scalability?,” *Computer Architecure News*, vol. 18, no. 4, 1990, pp. 18-21.

[17] P. Karger and R. Schell, “Thirty Years Later: Lessons from the Multics Security Evaluation,” *published in the proceedings of the Annual Computer Security Applications Conference*, 2002.

[18] J. Kephart and D. Chess, “The Vision of Autonomic Computing,” *IEEE Computer*, vol. 36, no. 1, 2003, pp. 41-50.

[19] P. Horn, *Autonomic Computing : IBM’s Perspective on the State of Information Technology*, IBM, 2001; http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[20] G. Deen , et al., “Optimal Grid : A research prototype of grid-enabled middleware designed to hide complexities of partitioning, distributing, and load balancing.,” 2003; http://www.alphaworks.ibm.com/tech/optimalgrid.

[21] M. Agarwal, et al., “Automate: Enabling Autonomic Applications On The Grid,” 2003; http://automate.rutgers.edu/.

[22] M. Desertot, et al., “Autonomic management of J2EE edge servers,” *published in the proceedings of the International Workshop on Middleware for Grid Computing*, 2005.

[23] P. Lalanda and J. Bourcier, “Towards autonomic residential gateways,” *published in the proceedings of the IEEE International Conference on Pervasive Services*, 2006, pp 329-332.

[24] R. Sterritt and M. Hinchey, “Autonomic computing - panacea or poppycock?,” *published in the proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems*, 2005, pp 535-539.

[25] IBM, “Autonomic Computing Toolkit,” 2004; http://www.ibm.com/developerworks/autonomic/overview.html.

[26] D. Xiangdong, et al., “Autonomia: an autonomic computing environment,” *published in the proceedings of the IEEE International Conference on Performance, Computing, and Communications*, 2003, pp 61-68.

[27] J. Lee, et al., “Enterprise integration with ERP and EAI,” *Communication of the ACM*, vol. 46, no. 2, 2003, pp. 54-60.

[28] S. Vinoski, “Integration with Web Services,” *IEEE Internet Computing*, vol. 7, no. 6, 2003, pp. 75-77.

[29] G. Wiederhold, “Mediators in the architecture of future information systems,” *IEEE Computer*, vol. 25, no. 3, 1992, pp. 38-49.

[30] IBM, “WebSphere Enterprise Service Bus & WebSphere Adapter,” 2004; http://www-306.ibm.com/software/integration/wsesb/.

[31] C. Herault, et al., “A distributed service-oriented mediation tool,” *published in the proceedings of the IEEE International Conference on Services Computing*, 2007, pp 403-409.

[32] P. Lalanda, et al., “Asynchronous mediation for integrating business and operational processes,” *IEEE Internet Computing*, vol. 10, no. 1, 2006, pp. 56-64.

[33] W.T. Tsai, et al., “Service-oriented system engineering (SOSE) and its applications to embedded system development,” *Service Oriented Computing and Applications*, vol. 1, no. 1, 2007, pp. 3-17.

[34] M. Satyanarayanan, “Pervasive computing: vision and challenges,” *IEEE Personal Communications*, vol. 8, no. 4, 2001, pp. 10-17.

[35] R. Want, et al., “The active badge location system. ,” *ACM Transactions on Information Systems*, vol. 10, 1992.

[36] B. Schilit, et al., “Context-aware computing applications,” *published in the proceedings of the Workshop on Mobile Computing Systems and Applications*, 1994, pp 85-90.

[37] J. Krumm, et al., “International Conference on Ubiquitous Computing,” *published in the proceedings of the International Conference on Ubiquitous Computing*, 2007.

[38] Google, “What is Android ?,” 2007; http://code.google.com/android/what-is-android.html.

[39] Symbian, “Symbian OS Version 9.4,” 2007; http://www.symbian.com/files/rx/file9468.pdf.

[40] M. Desertot, et al., “A Dynamic Service-Oriented Implementation for Java EE Servers,” *published in the proceedings of the IEEE International Conference on Services Computing*, 2006, pp 159-166.

[41] S. Ducasse, et al., “Seaside: a flexible environment for building dynamic web applications,” *IEEE Software*, vol. 24, no. 5, 2007, pp. 56-63.

[42] OSGi Alliance, *OSGi Service Platform Core Specification*, OSGi Alliance, 2005; http://osgi.org.

[43] Sun Microsystems, “JSR-3 Java Management Extensions (JMX) ” 2006; http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html.

[44] R.S. Fabry, “How to design a system in which modules can be changed on the fly,” *published in the proceedings of the International Conference on Software Engineering*, 1976, pp 470-476.

[45] J. Estublier, et al., “Impact of software engineering research on the practice of software configuration management,” *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 4, 2005, pp. 383-430.

[46] J. Estublier and G. Vega, “Reconciling software configuration management and product data management,” *published in the proceedings of the European Software Engineering Conference - Foundation on Sofware Engeenering*, 2007.

[47] A. Munch-Ellingsen, et al., “Argos, an Extensible Personal Application Server,” *published in the proceedings of the ACM Middleware*, 2007, pp 21-41.

[48] P.K. McKinley, et al., “Composing adaptive software,” *IEEE Computer*, vol. 37, no. 7, 2004, pp. 56-64.

[49] C. Ghezzi, et al., *Fundamentals of Software Engineering.*, Prentice Hall, 1991.

[50] L. Erlikh, “Leveraging Legacy System Dollars for E-Business,” *IT Professional*, vol. 2, no. 3, 2000, pp. 17-23.

[51] D. Garlan and B. Schmerl, “Model-based adaptation for self-healing systems,” *published in the proceedings of the Workshop on Self-healing systems*, 2002.

[52] A.G. Ganek and T.A. Corbi, “The drawing of the Autonomic Computing era,” *IBM Systems Journal*, vol. 41, no. 1, 2003, pp. 5-18.

[53] P. Oreizy, et al., “An architecture-based approach to self-adaptive software,” *IEEE Intelligent Systems*, vol. 14, no. 3, 1999, pp. 54-62.

[54] S.S. Yau, et al., “Reconfigurable context-sensitive middleware for pervasive computing,” *IEEE Pervasive Computing* vol. 1, no. 3, 2002, pp. 33-40.

[55] A.V.d. Hoek and A.L. Wolf, “Software release management for component-based software,” *Software: Practice and Experience*, vol. 33, no. 1, 2003, pp. 77-98.

[56] R.S. Hall, et al., “A cooperative approach to support software deployment using the software dock,” *published in the proceedings of the International Conference on Software Engineering*, 1999.

[57] J. Lee and K.C. Kang, “A feature-oriented approach to developing dynamically reconfigurable products in product line engineering,” *published in the proceedings of the International Software Product Line Conference*, 2006, pp 10-20.

[58] S. Hallsteinsen, et al., “Using product line techniques to build adaptive systems,” *published in the proceedings of the International Software Product Line Conference*, 2006.

[59] H. Gomaa and M. Hussein, “Dynamic Software Reconfiguration in Software Product Families,” *Software Product-Family Engineering*, volume 3014, 2004, pp. 435-444.

[60] J. Kramer and J. Magee, “The Evolving Philosophers Problem: Dynamic Change Management,” *IEEE Transactions on Software Engineering*, vol. 16, no. 11, 1990, pp. 1293-1306.

[61] C. Hofmeister and J. Purtilo, “Dynamic reconfiguration in distributed systems: adapting software modules for replacement,” *published in the proceedings of the International Conference on Distributed Computing Systems*, 1993, pp 101-110.

[62] M. Cremene, et al., “Autonomic Adaptation based on Service-Context Adequacy Determination,” *Electronic Notes in Theoretical Computer Science*, vol. 189, 2007, pp. 35-50.

[63] J. Kramer and J. Magee, “Self-Managed Systems : an Architectural Challenge,” *published in the proceedings of the Future of Software Engineering*, 2007.

[64] P. Oreizy and R.N. Taylor, “On the role of software architectures in runtime system reconfiguration,” *IEE Proceedings Software*, vol. 145, no. 5, 1998, pp. 137-145.

[65] J. Dowling and V. Cahill, “The K-Component Architecture Meta-model for Self-Adaptive Software,” *published in the proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, 2001, pp 81-88.

[66] G.S. Blair, et al., “The role of software architecture in constraining adaptation in component-based middleware platforms,” *published in the proceedings of the ACM International Conference on Distributed Systems Platforms*, 2000.

[67] D. Perry and A.L. Wolf, “Foundations for the study of software architecture,” *SIGSOFT Software Engineering Notes*, vol. 17, no. 4, 1992, pp. 40-52.

[68] D. Garlan and D. Perry, “Introduction to the Special Issue on Software Architecture,” *IEEE Transactions on Software Engineering*, vol. 21, no. 4, 1995, pp. 269-274.

[69] D. Garlan and M. Shaw, “An introduction to software architecture,” *published in the proceedings of the Advances in Software Engineering and Knowledge Engineering*, 1993.

[70] G. Kiczales, “Aspect-oriented programming,” *ACM Computing Surveys*, vol. 28, no. 4, 1996.

[71] G.S. Blair, et al., “The Design and Implementation of Open ORB 2,” *IEEE Distributed Systems Online*, vol. 2, no. 6, 2001, pp. 1-33.

[72] N. Medvidovic and R.N. Taylor, “A framework for classifying and comparing architecture description languages,” *published in the proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1997, pp 60-76.

[73] Y. Vandewoude, et al., “An alternative to Quiescence: Tranquility,” *published in the proceedings of the IEEE International Conference on Software Maintenance*, 2006, pp 73-82.

[74] Y. Vandewoude, et al., “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates,” *IEEE Transactions on Software Engineering*, vol. 33, no. 12, 2007, pp. 856-868.

[75] M. Wermelinger, “Specification of Software Architecture Reconfiguration,” Universidade Nova de Lisboa, 1999.

[76] S. Cheng, et al., “Software Architecture-Based Adaptation for Pervasive Systems,” *published in the proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, 2002.

[77] F.J. Silva, et al., *Modeling Dynamic Adaptation of Distributed Systems*, University of Illinois at Urbana-Champaign, 2000.

[78] P. Pal, et al., “Using QDL to specify QoS aware distributed (QuO) application configuration,” *published in the proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2000, pp 310-319.

[79] F. Kon, et al., “Dynamic resource management and automatic configuration of distributed component systems,” *published in the proceedings of the Conference on USENIX Conference on Object-Oriented Technologies and Systems*, 2001.

[80] K. Moazami-Goudarzi and J. Kramer, “Maintaining Node Consistency in the Face of Dynamic Change,” *published in the proceedings of the International Conference on Configurable Distributed Systems*, 1996.

[81] J. Bosch, “Product-line architectures in industry: a case study,” *published in the proceedings of the International Conference on Software Engineering*, 1999, pp 544-554.

[82] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*, ACM Press/Addison-Wesley Publishing Co., 2000, p. 354.

[83] P. Oreizy, et al., “Architecture-based runtime software evolution,” *published in the proceedings of the International Conference on Software Engineering*, 1998, pp 177-186.

[84] N. Badr, et al., “Policy-based autonomic control service,” *published in the proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004, pp 99-102.

[85] S.R. White, et al., “An architectural approach to autonomic computing,” *published in the proceedings of the International Conference on Autonomic Computing*, 2004, pp 2-9.

[86] L. Baresi, et al., “Style-based refinement of dynamic software architectures,” *published in the proceedings of the IEEE Working Conference on Software Architecture*, 2004, pp 155-164.

[87] D. Le Metayer, “Describing software architecture styles using graph grammars,” *IEEE Transactions on Software Engineering*, vol. 24, no. 7, 1998, pp. 521-533.

[88] G. Taentzer, et al., “Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems,” *published in the proceedings of the Theory and Application of Graph Transformations*, 2000.

[89] M. Wermelinger and J.L. Fiadeiro, “A graph transformation approach to software architecture reconfiguration,” *Science of Computer Programming*, vol. 44, no. 2, 2002, pp. 133-155.

[90] J. Magee and J. Kramer, “Dynamic structure in software architectures,” *published in the proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1996.

[91] D.C. Luckham and J. Vera, “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, vol. 21, no. 9, 1995, pp. 717-734.

[92] E. Gamma, et al., *Design patterns: Elements of reusable object-oriented software*, 1995.

[93] D.C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible and Maintainable ORB Middleware,” *IEEE Communications Magazine*, 1999.

[94] M. Aksit, et al., “Abstracting Object Interactions Using Composition Filters,” *published in the proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, 1994, pp 152-184.

[95] L. Bergmans and M. Aksit, “Composing crosscutting concerns using composition filters,” *Communication of the ACM*, vol. 44, no. 10, 2001, pp. 51-57.

[96] Sun Microsystems, “The essentials of Filters,” http://java.sun.com/products/servlet/Filters.html.

[97] G. Kiczales, et al., “Aspect-oriented programming,” *published in the proceedings of the European Conference on Object-Oriented Programming*, 1997.

[98] J. Bosch, “Superimposition: a component adaptation technique,” *Information and Software Technology*, vol. 41, no. 5, 1999, pp. 257-273.

[99] A. Adi, et al., “Modeling and Monitoring Dynamic Dependency Environments,” *published in the proceedings of the IEEE International Conference on Services Computing*, 2005, pp 208-216.

[100] L. Bergmans and M. Akşit, “Principles and design rationale of composition filters,” *Aspect-Oriented Software Development*, 2005, pp. 63-95.

[101] R. Pawlak, et al., “ Dynamic wrappers: handling the composition issue with JAC,” *published in the proceedings of the International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, 2001, pp 56-65.

[102] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Longman Publishing Co., Inc., 2002, p. 448.

[103] G. Heineman and W. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley Professional, 2001, p. 880.

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

[105] S. Microsystems, “Enterprise JavaBeans Technology,” http://java.sun.com/products/ejb/index.jsp.

[106] Spring Source, “Spring Framework,” http://www.springframework.org.

[107] R. Van Ommering, et al., “The Koala component model for consumer electronics software,” *Computer*, vol. 33, no. 3, 2000, pp. 78-85.

[108] M. Fowler, “Inversion of Control Containers and the Dependency Injection pattern,” 2004; http://www.martinfowler.com/articles/injection.html.

[109] J.A. Rowson and A. Sangiovanni-Vincentelli, “Interface-based Design,” *published in the proceedings of the Design Automation Conference*, 1997, pp 178-183.

[110] M. Lackner, et al., “Supporting Design by Contract in Java,” *Journal of Object Technology*, vol. 1, no. 3, 2002, pp. 57-76.

[111] A. Beugnard, et al., “Making components contract aware,” *IEEE Computer*, vol. 32, no. 7, 1999, pp. 38-45.

[112] E.I. Organick, *The multics system: an examination of its structure*, MIT Press, 1972, p. 423.

[113] C. Escoffier, et al., “Developing an OSGi-like service platform for .NET,” *published in the proceedings of the IEEE Consumer Communications and Networking Conference*, 2006, pp 213-217.

[114] G. Kiczales, et al., *The Art of the Metaobject Protocol*, MIT Press, 1991, p. 350.

[115] M. Shapiro, “Structure and Encapsulation in Distributed Systems: the Proxy Principle,” *published in the proceedings of the International Conference on Distributed Systems*, 1986, pp 198-204.

[116] Sun Microsystems, “Dynamic Proxy Classes,” 1999; http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[117] J. de Oliveira Guimaraes, “Reflection for statically typed languages,” *published in the proceedings of the European Conference on Object-Oriented Programming*, 1998, pp 460-461.

[118] A. Popovici, et al., “Dynamic weaving for aspect-oriented programming,” *published in the proceedings of the Conference on Aspect-oriented Software Development*, 2002.

[119] B. Redmond and V. Cahill, “ Iguana/J: Towards a dynamic and efficient reflective architecture for Java,” *published in the proceedings of the Workshop on Reflection and Metalevel Architectures*, 2000.

[120] S.M. Sadjadi, et al., “TRAP/J: Transparent Generation of Adaptable Java Programs,” *published in the proceedings of the International Symposium on Distributed Objects and Applications*, 2004.

[121] T. Batista and N. Rodriguez, “Dynamic reconfiguration of component-based applications,” *published in the proceedings of the Software Engineering for Parallel and Distributed Systems*, 2000, pp 32-39.

[122] F. Plasil, et al., “SOFA/DCUP: architecture for component trading and dynamic updating,” *published in the proceedings of the Fourth International Conference on Configurable Distributed Systems*, 1998, pp 43-51.

[123] E. Bruneton, et al., *The FRACTAL component model*, Specification fractal, Objectweb, 2004; http://fractal.objectweb.org.

[124] R. Bialek and E. Jul, “A Framework for Evolutionary, Dynamically Updatable, Component-Based Systems,” *published in the proceedings of the International Conference on Distributed Computing Systems Workshops*, 2004, pp 326-331.

[125] J. Hillman and I. Warren, “An Open Framework for Dynamic Reconfiguration,” *published in the proceedings of the International Conference on Software Engineering*, 2004.

[126] I. Warren, et al., “An Automated Formal Approach to Managing Dynamic Reconfiguration,” *published in the proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp 37-46.

[127] D. Soni, et al., “Software architecture in industrial applications,” *published in the proceedings of the International Conference on Software Engineering*, 1995.

[128] K.-K. Lau, et al., “Exogenous connectors for software components,” *published in the proceedings of the Component-based software engineering* 2005, pp 90-106.

[129] K.-K. Lau and W. Zheng, “A Taxonomy of Software Component Models,” *published in the proceedings of the Conference on Software Engineering and Advanced Applications*, 2005, pp 88-95.

[130] P. Oreizy, *Issues in the Runtime Modification of Software Architectures*, Department of Information and Computer Science, University of California, Irvine, 1996.

[131] N. Medvidovic and R. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages,” *IEEE Transactions on Software Engineering*, vol. 26, no. 1, 2000, pp. 70-93.

[132] J. Magee, et al., “A constructive development environment for parallel and distributed programs,” *published in the proceedings of the International Workshop on Configurable Distributed Systems*, 1994, pp 4-14.

[133] J. Magee, et al., “Specifying Distributed Software Architectures,” *published in the proceedings of the European Software Engineering Conference*, 1995, pp 137-153.

[134] I. Georgiadis, *Design Issues in the Mapping of the Darwin ADL to Java using RMI as the Communication Substrate*, Department of Computer Science, Imperial College, London, 1999.

[135] R. Allen, et al., “Specifying and Analyzing Dynamic Software Architectures,” *published in the proceedings of the Conference on Fundamental Approaches to Software Engineering* 1998.

[136] P. Oreizy, et al., “Software Architecture and Component Technologies: Bridging the Gap,” *published in the proceedings of the Workshop on Compositional Software Architectures*, 1998.

[137] M.P. Papazoglou and D. Georgakopoulos, *Service-oriented computing*, ACM, 2003.

[138] F. Leymann, “The (Service) Bus: Services Penetrate Everyday Life,” *published in the proceedings of the International Conference Service Oriented Computing*, 2005, pp 12-20.

[139] T. Magedanz, et al., “Evolution of SOA Concepts in Telecommunications,” *IEEE Computer*, vol. 40, no. 11, 2007, pp. 46-50.

[140] T. Margaria and B. Steffen, “Service Engineering: Linking Business and IT,” *published in the proceedings of the Annual IEEE/NASA Software Engineering Workshop*, 2006, pp 33-36.

[141] D.L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communication of the ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.

[142] M.P. Papazoglou and W.J.V.D. Heuvel, *Service-Oriented Computing : State of art and open research issues*, Research Agenda, Tilburg University (The Netherlands), 2003.

[143] M.N. Huhns and M.P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, vol. 9, no. 1, 2005, pp. 75-81.

[144] D. Jordan and J. Evdemon, *Web Services Business Process Execution Language Version 2.0*, OASIS, 2007; http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[145] M. Beisiegel, et al., *SCA Service Component Architecture: Assembly Model Specification*, Open Service Oriented Architecture, 2007; http://www.osoa.org/display/Main/Service+Component+Architecture+Home.

[146] P.V. Mockapetris, *Domain names - implementation and specification*, RFC Editor, 1987.

[147] W.K. Edwards, "Discovery systems in ubiquitous computing," *IEEE Pervasive Computing*, vol. 5, no. 2, 2006, pp. 70-77.

[148] S. Patil and E. Newcomer, "ebXML and Web services," *IEEE Internet Computing*, vol. 7, no. 3, 2003, pp. 74-82.

[149] S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 35, no. 2, 1997, pp. 46-55.

[150] Object Management Group, "CORBA 3.1," 2008; http://www.omg.org/spec/CORBA/3.1/.

[151] M. Marazakis, et al., "Aurora: An Architecture for Dynamic and Adaptive Work Sessions in Open Environments," *published in the proceedings of the International Conference on Database and Expert Systems Applications*, 1998.

[152] J. Magee, et al., "Composing Distributed Objects in CORBA," *published in the proceedings of the International Symposium on Autonomous Decentralized Systems*, 1997.

[153] J. Waldo, "The Jini architecture for network-centric computing," *Communications of the ACM*, vol. 42, no. 7, 1999, pp. 76-82.

[154] J. Waldo, *The Jini Specifications*, Addison-Wesley, 2000, p. 688.

[155] Y. Huang, "JISGA: A Jini-Based Service-Oriented Grid Architecture," *International Journal on High Performance Computing Applications*, vol. 17, no. 3, 2003, pp. 317-327.

[156] Sprint, "Sprint Titan," 2008; http://developer.sprint.com/getDocument.do?docId=98336.

[157] Objectweb, "Jonas: Java Open Application Server," 2007; http://wiki.jonas.objectweb.org/xwiki/bin/view/Main/WebHome.

[158] O. Gruber, et al., "The Eclipse 3.0 platform: adopting OSGi technology," *IBM System Journal*, vol. 44, no. 2, 2005, pp. 289-299.

[159] L.F. Andrade and J.L. Fiadeiro, "Coordination technologies for web-services," *published in the proceedings of the Workshop on Object-Oriented Web Services*, 2001.

[160] F. Curbera, et al., "The next step in Web services," *Communications of the ACM*, vol. 46, no. 10, 2003, pp. 29-34.

[161] A. Dogac, et al., “Improving the Functionality of UDDI Registries through Web Service Semantics,” *published in the proceedings of the Workshop on Technologies for E-Services*, 2002.

[162] M. Paolucci, et al., “Importing the Semantic Web in UDDI,” *published in the proceedings of the International Workshop on Web Services, E-Business, and the Semantic Web*, 2002.

[163] C. Peltz, “Web services orchestration and choreography,” *Computer*, vol. 36, no. 10, 2003, pp. 46-52.

[164] Microsoft, “Universal Plug and Play Device Architecture,” 2000; http://www.upnp.org/download/UPnPDA10_20000613.htm.

[165] Universal Plug and Play Forum, “About the Universal Plug and Play Forum,” 1999; http://www.upnp.org/forum/default.htm.

[166] Y. Goland, et al., *Simple Service Discovery Protocol v1.0*, IETF, 1999.

[167] J. Cohen, et al., *General Event Notification Architecture Base*, UPnP Forum, 1999; http://www.upnp.org/download/draftcohen-gena-client-01.txt.

[168] Y. Goland, *Multicast and Unicast UDP HTTP Messages*, IETF, 1999; http://tools.ietf.org/html/draft-goland-http-udp-00.

[169] Microsoft, “WSDAPI Client Application and Device Host Development ” 2008; http://msdn.microsoft.com/en-us/library/bb821828.aspx.

[170] N. Bussière, et al., “Optimized Contextual Discovery of Web Services for Devices,” *published in the proceedings of the International Workshop on Context Modeling and Management for Smart Environments*, 2007.

[171] E. Zeeb, et al., “Lessons learned from implementing the Devices Profile for Web Services,” *published in the proceedings of the IEEE-IES Digital EcoSystems and Technologies Conference*, 2007, pp 229-232.

[172] J. Yang, “Web service componentization,” *Communication of the ACM*, vol. 46, no. 10, 2003, pp. 35-40.

[173] H. Cervantes and R.S. Hall, “Autonomous adaptation to dynamic availability using a service-oriented component model,” *published in the proceedings of the International Conference on Software Engineering*, 2004, pp 614-623.

[174] R.S. Hall and H. Cervantes, “Gravity: supporting dynamically available services in client-side applications,” *SIGSOFT Software Engineering Notes*, vol. 28, no. 5, 2003, pp. 379-382.

[175] OSGi Alliance, *OSGi Service Platform Service Compendium*, OSGi Alliance, 2005; http://osgi.org.

[176] R.S. Hall and H. Cervantes, “Challenges in building service-oriented applications for OSGi,” *IEEE Communications Magazine*, vol. 42, no. 5, 2004, pp. 144-149.

[177] M. Offermans, *Automatically managing service dependencies in OSGi*, Luminis, 2005.

[178] A. Colyer, et al., “Spring Dynamic Modules for OSGi 1.0.1,” 2007; http://static.springframework.org/osgi/docs/1.0.1/reference/html/.

[179] C. Richardson, *POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks*, Manning Publications Co, 2006.

[180] F. Curbera, et al., "Toward a Programming Model for Service-Oriented Computing," *published in the proceedings of the International Conference on Service-Oriented Computing*, 2005, pp 33-47.

[181] F. Curbera, "Component Contracts in Service-Oriented Architectures," *IEEE Computer*, vol. 40, no. 11, 2007, pp. 74-80.

[182] M. Fowler, "POJO : An acronym for: Plain Old Java Object," 2000; http://www.martinfowler.com/bliki/POJO.html.

[183] D.E. Perry, "System Compositions and Shared Dependencies," *published in the proceedings of the Workshop on System Configuration Management*, 1996, pp 139-153.

[184] J. Magee and J. Kramer, *Concurrency: State Models And Java Programs*, John Wiley & Sons, 2006.

[185] P. Kriens, "Service Blinder," *published in the proceedings of the OSGi World Congress*, 2005.

[186] G. Kiczales, et al., "An Overview of AspectJ," *published in the proceedings of the European Conference on Object-Oriented Programming*, 2001, pp 327-353.

[187] B. Woolf, "The null object pattern," *published in the proceedings of the Pattern Languanges of Programs*, 1996.

[188] A. Bottaro, et al., "Context-Aware Service Composition in a Home Control Gateway," *published in the proceedings of the IEEE International Conference on Pervasive Services*, 2007, pp 223-231.

[189] G. Chen and D. Kotz, "Context-sensitive resource discovery," *published in the proceedings of the IEEE International Conference on Pervasive Computing and Communications*, 2003, pp 243-252.

[190] C. Escoffier and R.S. Hall, "Dynamically Adaptable Applications with iPOJO Service Components " *published in the proceedings of the Software Composition*, 2007.

[191] F. Duclos, et al., "Describing and using non functional aspects in component based applications," *published in the proceedings of the International Conference on Aspect-oriented Software Development*, 2002.

[192] E. Bruneton and M. Riveill, "An architecture for extensible middleware platforms," *Software - Practice and Experience*, vol. 31, no. 13, 2001, pp. 1237-1264.

[193] M. Fowler, "Fluent Interface," 2005; http://martinfowler.com/bliki/FluentInterface.html.

[194] E. Bruneton, et al., "ASM: a code manipulation tool to implement adaptable systems," *published in the proceedings of the Adaptable and extensible component systems*, 2002.

[195] E. Kuleshov, "Using ASM Framework to implement common bytecode transformation patterns," *published in the proceedings of the Aspect Oriented Software Development*, 2007.

[196] OSGi Alliance, *Listeners Considered Harmful: The Whiteboard pattern*, OSGi Alliance, 2004; http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf.

[197] C. Escoffier, et al., "Towards a Home Application Server," *published in the proceedings of the IEEE Consumer Communications and Networking Conference*, 2008, pp 321-325.

[198] S. Microsystem, “Sun Java ME - Foundation Profile,” 2006; http://java.sun.com/products/foundation/.

[199] J. Bourcier, et al., “A Dynamic-SOA Home Control Gateway,” *published in the proceedings of the IEEE International Conference on Services Computing*, 2006, pp 463-470.

[200] D. Donsez, et al., “A Multi-Protocol Service-Oriented Platform for Home Control Applications,” *Proc. IEEE Consumer Communications and Networking Conference*, 2007, pp. 1174-1175.

[201] A. Dearle, “Software Deployment, Past, Present and Future,” *published in the proceedings of the Future of Software Engineering*, 2007, pp 269-284.

[202] A. Carzaniga, et al., *A Characterization Framework for Software Deployment Technologies*, University of Colorado, 1998.

[203] R.S. Hall, *RFC-112 : OSGi Bundle Repository*, OSGi Alliance, 2006.

[204] A. Diaconescu, et al., “Autonomic Management via Dynamic Combinations of Reusable Strategies,” *published in the proceedings of the ACM International Conference on Autonomic Computing and Communication Systems (Autonomics)*, 2008.

[205] P. Lalanda and C. Marin, “A Domain-Configurable Development Environment for Service-Oriented Applications,” *IEEE Software*, vol. 24, no. 6, 2007, pp. 31-38.