HAL
open science

Ligne de produits dynamique pour les applications à services

Jianqi Yu

To cite this version:

Jianqi Yu. Ligne de produits dynamique pour les applications à services. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2010. Français. NNT : . tel-00493355

HAL Id: tel-00493355

https://theses.hal.science/tel-00493355

Submitted on 18 Jun 2010

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

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

UNIVERSITE JOSEPH FOURIER (GRENOBLE I)

THESE

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

DISCIPLINE : INFORMATIQUE

ECOLE DOCTORALE MATHEMATIQUES, SCIENCES ET TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE

PRESENTEE ET SOUTENUE PUBLIQUEMENT PAR

JIANQI YU

LE 16 JUIN 2010

LIGNE DE PRODUITS DYNAMIQUE POUR LES APPLICATIONS A SERVICES

DIRECTEUR DE THESE :

PHILIPPE LALANDA

JURY

Président	Dominique RIEU, Professeur à l'Université Pierre Mendès France, Grenoble
Rapporteurs	Isabelle DEMEURE, Professeur à TELECOM ParisTech, Paris
	Charles CONSEL, Professeur à l'Institut Polytechnique de Bordeaux
Examinateur	Stéphane FRENOT, Maître de conférences à l'INSA de Lyon
Encadrant	Philippe LALANDA, Professeur à l'Université Joseph Fourier, Grenoble

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

Résumé

Le développement d'applications par composition de services dynamiques et hétérogènes, c'est-à-dire implantés suivant des technologies différentes, est le sujet principal de cette thèse. Nous pensons, en effet, que l'approche orientée service apporte des changements considérables dans le domaine du logiciel et peut amener des gains significatifs en termes de réduction des coûts, d'amélioration de la qualité et de compression des temps de mise sur le marché. Les technologies à services ont dors et déjà pénétré de nombreux secteurs d'activité et répondent à certaines des attentes qu'ils suscitaient.

Le développement d'applications par composition de services hétérogènes demeure néanmoins très complexe pour plusieurs raisons. Tout d'abord, les diverses technologies existantes utilisent des mécanismes de déclaration, de recherche et de liaison très différents. Les services, eux-mêmes, sont décrits suivant des structures souvent éloignées. Des développements et des connaissances techniques très pointus sont ainsi nécessaires pour correctement associer des services utilisant des bases technologiques différentes. D'autre part, la gestion du dynamisme est complexe. Le principe de l'approche à service est de permettre la liaison retardée de service et, dans certains cas, le changement des liaisons en fonction de l'évolution du contexte. Cela demande des algorithmes de synchronisation très précis, difficiles à mettre au point et à tester. Nous nous sommes ainsi rendu compte que, dans de nombreux cas, les bénéfices de l'approche à service ne sont pas complètement obtenus, faute d'une gestion appropriée du dynamisme. Enfin, les services sont essentiellement décrits suivant une logique syntaxique. On ne peut donc pas garantir, dans un cas général, la compatibilité de plusieurs services ou, plus simplement, la correction de leur comportement global. Cela est d'autant plus difficile lorsque des services ont des interactions complexes, non limitées à un unique appel pour obtenir une information.

Nous apportons, dans cette thèse, une dimension « domaine » à la composition de service. La définition d'un domaine permet de restreindre les compositions possibles, aussi bien au niveau technologique qu'au niveau sémantique. C'est ainsi que nous avons trouvé une grande complémentarité entre les approches à service et les approches à base de lignes de produits. Les technologies à service apportent de façon naturelle le dynamisme, c'est-à-dire la capacité à créer des liaisons entre services de façon retardée à l'exécution. Les lignes de produits, quant à elles, définissent un cadre de réutilisation anticipée et planifiée. De façon plus précise, cette thèse défend une démarche outillée de composition de services structurée en trois phases, à savoir : la définition d'un domaine sous forme de services et d'architectures de référence à services, la définition d'applications sous forme d'architectures à service, et l'exécution autonomique des applications en fonction de l'architecture applicative du contexte.

Cette thèse est validée au sein d'un projet collaboratif dans le domaine de la santé dans l'habitat.

Mots-Clés : approche à services, Ingénierie de lignes de produits logiciels, application à services, OSGi, iPOJO.

Summary

Application development by composition of dynamic and heterogeneous services, that is to say, implemented according to different technologies, is the main subject of this thesis. We believe, actually, that service-oriented approach brings considerable changes in software computing and can bring significant benefits in terms of cost reduction, quality improvement and compression of time-to-market. The service-based technologies have already penetrated in many industries sections and answered certain expectations they aroused.

Application development by composing heterogeneous services is still very complex for several reasons. Firstly, the various existing technologies employ very different mechanisms for declaration, research and liaison. The services themselves are described following structures often different. Therefore, development and technical knowledge are necessary to correctly combine services using different technological bases. On the other hand, dynamism management is complex. The principle of the service-oriented approach is to allow late service binding and, in some cases, the change of bindings according to context evolution. This requires very precise synchronization algorithms and is difficult to develop and test. We are well aware that in many cases, the benefits of service-oriented approach are not fully achieved, lacking of appropriate dynamism management. Finally, services are essentially described using a logical syntax. Therefore, we cannot guarantee, in a general case, the compatibility of several services or, more simply, the correctness of their global behavior. This is even more difficult when services have complex interactions, not restricted to a single call to obtain information. In this thesis, we bring a domain-specific dimension into service composition. The domain definition allows restricting the possible compositions of services, both at technical and semantic level. Thus we have found great complementarities between software product lines approaches and service-oriented approaches. Dynamism is a natural characteristic of service-based technologies, that is to say, the ability to bind services as late as possible utile runtime. Software product-lines, in turn, define approaches for planned reuse. Specifically, this thesis provides a three-phase approach for development of service composition with tool support, namely: the definition of a domain in the form of services and reference architectures, the definition of application in the form of service-based architectures and the execution of autonomic applications following application architecture.

This thesis is validated in a collaborative project in the home healthcare domain.

Keywords: SOC, dynamic software product lines, application-based services, OSGi, iPOJO.

Remerciements

Je tiens à remercier toutes les personnes m'ayant soutenue et encouragée 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 tous les membres de mon jury et plus particulièrement madame Dominique RIEU pour avoir accepté de présider ce jury. Je tiens également à remercier madame Isabelle Demeure et monsieur Charles Consel pour avoir accepté de rapporter ce travail et leurs commentaires très pertinents. Je remercie également monsieur Stéphane Frénot pour avoir accepté d'examiner ce travail.

Je tiens également à remercier mon directeur de thèse Mr Philippe Lalanda pour tous ses conseils, son soutien, sa confiance, son aide, ses idées, ses encouragements, ses corrections et ses remarques…tout au long de ce travail.

Je remercie également l'ensemble des membres passés et présents de l'équipe ADELE pour leur accueil chaleureux et la très bonne ambiance qui règne dans cette équipe. Je remercie tous ceux avec qui j'ai travaillé et plus particulièrement Johann Bourcier, Pierre Bourret, German Vega, Ada, Jonathan, Walter, Elmehdi, Stéphane, Vincent, Etienne, Pierre-Alain, Jacky, Yoann…et tous ceux que j'ai oubliés, pour les longues discussions à la cafet (concrètement, sur la passerelle) et leur bonne humeur.

Je souhaite remercier également mes amis qui m'ont soutenu tout au long de cette thèse. En particulier, je remercie vivement Aurélien et Nana qui m'ont accompagnée, encouragée et partagé avec moi de bons moments tout au long de ce chemin. Merci à, ada, johann, pierre, jonathan, medhi, houda, marcia, xiaohong et philou…

Pour finir j'aimerais remercier ma famille pour m'avoir soutenue tout au long de ces 3 années. Je remercie du fond de mon cœur Xiangdong qui a accepté et même soutenu ma décision d'entreprendre une thèse en France, malgré les conséquences pour tous les deux. Merci également à mes parents et mes beaux parents ainsi que ma sœur et ma belle sœur pour leur soutien éloigné. Enfin, je souhaite remercier spécialement ma grand-mère (pour moi, l'une des personnes les plus importantes de ma famille), malgré le fait que je l'ai perdue lors de cette dernière année.

Sommaire

Chapitre 1 Introduction 12

1. La réutilisation en logiciel 13

1.1. Définition 13

1.2. Historique 14

1.2.1. Réutilisation ad-hoc 14

1.2.2. Objet 14

1.2.3. Composants 15

1.2.4. Services 16

1.2.5. Modèles 17

1.2.6. Ligne de produits 18

1.2.7. Conclusion 19

2. Objectifs de cette thèse 20

3. Organisation du document 21

Chapitre 2 Approche à services 23

1. Services : définitions et architecture 23

1.1. Services 23

1.2. SOC : Service-Oriented Computing 25

1.3. SOA : Service-Oriented Architecture 26

1.4. Besoins 27

1.5. SOC et le dynamisme 28

1.6. Caractérisation 29

2. Composition de services 30

2.1. Définitions 30

2.2. Composition par procédés 31
2.3. Composition structurelle 32
3. Services Web 33
3.1. Principes 33
3.2. WSDL : le langage de description des services Web 34
3.3. UDDI : l'annuaire de services Web 35
3.4. SOAP : les communications entre services Web 36
3.5. WS-BPEL 37
3.6. Synthèse 38
4. Composants orientés service 39
4.1. Principes 39
4.1.1. Les composants 40
4.1.2. Approche à composants orientés service 41
4.2. OSGi 42
4.3. iPOJO 44
4.4. Synthèse 46
5. Autres technologies à services 47
5.1. CORBA et Jini 47
5.1.1. CORBA 47
5.1.2. Jini 48
5.1.3. Synthèse 49
5.2. UPnP et DPWS 50
5.2.1. UPnP 51
5.2.2. DPWS 52
5.2.3. Synthèse 53

5.3. SCA 54

6. Conclusion 55

Chapitre 3 Lignes de produits 57

1. Définitions 58

2. L'ingénierie des domaines 62

2.1. Définitions 62

2.2. Analyse du domaine métier 63

2.3. Conception de l'architecture de référence 65

2.4. Conception et implantation des composants 67

2.5. Conclusion 68

3. L'ingénierie des applications 69

3.1. L'analyse des exigences d'applications 69

3.2. La conception d'applications et l'implémentation 70

4. Architecture 72

4.1. Définitions 72

4.2. ADLs 74

4.3. Styles d'architecture 75

4.4. Architectures de référence 76

4.4.1. Définition 76

4.4.2. Evaluation et évolution 77

4.4.3. Gestion de la variabilité 78

5. Conclusion 81

Chapitre 4 Proposition 83

1. Rappel du contexte 84

2. Proposition 85

3. La phase d'ingénierie du domaine 87

4. La phase d'ingénierie applicative 88

5. La phase d'exécution 89

6. Synthèse 90

Chapitre 5 Services et architectures 91

1. Introduction 92

2. Spécification de service 93

2.1. Définition 93

2.2. Exemple 95

2.3. Conclusion 97

3. Implantation de service 97

3.1. Définition 97

3.2. Exemple 99

3.3. Conclusion 100

4. Instance de service 100

5. Architecture 101

5.1. Définition 101

5.2. Variabilité au sein d'une architecture 103

5.3. Raffinement des architectures de référence 103

5.3.1. Règle du raffinement d'architectures de référence 104

5.4. Exemple 110

6. Conclusion 113

Chapitre 6 Le prototype-*ChiSpace* 114

1. Introduction 115

2. Architecture globale de *ChiSpace* 116

3. Outil de l'ingénierie du domaine 118
3.1. Vision globale 118
3.2. Définition d'un domaine dans ChiSpace 120
3.3. Définition d'une spécification de service 121
3.4. Définition d'architectures de référence 123
3.5. Définition d'implantations de service 128
4. Outil d'ingénierie applicative 129
4.1. Génération de l'outil 129
4.2. Définition d'architecture applicative 133
5. L'outil de la phase d'exécution - *Machine d'Exécution* 135
5.1. Vision globale 135
5.2. Modèle d'exécution 136
5.3. Plate-forme d'intégration de service 137
5.4. Moteur d'interprétation 140
5.4.1. Créer l'application à l'exécution 140
5.4.2. Gérer l'évolution de l'application à l'exécution 142
6. Synthèse 144
Chapitre 7 Validation 145
1. Contexte 146
2. Objectifs et méthodologies 147
3. Applications 148
3.1. Rappel de l'architecture de référence dédiée aux applications médicales 148
3.2. Application de rappel de prise des médicaments 149
3.2.1. L'architecture applicative 149
3.2.2. Création et gestion de l'évolution de l'application 150

3.2.3. Résultat 152

3.3. Application de rappel des rendez-vous avec les médecins 154

3.3.1. L'architecture applicative 155

3.3.2. Création et gestion de l'évolution de l'application 156

3.3.3. Résultat 157

4. Conclusion 160

Chapitre 8 Conclusion et Perspectives 161

1. Synthèse 162

1.1. Défis 162

1.2. Approche en trois phases outillées : 162

2. Perspectives 164

2.1. Environnement de développement de services 164

2.2. Extension « domaine » de la machine d'exécution 164

2.3. Extension de la machine d'exécution pour la gestion autonomique d'applications à services 165

Chapitre 9 BIBLIOGAPHIE 167

Chapitre 1 Introduction

Au cours des dernières décennies, les développements informatiques ont considérablement évolué. Ils atteignent, dans de nombreuses situations, des complexités importantes. Ceci est dû à divers facteurs tels que la taille des logiciels attendus, la réduction des moyens et des délais de mise en production ou de mise sur le marché, les besoins de plus en plus pointus ou, encore, l'accélération du temps de renouvellement des technologies. De façon plus précise, les développeurs doivent relever aujourd'hui des défis importants de diverses natures, incluant :

- L'augmentation de la complexité des exigences. Par exemple, dans les domaines de l'informatique ambiante, des réseaux sociaux ou des entreprises étendues, les besoins sont très difficiles à gérer. A cela est liée l'augmentation de la complexité des technologies, nécessaire au regard des nouveaux besoins. Les nouvelles technologies, souvent immatures, sont difficiles à maîtriser et sont rarement utilisées correctement.
- L'augmentation des demandes d'évolution. Celles-ci proviennent de l'évolution des besoins des utilisateurs qui apparaissent de plus en plus rapidement aujourd'hui, de l'évolution des technologies et de l'évolution des plates-formes d'exécution. Cela demande de produire des logiciels facilement adaptables, et notamment lors de l'exécution.
- L'augmentation de la diversité. Lors d'un développement, les ingénieurs sont confrontés à une grande diversité de technologies et d'outils. Ils sont aussi amenés à intégrer des équipements très hétérogènes pour, par exemple, obtenir des données ou afficher des résultats.
- L'augmentation des contraintes économiques. Une demande constante, surprenante d'une certaine façon, est la réduction des coûts et des échéances liés au développement logiciel. Dans certains domaines, les délais de mise sur le marché sont si courts qu'ils posent un véritable casse-tête aux organisations de développement.

Une façon de répondre à ces défis est de mettre en place des politiques de réutilisation et d'intégration au sein des organisations informatiques. La réutilisation vise à reprendre des artéfacts logiciels de nature diverse lors de nouveaux développements. L'intégration, quant à elle, vise à fournir les moyens pour interagir effectivement avec des applications ou des équipements existants.

Dans cette introduction, nous définissons la notion de réutilisation logicielle et dressons un bref historique des travaux dans ce domaine. Nous situons également les objectifs de cette thèse et présentons l'organisation de ce manuscrit.

1. La réutilisation en logiciel

1.1. Définition

Les entreprises logicielles cherchent, depuis des années, à améliorer leur productivité et la qualité de leurs produits de façon à faire face à des développements de plus en plus contraints économiquement et complexes techniquement. L'idée de la réutilisation logicielle s'est ainsi imposée il y a longtemps : reprendre des éléments logiciels existants pour produire de nouveaux logiciels doit, en théorie, permettre de réduire significativement le temps et l'effort nécessaires pour développer des applications logicielles.

Krueger [1] fournit une définition large de la réutilisation que nous reprenons ci-dessous :

> *"Software reuse is the process of creating software systems from existing software artifacts rather than building them from scratch. Typically, reuse involves the selection, specialization and integration of artifacts, although different reuse techniques may emphasize or de-emphasize certain of these."*

Cette définition met en lumière la diversité de la réutilisation. Certes, la réutilisation concerne la reprise de code logiciel, structuré ou pas, pour implanter une fonction ou une application. Il peut s'agir, par exemple, d'une bibliothèque logicielle qui est réutilisée dès que possible lors de nouveaux développements. Mais, la réutilisation du logiciel ne se limite pas à la seule réutilisation de code. Elle concerne en effet tous les artefacts définis, et éventuellement implantés, produits lors d'une première réalisation logicielle. Ces artefacts comprennent du code mais aussi des connaissances liées, par exemple, à la conception, à l'architecture, aux conditions et contextes d'exécution, aux cas de tests, etc. La réutilisation peut ainsi être mise en œuvre tout au long du cycle de vie d'un développement logiciel.

Mili fournit une autre définition de la réutilisation en logiciel :

> *"Software reuse is the process whereby an organization defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in its development activities." [2]*

Cette définition souligne le côté systématique des procédures permettant le développement planifié des artéfacts logiciels réutilisables, c'est-à-dire que le développement des artéfacts réutilisables est déterminé et planifié dans un contexte donné. Lors de nouveaux développements, des artéfacts logiciels sont réutilisés de façon systématique pour atteindre des objectifs spécifiques. Du point de vue des procédures de développement d'artéfacts réutilisables, les approches de réutilisation logicielle sont divisées en deux catégories : la réutilisation *ad-hoc* (introduite par la suite) et la réutilisation planifiée. L'ensemble des procédures systématiques présentées dans cette définition se réfère alors à la deuxième catégorie de réutilisation du logiciel.

1.2.Historique

Comme mentionné précédemment, l'idée de la réutilisation en logiciel est ancienne : on peut considérer qu'elle a été introduite pour la première fois lors d'une conférence organisée par l'OTAN, fondatrice des travaux en Génie Logiciel [3]. Au cours des années, de nombreuses approches ont été proposées pour mettre en place la réutilisation.

Nous examinons ici les approches importantes, de la réutilisation *ad-hoc* jusqu'aux approches de lignes de produits. Toutes ces approches cherchent à améliorer la qualité et la productivité du développement logiciel en construisant des applications par assemblage, plus ou moins complet, d'artefacts logiciels déjà existants. Les approches se différencient par le grain et le niveau d'abstraction des artefacts réutilisés et, aussi, par les stratégies d'assemblages mises en œuvre. Celles-ci peuvent aller des plus désordonnées (absence de stratégie) aux plus codifiées (systématique). Nous constatons un net progrès au cours des années : les approches de réutilisation plus récentes sont en général basées sur les avancées des approches précédentes. Enfin, nous pensons qu'il faut employer et intégrer plusieurs d'approches pour atteindre l'objective recherché de la réutilisation du logiciel.

1.2.1. Réutilisation ad-hoc

La première forme de réutilisation, et la plus ancienne, est souvent appelée réutilisation *ad-hoc*. Il s'agit pour un programmeur de réutiliser de façon opportuniste un morceau de code ou un schéma de réutilisation qui n'avait pas été préparé en ce sens. La réutilisation est ici non préparée, entièrement décidée et mise en œuvre par celui qui réutilise, en fonction de ses besoins du moment. Cette forme de réutilisation peut être très avantageuse en ce sens qu'elle ne coûte rien au niveau de la préparation (du processus de mise à disposition au sens large) et qu'elle peut rapporter gros. Un programmeur peut en effet faire avancer de façon significative son projet en récupérant de façon heureuse des artefacts logiciels. Cette approche est également très risquée. La qualité et l'adéquation des artefacts réutilisés ne sont pas garanties. Elle dépend complètement du talent, et de la chance, de celui qui réutilise. C'est la forme de réutilisation la plus utilisée, encore aujourd'hui.

1.2.2. Objet

L'approche orientée objets a été introduite comme un paradigme de conception et de programmation logicielle durant les années 1980. Elle utilise les objets (élément de base) et leurs interactions pour constituer des applications logicielles et des systèmes informatiques. Un objet est un représentant d'un type abstrait, une classe, qui contient des attributs permettant de décrire les propriétés de l'objet et des interfaces permettant d'opérer sur ses attributs et de collaborer avec d'autres objets. L'approche orientée objets repose également sur les notions d'héritage et de polymorphisme. L'héritage est un des premiers mécanismes permettant explicitement la réutilisation et l'adaptation de type par surcharge et redéfinition. Le polymorphisme participe également à l'amélioration de la réutilisation en permettant l'utilisation de mêmes interfaces sur des objets différents.

Malgré ces atouts, la pratique a démontré que les applications construites suivant l'approche orientée objets sont difficilement adaptables et réutilisables. Le principal inconvénient de cette approche est le fait que l'adaptation d'une application doit être réalisée à un niveau architectural trop bas - celui des classes. Il est en effet difficile, au niveau d'un langage de programmation, de déterminer des éléments

réutilisables. Les classes sont en effet très liées à leur environnement immédiat (l'application au sens large), qui change de projet en projet. L'approche orientée objets permet donc la réutilisation dans un cadre très limité. Pour aller plus loin, il manque à cette approche une vision architecturale plus élevée qui permettrait de structurer des applications comme assemblages de classes réutilisables.

1.2.3. Composants

Les approches à composants [4], apparues dans les années 1990, visent explicitement la réutilisation. Les composants ont en effet pour vocation de servir comme éléments de base pour la construction d'applications logicielles. Plus précisément, les approches à composants perçoivent le développement d'applications logicielles comme un assemblage de composants, et gèrent la maintenance et l'évolution d'applications par la personnalisation et le remplacement de composants réutilisables [5]. Elles se situent à un niveau architectural plus élevé que celui proposé par des objets. Typiquement, un composant peut être constitué de plusieurs objets pour la réalisation d'une tâche spécifique.

Un composant est souvent proposé sous la forme d'une unité de déploiement qui encapsule un certain nombre d'éléments parmi lesquels son implémentation. Un composant possède un ensemble d'interfaces. Ces interfaces définissent les fonctionnalités fournies et requises d'un composant sous la forme d'un contrat spécifique et visible de l'extérieur. Des modèles spécifiques à composants, comme EJB[1][41], CCM[2] [50], Fractal[3][48], définissent la structure standard des composants, la manière de réaliser des assemblages et fournissent les mécanismes nécessaires pour la prise en compte de certains aspects non-fonctionnels tels que la distribution, la sécurité ou bien encore les transactions. Les applications basées sur les composants emploient les contrats des composants pour décrire leurs interactions et leurs dépendances fonctionnelles sous un contexte donné avec une vision structurale. Cette représentation explicite de l'architecture élève le niveau d'abstraction par rapport aux assemblages d'objets, puisque la granularité de développement devient plus grande lors du passage de l'objet au composant. En outre, il existe plusieurs grains de la réutilisation dans l'approche à composant, par exemple, une bibliothèque réutilisable fournie par le système d'exploitation, une base de données et interface utilisateur graphique uniforme (GUI[4]), etc.

Cependant, plusieurs défis et limites entravent encore la mise en pratique des approches à composant :

- Le développement d'une bibliothèque de composants réutilisables est souvent coûteux. En effet, la durée et l'effort de développement d'une bibliothèque de composants réutilisables sont de trois à cinq fois plus importants par rapport au développement d'un composant spécifique. De plus, une bibliothèque n'est pas toujours efficace pour la réutilisation.
- Les dépendances de fonctionnalités des composants reposent sur la granularité des interfaces fournies et requises. Ceci cause un couplage très fort entre certains composants d'une application. En conséquence, le fort couplage contractuel de composants d'une application supporte peu de dynamisme et de flexibilité. En d'autres termes, une fois l'assemblage de

[1] EJB: *Enterprise JavaBeans*
[2] CCM: *CORBA Component Model*
[3] Fractal: http://fractal.objectweb.org/
[4] GUI: *Graphical user interface*

composants créé, les changements dans l'architecture sont difficilement gérés au cours de l'exécution de l'application. Il est pourtant important, dans un nombre croissant de systèmes tels que les systèmes embarqués en temps réel, de pouvoir changer dynamiquement des comportements de système en cours d'exécution.

- Les approches à composants ne définissent généralement pas de mécanismes systématiques permettant de concevoir de façon stratégique le développement de composants de sorte qu'ils soient réutilisables lors de développements futurs. De tels mécanismes permettraient de mieux structurer les composants préexistants et de planifier leur assemblage dans une application. De plus, cela permettrait de gérer les changements de composants dans l'architecture d'une application de sorte qu'elle satisfasse des exigences spécifiques. Ce mécanisme réduirait le risque de la réutilisation opportuniste. Il pourrait conduire à la réussite de la mise en pratique d'approches à composants – l'approche de réutilisation.

1.2.4. Services

Les approches orientées services [6] ont été introduites comme un nouveau paradigme pour le développement logiciel durant les années 2000. Ce paradigme utilise la notion de service comme élément de base pour la construction d'applications logicielles. Un service est défini comme une entité logicielle qui peut être utilisée de façon statique ou dynamique pour la réalisation d'une application logicielle. Un consommateur, ou client, sélectionne un service à partir de sa description. Il l'utilise sans avoir connaissance de la technologie sous-jacente nécessaire à son implantation ni de sa plate-forme d'exécution. Inversement, le service ne connaît pas le contexte dans lequel il va être utilisé par un client. Cette indépendance à double sens est une propriété forte des services qui facilite le faible couplage.

Chaque service est constitué de deux parties : sa description et son implémentation. Le fournisseur de service définit la syntaxe de l'interface, la sémantique des opérations et les comportements du service dans la description de service. Il peut également décrire certaines propriétés non fonctionnelles telles que la qualité du service, le coût, la localisation, le nombre d'appels autorisés à ce service et par ce service, etc. Ces propriétés sont généralement décrites en utilisant des langages fondés sur XML et des protocoles standards de l'Internet. Une architecture à service (SOA pour *Service-Oriented Architecture*) regroupe un ensemble de services et des mécanismes d'assemblage permettant le développement d'applications basées sur la réutilisation de services. Lors de la sélection d'un service, un contrat est mis en place, de façon tacite ou explicite, entre consommateur et fournisseur. La description du service peut être considérée comme le contrat de base. Cependant, une négociation peut avoir lieu entre l'utilisateur du service et le fournisseur du service. Les contrats de services permettent de réduire le couplage lié aux dépendances lors de la création d'une application par assemblage de services. Ceci améliore aussi le niveau d'abstraction des applications et élève la granularité de la réutilisation.

Les caractéristiques des applications à services, telles que la substituabilité transparente, le faible couplage, la liaison retardée et la technologie d'implémentation neutre, sont particulièrement intéressantes dans de nouveaux domaines d'applications ayant de fortes contraintes de dynamisme. L'orientation service est ainsi de plus en plus utilisée dans les applications dites pervasives, faisant usage de réseaux de capteurs. Cette approche permet de gérer également l'arrivée et la disparition de capteurs et l'intégration d'environnements et de plates-formes hétérogènes. Toutefois, la composition de services pour former une application est clairement une tâche complexe, source de nombreuses erreurs potentielles pour plusieurs raisons. En premier lieu, il existe de nombreuses technologies pour décrire, éditer et composer des services. Différents protocoles et mécanismes peuvent être utilisés pour mettre en œuvre une architecture

orientée services (SOA), tels que Web services[5], UPnP[6], DPWS[7], OSGi[8]. Deuxièmement, il est très difficile de vérifier la conformité d'une composition de services, incluant les conformités syntaxique et sémantique. Actuellement, les technologies capables de vérifier la conformité d'une composition de services sont seulement émergentes.

D'autre part, les développeurs doivent faire face à plusieurs difficultés pour la mise à disposition de la description de service. Aujourd'hui, de nombreux langages servent à la réalisation de la description de services. Le développeur doit connaître les détails techniques des langages et leurs capacités avant de déterminer lequel utiliser. Dans certaines situations où de nombreux services sont disponibles, la stratégie de recherche, sélection et composition de services est difficile à spécifier pour les développeurs. Enfin, la qualité des compositions de services est une problématique critique. Certaines applications distribuées en temps réel désirent faire collaborer plusieurs services pour réaliser une transaction en temps opportun. Mais, le dynamisme des services soulève le défi de la gestion de la qualité des applications distribuées sous la forme de composition de services. L'approche à service seule est donc encore insuffisante pour une réutilisation effective.

1.2.5. Modèles

Les approches précédentes se concentrent sur la réutilisation de code. Comme nous l'avons dit précédemment, la réutilisation ne se limite pas au code mais concerne tous les artefacts d'un développement logiciel. Les modèles en particulier sont des éléments de réutilisation particulièrement intéressants. L'utilisation de modèles est investiguée de façon systématique depuis quelques années et a donné naissances au MDE (*Model Driven Engineering [7]*) et MDA (*Model Driven Architecture*[9]), le second pouvant être vu comme une déclinaison du premier. Ces approches prônent l'étude et l'utilisation systématiques de modèles. Un de leurs buts majeurs est de fournir des technologies permettant aux développeurs de s'abstraire de complexités grandissantes liées aux langages et plates-formes d'exécution.

Un modèle est une abstraction de certains aspects d'un système [8]. Il est important de comprendre qu'un modèle est une simplification d'une réalité (qui peut ne pas encore exister au moment de la création du modèle) et qu'il est crée dans un but bien précis. Par exemple, il peut s'agir de présenter de façon humainement compréhensible un aspect particulier d'un système pour favoriser une explication ou une discussion ou encore de décrire un aspect de façon à permettre une analyse automatique ou une génération de code. Quoiqu'il en soit, de nombreux modèles sont créés lors d'un développement logiciel, tels que des exigences, des architectures, des spécifications de sécurité, etc. Il existe en fait différents types de modèles. En particulier, on peut différencier les modèles de développement et les modèles à l'exécution [8]. Les modèles de développement sont utilisés pour générer du code exécutable (comme du C ou du Java) ou des fichiers d'intégration ou de déploiement tels que des fichiers de configuration XML ou des ponts (ou bridge) pour faire communiquer des systèmes hétérogènes. Les modèles à l'exécution maintiennent une vue de l'environnement d'exécution d'un programme et sont généralement utilisés pour adapter, à l'exécution, le comportement d'un programme. Le but de ces modèles est de cacher la

[5] Services Web: www.w3.org/2002/ws/
[6] UPnP: *Universal Plug and Play* www.upnp.org
[7] DPWS: *Devices Profile for Web Services* http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01
[8] OSGi: Open Services Gateway initiative www.osgi.org
[9] MDA: *Models-driven Architecture* http://www.omg.org/mda/

complexité des phénomènes se produisant durant l'exécution pour permettre à des agents (des programmes) de faire évoluer le système au mieux (en déployant de nouveaux artefacts par exemple).

Les modèles possèdent une grande valeur intrinsèque et les réutiliser présente une forte valeur ajoutée. Il existe d'ailleurs plusieurs initiatives aujourd'hui pour créer des répertoires de modèles sur le Web pour influencer les nouveaux développements. C'est cette volonté de réutilisation qui a donné naissance aux patterns de conception, ou *design patterns*. Les patterns de conception ont été introduits dans le livre « *Design patterns :Elements of Reusable Object-Oriented Software* » sorti en 1994 [9]. Un pattern est en fait un ensemble de modèles décrivant une solution à un problème donné. Chaque modèle se focalise sur un aspect donné de la solution. Par exemple, un modèle peut décrire la structure de la solution en termes de composants ou d'objets, un autre modèle peut se focaliser sur la dynamique de la solution, etc. De nombreux patterns ont été proposés dans des domaines variés. Par exemple, dans le domaine de l'informatique distribuée, des patterns ont été publiés pour traiter les problèmes de répartition, de communications, de séparation entre technologies logicielles et matérielles. Dans [10], de nombreux patterns reposent sur le découplage entre le code de logique métier et le code technique liés à la distribution. Nous pouvons, par exemple, citer les patterns *Interceptor, Asynchronous Completion Token, Reactor,* ou *Proactor* qui sont particulièrement utiles. Des patterns ont également été proposés pour le développement d'applications Internet ou pour l'intégration d'applications IT [11].

La réutilisation de modèles n'est malheureusement pas si simple. Comme indiqué précédemment, les modèles constituent une abstraction partielle, très focalisée, d'un système et sont ainsi très liés à un contexte. Pour les réutiliser directement, il faut donc se trouver dans les mêmes conditions de développement (qui ne sont souvent pas formalisées).

1.2.6. Ligne de produits

L'ingénierie de lignes de produits logiciels est un paradigme récent favorisant la réutilisation lors du développement d'applications logicielles. Ce paradigme a pour but d'améliorer la réutilisation en se concentrant non plus sur le développement d'un produit unique mais sur la production de familles de produits logiciels. Cette approche est investiguée, sous une forme ou sous une autre, depuis de nombreuses années. Dijkstra a proposé l'idée de familles de produits logiciels en 1972 [12]. David Parnas a mis en pratique la conception de familles de produits logiciels pour développer des systèmes embarqués de manière à mieux gérer les aspects non fonctionnels en 1976 [13]. Jim Neighbors a proposé le thème d'analyse du domaine dans les années 1980. Les processus systématiques de développement de lignes de produits et leurs premières applications sont apparus pendant des années 1980 et 1990 [14].

Nous considérons qu'un groupe d'applications logicielles satisfaisant des besoins similaires est une famille de produits. Dans le reste de cette thèse, nous utiliserons le terme de famille de produits pour désigner un ensemble de produits liés. Ainsi, une approche de lignes de produits logiciels est une méthodologie systématique pour réaliser une famille de produits en se basant sur un ensemble d'artéfacts logiciels réutilisables. Une famille de produits est construite en regroupant les connaissances logicielles réutilisables (soit des concepts abstraits, soit des concepts implémentés) concernant un domaine spécifique. Des ingénieurs peuvent alors les utiliser pour construire et modifier des applications spécifiques dans le domaine concerné. Les artéfacts réutilisables logiciels sont caractérisés par des points communs à tous les produits, ce sont les invariants, et des points variables qui peuvent différer entre produits. Les points communs servent à structurer une famille et les variabilités servent à personnaliser un produit spécifique lors du processus de l'ingénierie des applications.

L'architecture de référence est un point commun majeur de l'approche. Une architecture de référence apparaît généralement comme un modèle spécifiant un ensemble de composants, ou de services, en interaction. Elle contient des règles, des contraintes, des dépendances caractérisant une solution architecturale dans un domaine. Elle inclut des parties fixes et des parties variables où des choix doivent être faits afin de personnaliser les applications spécifiques. Toutes les architectures des applications spécifiques donc doivent être conformes à l'architecture de référence de la ligne de produits. Ceci augmente le niveau d'abstraction de la réutilisation au niveau de l'architecture logicielle. Un apport majeur des lignes de produits est qu'un mécanisme de gestion de la variabilité est introduit. Ce mécanisme permet de constituer les applications en réutilisant des artéfacts logiciels de manière stratégique pour adapter un (des) besoin(s) spécifique(s).

Récemment, les approches de lignes de produits ont été appliquées avec succès dans plusieurs grands projets européens, y compris ARES, PRAISE, ESAPS et CAFE [15]. Des lignes de produits ont ainsi été mises en place, de façon plus ou moins avancée, dans des milieux industriels comme l'automobile ou encore l'électronique médicale. Il est intéressant de noter à ce niveau que les approches de lignes de produits ne se limitent pas aux seuls aspects techniques de développement ; elles considèrent également avec succès les aspects économiques et organisationnels liés à la production logicielle [16].

1.2.7. Conclusion

Les techniques et approches de réutilisation ont considérablement évolué ces dernières années. Cela a permis quelques succès dans un domaine qui se caractérisait jusqu'alors par des résultats mitigés. Les modèles, sous la forme de patterns, sont en effet largement utilisés aujourd'hui. Pareillement, comme en témoignent les retours d'expérience du SEI[10], des lignes de produits sont installées avec succès dans de nombreuses industries. Enfin, les services représentent une intéressante opportunité de réutilisation pour les entreprises. De nombreux projets sont du reste en cours dans ce domaine.

Ces différentes approches ont cependant des limites. Les modèles sont surtout réutilisables quand on reste à un haut niveau d'abstraction. Les lignes de produits se concentrent généralement sur les processus et ne bénéficient pas toujours des techniques nécessaires, notamment pour retarder au maximum les prises de décision de conception. Enfin, la composition de services pour former de nouvelles applications demeure une activité fort complexe.

Nous examinerons ces limites plus en détail dans la suite de ce manuscrit. Nous présentons, également dans cette optique, les objectifs de cette thèse dans la section suivante.

[10] SEI : *Software Engineering Institute* http://www.sei.cmu.edu/productlines/

2. Objectifs de cette thèse

Le développement d'applications par composition de services dynamiques et hétérogènes, c'est-à-dire implantés suivant des technologies différentes, est le sujet principal de cette thèse. Nous pensons, en effet, que l'approche orientée service apporte des changements considérables dans le domaine du logiciel et peut amener des gains significatifs en termes de réduction des coûts, d'amélioration de la qualité et de compression des temps de mise sur le marché. Nos travaux au sein de projets collaboratifs, avec Schneider Electric, Bull ou France Telecom par exemple, nous ont confortés dans cette conviction. Les technologies à services ont dors et déjà pénétré de nombreux secteurs d'activité et répondent aux attentes qu'ils suscitaient. On retrouve par exemple des services au cœur du serveur d'applications Jonas de Bull, à la base des passerelles pervasives de Schneider Electric, et au sein du système d'information de France Telecom.

Le développement d'applications par composition de services dynamiques et hétérogènes demeure néanmoins très complexe pour plusieurs raisons :

- Les diverses technologies existantes utilisent des mécanismes de déclaration, de recherche et de liaison très différents. Les services, eux-mêmes, sont décrits suivant des structures souvent éloignées. Des développements et des connaissances techniques très pointus sont ainsi nécessaires pour correctement associer des services utilisant des bases technologiques différentes. La composition de service demande du code de « médiation » complexe pour assurer des alignements syntaxiques, sémantiques, l'ajout de propriétés non fonctionnelles, etc. Les applications requérant des services hétérogènes sont de plus en plus nombreuses aujourd'hui, notamment dans le domaine de l'intelligence ambiante, ou « pervasive computing ».
- La gestion du dynamisme est complexe. Le principe de l'approche à service est de permettre la liaison retardée de service et, dans certains cas, le changement des liaisons en fonction de l'évolution du contexte. Cela demande des algorithmes de synchronisation très précis, difficiles à mettre au point et à tester. Nous nous sommes d'ailleurs rendu compte que, dans de nombreux cas d'applications, les bénéfices de l'approche à service ne sont pas complètement obtenus, faute d'une gestion appropriée du dynamisme.
- Les services sont essentiellement décrits suivant une logique syntaxique. On ne peut donc pas garantir, dans un cas général, la compatibilité de plusieurs services ou, plus simplement, la correction de leur comportement global. Cela est d'autant plus difficile lorsque des services ont des interactions complexes, non limitées à un unique appel pour obtenir une information (demande de météo par exemple)

A la lumière de ces limites, nous pensons qu'il est intéressant d'apporter une dimension « domaine » à la composition de services. La définition d'un domaine permet de restreindre les compositions possibles, aussi bien au niveau technologique qu'au niveau sémantique. C'est ainsi que nous avons trouvé une grande complémentarité entre les approches à service et les approches à base de lignes de produits. Les technologies à service apportent de façon naturelle le dynamisme, c'est-à-dire la capacité à créer des liaisons entre services de façon retardée à l'exécution. Les lignes de produits, quant à elles, définissent un cadre de réutilisation anticipée et planifiée.

L'objectif de cette thèse est ainsi de concilier ces deux approches qui, toutes deux, visent à la réutilisation de composants logiciels par la montée en abstraction et l'utilisation de modèles. De façon plus précise, notre proposition est de structurer le développement des applications à services en trois phases :

- Une phase d'ingénierie domaine dont le but est de définir des services et des architectures de référence pour la composition de ces services. L'architecture de référence est caractérisée par des parties variables laissant la place à des adaptations lors des phases plus amont ;
- Une phase d'ingénierie applicative dont le but est de définir une architecture à services apportant une solution à un problème précis. Cette architecture à service reprend une architecture de référence et élimine certains points de variabilité. Elle peut conserver certains aspects variables qui seront résolus lors de la phase suivante ;
- Une phase d'exécution qui exécute de façon autonomique une application à services en fonction du contexte à l'exécution et des contraintes architecturales exprimées par l'architecture applicative, définie précédemment. Cette phase repose sur une machine d'exécution à service supportant la découverte, la sélection et la liaison de services hétérogènes, conformément à une architecture.

Comme nous le verrons plus tard, cette proposition revient, d'une certaine façon, à juxtaposer les phases de développement définies par les lignes de produits et la phase d'exécution en environnement dynamique des services.

L'approche proposée est entièrement outillée et validée dans le domaine de la santé. La mise en œuvre a été réalisée en suivant une approche dirigée par les modèles.

3. Organisation du document

Ce document est organisé comme suit. Après cette introduction, le manuscrit est divisé en deux grandes parties : l'état de l'art et la proposition.

La première partie présente l'état de l'art. Les travaux de cette thèse se situent à la convergence de deux domaines, à savoir les applications à services et les lignes de produits logicielles. Pour cette raison, l'état de l'art est divisé en deux chapitres :

- Le chapitre 2 présente les principes de l'approche à services. Nous introduisons les notions de services et d'architecture à services. Nous examinons ensuite certaines des approches à services les plus importantes et les plus utilisées aujourd'hui, à savoir les services Web et les composants orientés service. Nous nous concentrons ensuite sur la composition de services et mettons en lumière les faiblesses des solutions actuelles
- Le chapitre 3 présente les lignes de produits logicielles, une approche de réutilisation planifiée, aujourd'hui particulièrement populaire dans les milieux industriels et académiques. Nous donnons tout d'abord un ensemble de définitions. Nous examinons ensuite deux processus de développement qui forment l'essence de l'approche : l'ingénierie domaine et l'ingénierie applicative. Enfin, nous terminons en présentant la notion d'architecture logicielle, qui représente un élément technique fondamental de l'approche à service.

La deuxième partie introduit notre contribution et est structurée en trois chapitres :

- Le chapitre 4 présente les grands principes de notre proposition qui structure le développement d'applications à services en trois phases. Les phases d'ingénierie domaine, d'ingénierie application et d'exécution que nous proposons sont ainsi présentées de façon globale.
- Le chapitre 5 présente en détails les modèles qui nous avons été amenés à spécifier pour mettre en œuvre ces trois phases. En particulier, nous présentons la modélisation des notions de spécification de service, d'implantation de services et d'architectures à services incluant des points de variabilité.
- Le chapitre 6 présente en détail les outils supportant notre approche. Ces outils, supportant les trois phases que nous avons définies, font usage de technologie issue de l'ingénierie dirigé par les modèles
- Le chapitre 7 traite de la validation de notre approche. Il présente deux applications liées au maintien à domicile des personnes fragiles, et présente certains éléments quantitatifs permettant de juger de l'intérêt de l'approche.

Finalement, le chapitre 8 conclut cette thèse. Il présente une synthèse de cette thèse, et notamment de notre proposition. A l'occasion, nous rappelons certaines réflexions dans le but de mettre en avant les principales contributions, d'identifier les questions ouvertes et les perspectives de ce travail.

Chapitre 2 Approche à services

1. Services : définitions et architecture

L'approche à services est une approche relativement récente qui présente un certain nombre d'avantages pour la réalisation d'applications. Nous allons, dans un premier temps, définir l'élément clé de l'approche à services, c'est-à-dire le service. Ensuite, nous présenterons les différents acteurs et leurs interactions pour cette approche. Dans une troisième partie, nous détaillerons l'architecture à services. Nous expliquerons aussi les besoins qui en découlent. Finalement, nous mettrons en évidence les éléments spécifiques qui nous permettront par la suite de caractériser les différentes technologies qui mettent en œuvre cette approche.

1.1. Services

La notion de service reste floue. Papazoglou a proposé la définition suivante :

> *"Services are self-describing, platform agnostic computational elements." [6]*

Un service est vu comme une entité logicielle qui peut être utilisée grâce à sa description. Le consommateur du service l'utilise sans avoir connaissance de la technologie sous-jacente pour son implantation ainsi que de sa plate-forme d'exécution. De plus, le service ne connaît pas le contexte dans lequel il va être utilisé par le client. Cette indépendance à double sens est une propriété forte des services qui facilite le faible couplage. Cette définition contient une faiblesse : elle sous-entend qu'un service est exécuté sur une plate-forme distante et qu'il ne peut pas être importé sur une plate-forme locale.

Le consortium OASIS[11], qui est en charge de la normalisation et de la standardisation des applications Internet, a aussi donné une définition du terme service :

> *"A service is a mechanism to enable access to one or more capabilities, where the access is provided by using a prescribed interface and is exercised consistent with constraints and policies as specified by the service composition. A service is accessed by means of a service interface where the interface comprises the specifics of how to access the underlying capabilities. There are no constraints on what constitutes the underlying capability or how access is implemented by the service provider. A service is opaque in that its implementation is typically hidden from the service consumer except for (1) the information and behavior models exposed through the service*

[11]OASIS: Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org/home/index.php.

interface and (2) the information required by service consumers to determine whether a given service is appropriate for their needs." [17]

Pour le consortium OASIS, un service fournit un ensemble de fonctionnalités décrites dans une spécification, appelée interface, ainsi qu'un ensemble de contraintes et de politiques d'accès aux fonctionnalités offertes. L'implantation du service n'est pas visible à l'utilisateur. Seules les informations qui peuvent permettre de savoir si le service correspond aux besoins de l'utilisateur sont disponibles. Nous pouvons noter qu'il est quand même difficile de discerner une information utile d'une information inutile pour le choix d'un service.

Arsanjani définit lui aussi le terme de service. A la différence des autres définitions, il met en avant les principales interactions qui permettent l'utilisation des fonctionnalités du service :

"A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. Services should be ideally be governed by declarative policies and thus support a dynamically reconfigurable architectural style." [18]

Cette définition s'intéresse à l'utilité de la description du service. Dans un premier temps, le client recherche le service correspondant à ses besoins en fonction de la description de service fournie. Ensuite, il fait la liaison avec lui et il l'invoque. Ces actions peuvent être réalisées avant ou pendant l'exécution de l'application à services.

Suite à ces définitions, nous pouvons dire qu'un service est :

« Une entité logicielle qui fournit un ensemble de fonctionnalités définies dans une description de service. Cette description comporte des informations sur la partie fonctionnelle du service mais aussi sur ses aspects non-fonctionnels. A partir de cette spécification, un consommateur de service peut rechercher un service qui correspond à ses besoins, le sélectionner et l'invoquer en respectant le contrat qui a été accepté par les deux parties. »

Nous avons ajouté la notion de contrat dans cette définition. Un contrat entre deux parties permet de s'assurer que chacun respectera ce qu'il a annoncé de faire. Un contrat est le résultat d'une négociation entre le fournisseur et le consommateur. Ce contrat représente un accord de niveau de service, en anglais *Service Level Agreement* (SLA) [19], qui définit les engagements que prend le fournisseur quant à la qualité de son service, et les pénalités encourues en cas de manquement. Cette qualité doit être mesurable et mesurée selon des critères objectifs acceptés par les deux parties. Un exemple d'accord de service peut être le temps de rétablissement d'un service en cas d'incident, le fournisseur et le consommateur définissent un délai pour le rétablissement du service. Si le délai est dépassé, le fournisseur doit indemniser le consommateur selon les termes du contrat. La définition de l'accord de niveau de service peut se faire selon plusieurs niveaux, comme défini dans [20] :

- le **niveau syntaxique** : les différents acteurs se mettent d'accord sur la signature des méthodes proposées par le service, ce qui correspond au nom des méthodes, aux types de paramètres d'entrée et de sorties ainsi qu'aux types d'exceptions qui peuvent être levées. Ces éléments font partie, en général, de l'interface programmatique du service.

- le **niveau comportemental** : c'est une extension du niveau précédent, qui prend également en considération les pré-conditions, les post-conditions et les invariants.
- le **niveau synchronisation** : ce niveau gère le comportement global pour l'enchaînement des appels de méthodes sous forme de synchronisation. Le contrat décrit les dépendances entre les services. Les appels peuvent se faire de manière séquentielle, parallèle ou sans contrainte.
- le **niveau qualité de service** : ce niveau s'appuie sur tous les précédents. Il ajoute des contraintes de qualité aux services et à leurs interactions : des facteurs de qualité qui respectent des critères qui peuvent être mesurés. Ces quatre niveaux d'accord de service permettent donc aux consommateurs et aux fournisseurs de s'entendre sur la qualité du service attendue, tout comme sur ses fonctionnalités présentées dans une interface.

1.2.SOC : Service-Oriented Computing

Les services sont l'élément clé de l'approche orientée service, en anglais *Service-Oriented Computing* (SOC). Son but est de permettre la construction d'applications à partir d'entités logicielles particulières, que sont les services, tout en assurant un faible couplage. Cette approche n'est pas une technologie ; elle peut être vue comme un style architectural [21].

Ce style architectural repose sur un patron qui définit un ensemble d'interactions entre différents acteurs. Ce patron est présenté dans la figure 2.1. Le modèle d'interactions et les acteurs découlent de la définition des services. Les acteurs sont au nombre de trois :

- le **fournisseur de service** qui propose un service décrit dans une spécification ;
- le **consommateur de service** qui utilise des services des fournisseurs ;
- le **registre de services ou annuaire** qui stocke l'ensemble des descriptions de services déclarées par le fournisseur de service. Il permet aussi aux consommateurs de service de rechercher et de sélectionner le service qui leur sera utile.

Aux trois acteurs de l'architecture orientée service, il a été ajouté trois primitives de communication :

- la **publication de services** : les fournisseurs de service enregistrent leur service dans l'annuaire ;
- la **découverte d'un service** : les consommateurs de service interrogent l'annuaire pour trouver un service qui correspond à leurs besoins ;
- la **liaison et l'invocation d'un service** : un fois le service choisi, le consommateur de service peut se lier au fournisseur et utiliser le service.

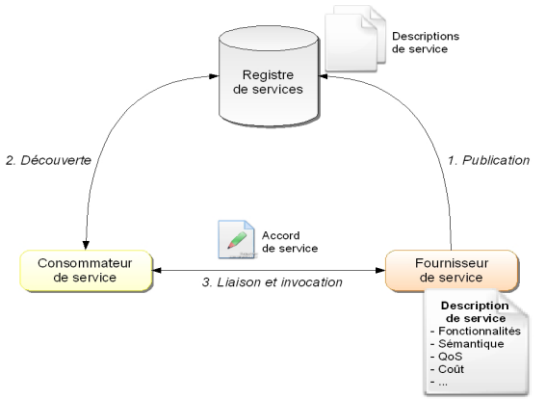


Figure 2.1. Acteurs et interactions dans l'architecture à services

L'avantage certain de cette architecture est que seule la description de service est partagée entre les différents acteurs, ceci permet d'obtenir un très faible couplage. La description de service peut prendre différentes formes et fournir différents degrés de précision selon les approches, mais son but principal est de spécifier les fonctionnalités offertes par le service. Comme cette architecture procure un faible couplage, il apparaît un autre avantage : l'hétérogénéité des implantations et des plates-formes est masquée au consommateur de service, tout comme la localisation du service.

En conséquence du faible couplage obtenu grâce à cette architecture, nous obtenons une nouvelle propriété : la substituabilité. En effet, il est possible de remplacer un service par un autre de façon transparente grâce à l'interface du service dès lors qu'il respecte le contrat que le fournisseur et le client ont passé.

Finalement, cette architecture favorise la communication entre un client et un fournisseur de services appartenant à des domaines d'administration différents. Ceci est un des éléments importants de l'approche à services, même si dans la plupart des cas, les services sont utilisés au sein d'une même entreprise.

L'intérêt grandissant pour les services Web a conduit à une confusion entre les termes de services et architecture à services avec la notion de services Web. Les services Web ne sont qu'une implantation particulière des principes de l'approche à services. Il existe de nombreuses autres implantations telles que Jini [22], UPnP (*Universal Plug and Play*) [23] utilisé dans le contexte des services répartis, OSGi [12][24] pour des services localisés sur une même machine virtuelle. Toutes ces technologies seront détaillées dans les sections 4 et 5 de ce chapitre.

1.3.SOA : Service-Oriented Architecture

Pour réaliser le style architectural présenté précédemment, il faut mettre en place un environnement d'intégration et d'exécution des services. Cet environnement doit être capable de gérer les interactions entre les différents acteurs. Nous pouvons diviser en deux catégories les éléments d'un tel environnement, illustrés par la Figure 2.2 :

[12] OSGi: Open Services Gateway Initiative

- les **mécanismes de base** qui assurent la publication, la découverte, la composition, la négociation, la contractualisation et l'invocation des services ;
- les **mécanismes additionnels** qui prennent en charge les besoins non-fonctionnels tels que la sécurité, les transactions ou encore la qualité de service.

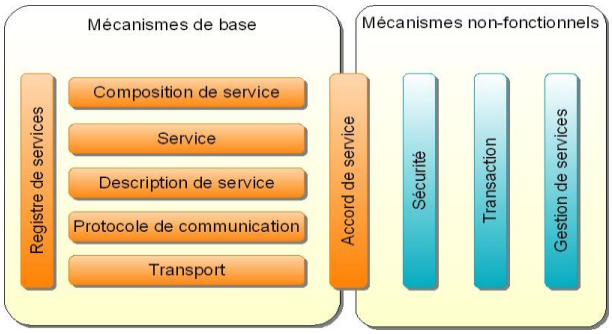


Figure 2.2. Mécanismes nécessaires pour un environnement d'intégration de services

Les mécanismes de Transport et le Protocole de communication sont la base d'un environnement d'intégration de services. Ils permettent d'assurer les communications, c'est-à-dire les requêtes et les réponses, entre les différents acteurs. Pour réaliser les interactions de base, il faut aussi pouvoir décrire le service dans un langage de description spécifique à l'environnement d'intégration. Cette description doit comprendre les fonctionnalités du service, ses éléments non-fonctionnels ainsi que la manière dont il doit être invoqué. De manière transverse, il est nécessaire d'ajouter un *Registre de services* qui permet pour le fournisseur d'enregistrer son service, pour le client de rechercher un service qui répond à ses besoins.

L'*accord de service* est un élément qui fait partie des mécanismes de base mais aussi de la partie non-fonctionnelle. Il représente le contrat qui existe entre le fournisseur de service et le client. Dans ce contrat, sont définies les fonctionnalités que le service doit rendre. Mais, il y est aussi spécifié les propriétés non-fonctionnelles, comme le temps de réponse ou la fiabilité, que le service s'engage à respecter.

Les autres éléments non-fonctionnels nécessaires à un environnement d'intégration de services sont :

- la *Sécurité* qui permet par exemple au fournisseur de service de gérer l'accès à son service.
- la *Transaction* qui est utile dans le cas où divers services sont utilisés en collaboration. Dans ce cas, les transactions permettent de s'assurer de la cohérence des données.
- la *Gestion de services* qui assure l'administration des ressources et de leur utilisation au sein de la plate-forme pour un bon fonctionnement des applications.

1.4.Besoins

Nous avons présenté jusqu'ici l'approche à services comme un nouveau paradigme qui a pour but de faciliter la réalisation d'applications à partir des services. Cependant, dans la section précédente, nous n'avons pas introduit la notion d'application à services ; nous avons seulement défini les mécanismes nécessaires pour construire un environnement d'intégration de services. Ces mécanismes sont nécessaires

à la réalisation d'application à services mais ils ne sont pas suffisants pour répondre à tous les besoins d'une application. Papazoglou [6] a donc proposé une architecture orientée service étendue pour exprimer la composition de services et l'administration d'application, tout en s'appuyant sur les mécanismes de base de l'approche à services. La Figure 3 est la pyramide définie par Papazoglou sur les fonctionnalités attendues de l'architecture à services.

La base de la pyramide contient les fonctionnalités que requiert un intergiciel proposant une approche à services. Ces fonctionnalités sont celles présentées précédemment : la publication, la découverte, la liaison, la sélection et la spécification.

Le deuxième étage de la pyramide introduit les mécanismes de composition de services. Ces mécanismes permettent de coordonner les services, de gérer les compositions et de s'assurer de la conformité de la composition. Le service composite créé doit respecter les principes de base de l'approche à services. La composition de services sera présentée dans la section 2 de ce chapitre.

Le sommet de la pyramide concerne l'administration des services et leurs compositions, c'est-à-dire les applications à services. Ceci correspond à la gestion et à la surveillance d'applications.

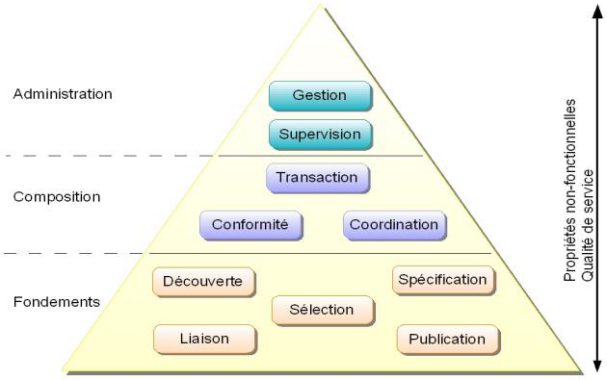


Figure 2.3. Fonctionnalités d'une architecture orientée service étendue

De manière transversale à ces niveaux de la pyramide, une architecture orientée service étendue doit prendre en considération la gestion des propriétés non-fonctionnelles, comme la sécurité ou encore la qualité de service.

Cependant, cette pyramide de l'architecture orientée service est difficile à mettre en œuvre. La plupart des intergiciels ne proposent pas l'ensemble des fonctionnalités présentées. Ils ne fournissent, en général, que les principes de base. Mais, il est très important de mettre en place la composition et l'administration ; car sans ces niveaux, l'utilisateur n'a pas de vision complète de l'application.

1.5. SOC et le dynamisme

Nous avons présenté la vision communément admise de l'approche à services dans les sections précédentes en présentant les acteurs et les interactions. Cependant, certaines définitions introduisent des interactions supplémentaires pour ce que l'on appelle l'approche à services dynamique. Cette approche

s'intéresse aux modifications de l'environnement d'exécution des services. Il a donc été ajouté deux primitives à l'approche à services qui sont les suivantes :

- le **retrait de service** qui signale qu'un fournisseur de service n'est plus en mesure de proposer son service ;
- la **notification** qui informe les consommateurs de l'arrivée ou du départ d'un fournisseur qui propose un service répondant à leurs besoins.

Ces deux nouvelles primitives, présentées dans la Figure 2.4, si elles sont prises en compte par l'architecture à services, permettent au consommateur de choisir à l'exécution son fournisseur de service, voire d'en changer. La prise en compte de l'évolution dynamique de l'environnement est un nouvel avantage que l'on ajoute à ceux de l'approche à services.

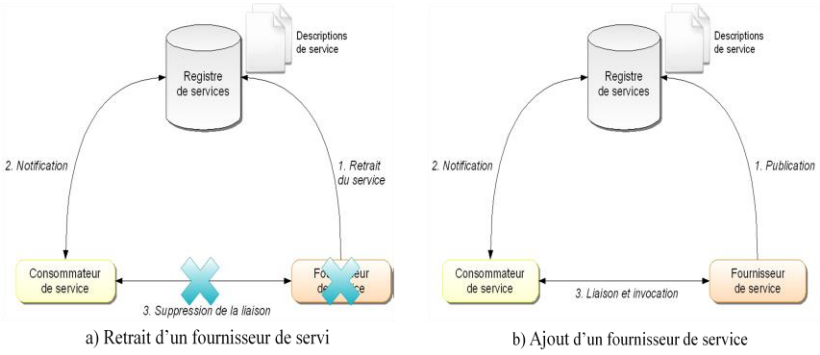


Figure 2.4. Interactions de l'approche à services dynamique

La pyramide de l'architecture orientée service étendue peut, elle-même, être étendue au dynamisme. Il faut ajouter à la base les deux nouvelles primitives : retrait et notification. Ensuite, l'ajout de fonctionnalités dans les niveaux supérieurs s'impose pour prendre en compte le dynamisme de la couche basse. La composition de services dans l'architecture dynamique doit permettre de s'assurer à tout moment de la conformité de cette composition. Et, la partie administration doit être capable de gérer la supervision et la gestion de l'application en fonction des arrivées et des départs des services.

1.6. Caractérisation

Dans les sections précédentes, nous avons défini ce que sont les services. Nous avons montré quelle architecture est proposée pour mettre en application les concepts de l'approche à services et quels sont les besoins qui découlent de cette approche. Par la suite, nous allons étudier plusieurs technologies dites à services que nous allons caractériser selon les critères suivants :

- **la spécification :** comment sont décrites les fonctionnalités et les propriétés non-fonctionnelles des services ?
- **l'implantation :** dans quel langage de programmation sont réalisés les services ?
- **la découverte :** quel est le registre de services ? quel type de découverte est possible ?
- **la composition :** est-ce que la composition est possible ? Si oui, comment ?
- **la communication :** quel est le protocole de communication entre les différents acteurs ?

- **la liaison :** quelle est la politique de liaison pour faire face au dynamisme ?

Dans les parties suivantes, nous allons répondre à ces questions pour chacune des technologies à services étudiées, comme par exemple les services Web un standard très répandu, OSGi et iPOJO des technologies à composants orientés service, UPnP et DPWS des technologies à services pour la domotique.

2. Composition de services

2.1. Définitions

Comme nous l'avons vu précédemment, la mise en œuvre de l'approche à services ouvre des perspectives pour la composition de services dans le but de construire des applications. La composition de services peut être vue comme un mécanisme qui permet l'intégration des services pour réaliser une application. Le résultat d'une composition peut être un nouveau service, appelé service composite. Ce type de composition est dite récursif ou hiérarchique. La Figure 2.5 présente le principe de la composition de services ; à partir d'un ensemble de services disponibles dans un registre, nous pouvons construire une application à services.

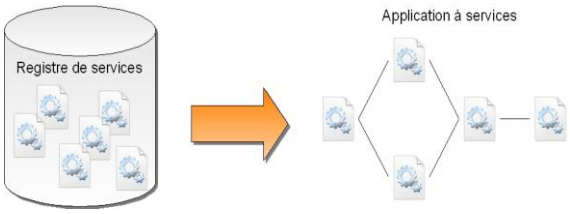


Figure 2.5. Composition de services

Cependant, pour passer d'un ensemble de services à une composition de services correctement structurée, il faut suivre un certain nombre d'étapes de la spécification à la composition concrète qui peut être directement exécutée :

1. la définition de l'architecture fonctionnelle : cette phase est faite pour identifier les fonctionnalités attendues pour l'application résultant de la composition de services. Des travaux d'aide au niveau de cette phase commencent à exister [25].

2. l'identification des services : selon les fonctionnalités attendues, on détermine les services nécessaires à la composition ;

3. la sélection des services et leur implantation : à partir des services identifiés à l'étape précédente, il faut sélectionner les services qui répondent correctement aux besoins ainsi que les implantations adaptées ;

4. la médiation entre services : même si à l'étape précédente, les services les plus adaptés ont été sélectionnés, en général, il n'est pas possible de les assembler tel quel. Il faut souvent ajouter de la

médiation, par exemple sémantique, pour que les interactions entre services fonctionnent comme escomptés.

5. le déploiement et l'invocation des services : une fois la composition correctement réalisée, il faut déployer les services sur les plates-formes d'exécution. Il est ainsi possible d'invoquer les services pour obtenir la composition concrète.

Malgré cette décomposition en tâches pour réaliser une application à base de composants, ce processus reste une activité difficile qui peut même être longue. Chaque tâche peut être divisée en sous-tâches que les développeurs doivent parfois exécuter manuellement. La situation idéale qui consiste à avoir uniquement les « bons » services, c'est-à-dire des services avec les fonctionnalités attendues et tous compatibles, ne se présente que rarement aux développeurs. Ces derniers doivent gérer les problèmes d'incompatibilités par une couche de médiation qui permet d'adapter les services les uns aux autres. Les problèmes d'incompatibilités sont, par exemple, des problèmes de types de données pour les entrées et les sorties des services.

La complexité de la composition est double : d'une part, la complexité du logique métier inhérent aux applications qui nécessite une expertise métier ; d'autre part, la complexité de la mise en œuvre de l'approche à services.

La composition de services est spécifiée selon une logique de coordination des services, c'est-à-dire selon le contrôle de la composition qui peut être extrinsèque ou intrinsèque aux services. Ces deux possibilités de gestion du contrôle définissent deux styles de compositions : le composition par procédés, aussi appelée composition comportementale, et la composition structurelle qui sont présentées dans les deux sections suivantes.

2.2. Composition par procédés

Dans ce type de composition, la logique de coordination des services est spécifiée par un procédé. Un procédé est représenté par un graphe orienté d'activités et un flot de contrôle qui donne l'ordre d'exécution des activités, comme illustré dans la Figure 2.6. Chaque activité représente une fonctionnalité et cette dernière est réalisée concrètement par un service.

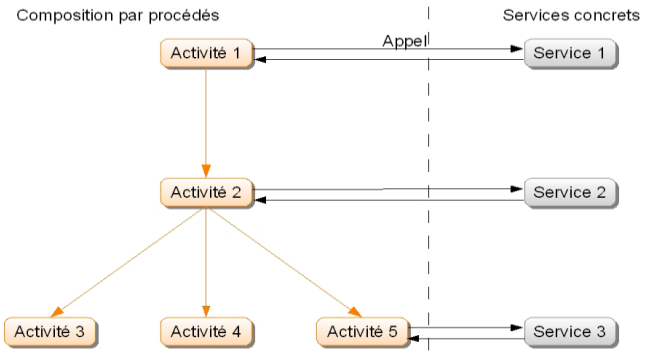


Figure 2.6. Composition par procédés

En pratique, la composition est décrite dans un langage spécifique qui est interprété par un moteur d'exécution. Toutes les communications avec les services sont gérées par le moteur, tout comme les erreurs.

Nous distinguons deux sous-types de composition par procédés :

- **l'orchestration de services** qui décrit, du point de vue d'un service, les interactions de celui-ci ainsi que les étapes internes (ex. transformations de données, invocations à des modules internes) entre ses interactions [26]. C'est une vision centralisée sur les services.
- **la chorégraphie de services** qui décrit la collaboration entre une collection de services dont le but est d'atteindre un objectif donné. L'accomplissement de ce but commun se fait alors par des échanges ordonnés de messages [27]. Dans ce cas, nous avons une vision globale des services et de leurs interactions.

Ces deux points de vue de la composition par procédés ne sont utilisés actuellement que pour la technologie des services Web. WS-BPEL[13] [28], qui sera présenté par la suite dans la section 3.5, est un exemple de langage d'orchestration de services Web ; WS-CDL[14] [29] est, quant à lui, un langage de description de chorégraphie de services Web.

La composition par procédés permet de séparer distinctement le contrôle d'une application des fonctionnalités. Elle est utilisée par les développeurs pour construire des applications à base de briques logicielles dont le fonctionnement interne n'est pas connu. Ces briques logicielles peuvent être vues comme des « boîtes noires ». Les fonctionnalités ainsi identifiées peuvent être plus facilement réutilisées pour d'autres compositions et l'expression de la logique de contrôle est exprimée simplement. Cependant, il existe peu d'interactions entre les activités qui sont assemblées, puisque les langages de composition ne permettent pas de réaliser des algorithmes complexes. De plus, il n'est pas possible de détailler, par exemple, les types d'interactions qui existent entre les activités. La composition par procédés propose un mode de contrôle très avantageux mais qui reste limité à certains domaines.

2.3. Composition structurelle

Par opposition à la composition par procédés, le contrôle dans une composition structurelle est exprimé à l'intérieur des services. Le contrôle n'est alors connu que du développeur et les seules informations qu'il possède sont celles concernant les fonctionnalités que le service fournit et celles que le service requiert.

Dans le cas de la composition structurelle, les services sont donc clairement identifiés avec leurs interactions. Il faut à l'assemblage résoudre les dépendances syntaxiques et sémantiques entre les composants pour s'assurer de la validité de la composition. C'est pourquoi pour définir le fonctionnement de la composition, le développeur livre aussi la logique de coordination, qui peut être, par exemple, sous forme de classe Java. Aujourd'hui, la spécification SCA[15] [30], présentée dans la section 5.3, est l'une des rares propositions pour la spécification de compositions structurelles avec [31].

---

[13] WS-BPEL: *Web Service Business Process Execution Language*
[14] WS-CDL: *Web Service Choregraphy Description Language*
[15] SCA : *Service Component Architecture*

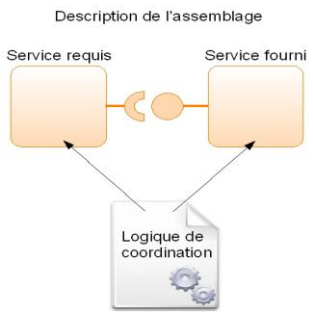


Figure 2.7. Composition structurelle

A l'inverse de la composition par procédés, la composition structurelle ne permet pas facilement la réutilisation des composants puisque le contrôle est interne aux services. Par contre, elle est plus efficace puisque la communication entre services est directe, elle ne passe pas par un intermédiaire comme dans la composition par procédés. De plus, le contenu du service est réalisé par le développeur, les algorithmes et les interactions entre services peuvent être plus complexes que ceux des compositions par procédés, qui ont un langage plus restreint.

3. Services Web

Les services Web sont certainement la technologie la plus connue et la plus populaire dans le monde industriel et académique pour la mise en place de d'architectures à services. Dans une première partie, nous allons présenter les principes des services Web. Ensuite, nous détaillerons les trois standards utilisés par cette technologie, que sont WSDL, UDDI et SOAP. Pour terminer, nous traiterons de la composition de services Web[16] avec le standard WS-BPEL.

3.1.Principes

Les services Web ont été créés pour rendre disponibles des applications sur l'Internet ou dans un Intranet. Ces services respectent les principes de l'approche orientée service précédemment présentés ; ils sont donc décrits, publiés et découverts. Un fournisseur de service Web enregistre son service, en décrivant ses fonctionnalités et certains de ses aspects non-fonctionnels dans un fichier WSDL, auprès d'un annuaire UDDI. Un client interroge un annuaire UDDI pour trouver un service qui répond à ses besoins. Pour le consommateur, le service Web est une boîte noire qui ne donne pas de détails techniques sur son implantation, seulement des informations sur ses fonctionnalités et quelques propriétés, sa localisation et les moyens pour l'interroger. Les communications se font par le protocole SOAP. L'architecture des services Web est illustrée dans la Figure 2.8.

[16] Nous étudions ici les services Web standards et non les services Web REST

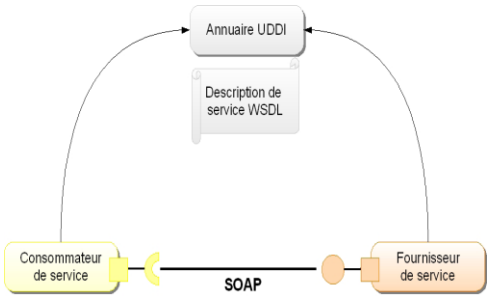


Figure 2.8. Architecture pour les services Web

Cette architecture cache à l'utilisateur toute la complexité de l'implantation du service Web. En effet, un service Web peut faire appel à d'autres services, Web ou non, pour réaliser les fonctionnalités qu'il propose sans que le client en soit conscient. Cela nécessite une coordination entre les différents appels de services. Le détail de chacune des technologies présentées dans cette architecture est donné dans les trois sections suivantes.

3.2. WSDL : le langage de description des services Web

Le succès des services Web repose en partie sur le faible couplage qu'il existe entre les consommateurs de services et le fournisseur de service. La description dans un langage standard des différentes fonctionnalités du service par le fournisseur de service permet aux consommateurs de s'abstraire des langages de programmation utilisés pour réaliser les services Web. Seules les fonctionnalités du service sont présentées dans le fichier de description, ainsi les détails techniques propres au fournisseur de service ne sont pas dévoilés aux consommateurs.

La description d'un service Web est faite dans le langage WSDL[17] [32]. Un fichier WSDL comprend une description des fonctionnalités d'un service, mais il ne se préoccupe pas de l'implantation de celles-ci. Il contient aussi des informations concernant la localisation du service, ainsi que les données et les protocoles à utiliser pour l'appeler. En pratique, le fichier WSDL est un fichier XML qui se divise en deux parties :

- la **définition abstraite** de l'interface du service avec les opérations supportées par le service Web, ainsi que leurs paramètres et les types des données ;
- la **définition concrète** de l'accès au service avec la localisation, par une adresse réseau du fournisseur de service, et les protocoles spécifiques d'accès.

[17] WSDL: Web Services Description Language

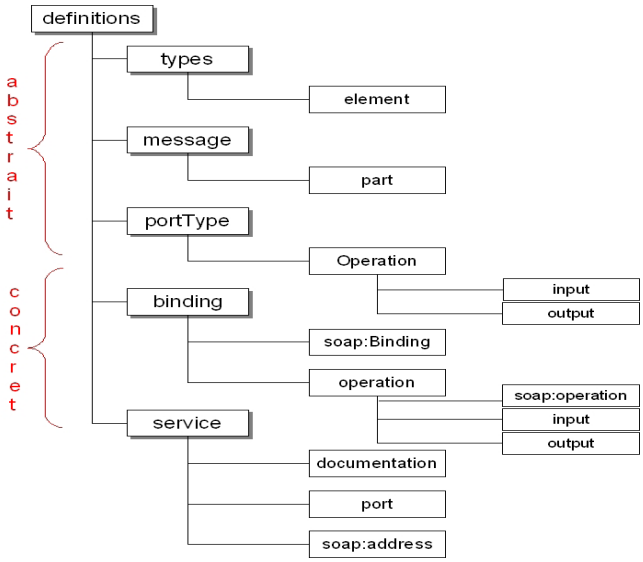


Figure 2.9. La structure d'un fichier WSDL

La Figure 2.9 correspond à la structure d'un fichier WSDL. La partie abstraite est définie par les balises types, messages et port types. Ainsi sont décrits les types de données utilisés, les messages échangés et les opérations possibles pour un service Web. La partie concrète, qui est spécifique à l'implantation, comprend les balises binding et service. L'élément binding spécifie le protocole avec lequel les clients peuvent invoquer le service. L'élément service permet d'associer au service Web une adresse réseau.

Le fichier WSDL détaille les fonctionnalités d'un service Web. Mais pour l'instant, il ne tient pas compte des propriétés non-fonctionnelles, telles que la sécurité, les transactions par exemple, qui peuvent lui être associé. De nouvelles extensions pour les fichiers WSDL sont proposées pour traiter ces aspects non-fonctionnels comme les spécifications WS-Security [33] pour la sécurité, WS-Transaction [34] pour les transactions. Cependant, ces travaux de spécifications avancent lentement et évoluent régulièrement.

La description détaillée de l'interface d'un service Web avec un fichier WSDL n'est pas suffisante pour son utilisation par un client. Avant d'utiliser un service Web, le client doit d'abord le rechercher et s'assurer qu'il corresponde à ses besoins. L'architecture des services Web propose l'utilisation d'un annuaire de services UDDI[18] présenté dans la section suivante.

3.3. UDDI : l'annuaire de services Web

Un des principaux avantages de l'adoption des services Web par les entreprises est le partage de services sur Internet ou dans un Intranet. Le partage de services Web permet le développement plus rapide d'applications, soit un gain temps pour l'entreprise ainsi qu'un gain d'argent. L'élément clé pour ce partage des services est l'annuaire UDDI [35], qui permet de référencer et de classifier leurs fonctionnalités.

[18] UUDI : Universal Description Discovery and Integration

UDDI est une spécification d'annuaire qui propose d'enregistrer et de rechercher des fichiers de description de services Web correspondant aux attentes d'un client. UDDI a été initialement conçu par et pour des industriels en ayant pour but d'avoir un standard, indépendant des plates-formes d'implantation, afin de :

- **connaître** les entreprises qui fournissent des services Web ;
- **découvrir** les services Web disponibles qui répondent aux attentes du client. Pour simplifier les recherches de services, le standard UDDI propose trois types d'annuaires qui ont des critères particuliers :
- les **pages blanches** permettent de connaître les informations concernant les entreprises ;
- les **pages jaunes** présentent les services selon leurs fonctionnalités en utilisant une taxonomie industrielle standard ;
- les **pages vertes**, quant à elles, informent sur les services fournis par les entreprises. Elles référencent la localisation des fichiers de descriptions WSDL des services Web.

Les annuaires UDDI ne sont pratiquement pas utilisés aujourd'hui. Il apparaît que les entreprises mettent en place des architectures à base de services Web en développant leur propre annuaire, par exemple la poste Allemande : *Deutsche Post*. Cette solution, coûteuse mais compréhensible, s'effacera quand des technologies d'annuaire plus complètes et plus robustes s'affirmeront.

WSDL et UDDI sont deux standards pour les services Web. Le premier permet d'afficher les fonctionnalités et les moyens d'utilisation du service d'un fournisseur. Ce dernier peut aussi enregistrer son service dans un annuaire UDDI pour qu'il soit référencé et utilisé par des consommateurs de service. Il existe un autre point important dans l'architecture des services Web ; ce sont les communications entre les différentes parties qui se font avec le protocole SOAP, présenté dans la section suivante.

3.4. SOAP : les communications entre services Web

SOAP, à l'origine acronyme de *Simple Object Access Protocol*, est un protocole dont la syntaxe est basée sur XML. Son but est de permettre des échanges standardisés de données dans des environnements distribués. Il permet d'accéder à des services distants indépendamment de la plate-forme d'exécution. Initialement proposé par Microsoft et IBM, la spécification SOAP est aujourd'hui une recommandation W3C. Il faut noter que depuis la version 1.2 SOAP n'est plus un acronyme, la notion d'objet étant devenue obsolète.

Un message SOAP contient une enveloppe obligatoire, un en-tête facultatif et un corps obligatoire qui se présente sous la forme d'un document XML :

- l'élément **en-tête** contient des éléments fils, appelés *entrées*, permettant d'ajouter des extensions à un message. Ces extensions permettent, par exemple, de prendre en charge les transactions et la sécurité. SOAP se voulant un protocole léger et simple a volontairement ignoré la sécurité.
- l'élément **corps** du message contient la méthode à invoquer ainsi qu'éventuellement ses paramètres d'appel.

Le protocole SOAP s'appuie sur des standards de communication comme le protocole HTTP[19], mais il peut aussi utiliser des protocoles autres comme SMTP[20]. L'avantage d'utiliser SOAP avec le protocole HTTP est que la communication est facilitée, en particulier les *proxys* et les pare-feu peuvent être franchis sans problème. Il est ainsi facilement adaptable à toutes les technologies antérieures, tout en restant simple et extensible. Le protocole SOAP a pour avantage d'être indépendant de la plate-forme d'exécution et du langage de programmation.

```
<?xml version="1.0" encoding="utf-8"?>

<soapenv:Envelope

xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Figure 2.10. message SOAP pour interroger un service Web

SOAP est un protocole limité par sa simplicité et ses faibles performances. Des alternatives apparaissent aujourd'hui, notamment pour apporter plus d'efficacité.

3.5. WS-BPEL

L'assemblage de services Web repose sur l'orchestration, il n'existe pas de composition structurelle de services Web. WS-BPEL [28], acronyme de *Web Services Business Process Execution Language*, est une spécification du consortium OASIS. Elle en est à sa version 2.0 depuis mars 2007. Cette spécification est l'une des plus connues pour l'orchestration de services Web. Elle remplace les précédentes spécifications XLANG[21] de Microsoft, et WSFL[22] d'IBM.

WS-BPEL est un langage de procédés basé sur la technologie XML, tout comme les autres standards des services Web. WS-BPEL permet de construire des procédés interprétables et exécutables par un moteur d'orchestration. Les procédés peuvent être modélisés de deux manières :

- **abstraite** : à ce niveau, seuls les échanges de messages entre les différents participants sont spécifiés. Mais le comportement interne de ces participants n'est pas explicité.
- **exécutable** : les activités du procédé sont ordonnées, les partenaires impliqués sont identifiés ainsi que les messages qui sont échangés. A ceci s'ajoute le traitement des fautes et des exceptions pour les cas d'erreurs.

Un procédé est composé d'activités qui s'enchaînent grâce à des échanges de données. Les activités peuvent être de deux types : basiques ou complexes. Les activités basiques sont des types de

[19] HTTP: HyperText Transfer Protocol
[20] SMTP : Simple Mail Transfer Protocol
[21] XLANG: http://www.ebpml.org/xlang.htm
[22] WSFL: *Web Services Flow Language*, http://xml.coverpages.org/wsfl.html

base comme *invoke* pour appeler un service Web, *receive* pour attendre une invocation, *reply* pour une réponse... A partir de ces types de base, il est possible de créer des activités composites avec des structures de construction du contrôle du flot de données : *flow* pour une ou plusieurs activités concurrentes, *sequence* pour une séquence d'activités, *switch* pour des conditions, *while* pour une boucle... Il faut quand même noter que l'échange de données n'existe pas en tant que tel, puisqu'il faut passer par des affectations de variables entre les activités. En général, ces outils sont munis d'interfaces graphiques qui simplifient la réalisation des compositions de services.

L'exécution des procédés se fait avec un moteur d'exécution spécifique. Il existe actuellement sur le marché de nombreux moteurs d'exécution de procédés comme Orchestra[23] de Bull, ActiveBPEL[24] qui sont *open source* et en Java, Oracle BPEL Process Manager[25] ou encore Netbeans BPEL[26] Service Engine. Le moteur d'exécution est centralisé et permet d'exécuter la composition de services définie sous forme d'orchestration. Il permet de réaliser les communications avec les services concrets et l'invocation des fonctionnalités de ces services.

WS-BPEL est aujourd'hui très limité et de nombreux travaux sont en cours pour l'étendre. Deux cas privilégiés sont l'utilisation de spécification abstraite de service, comme dans SCA, et l'ajout de propriétés non-fonctionnelles, comme la sécurité.

3.6. Synthèse

La technologie des services Web est l'une des plus connues et répandues pour appliquer les principes de l'approche orientée service. Elle est basée sur le standard XML et permet de rendre disponibles dans un annuaire UDDI des services dont les fonctionnalités sont décrites dans des fichiers WSDL. Le protocole SOAP permet aux consommateurs d'interroger l'annuaire et d'invoquer le service facilement à travers l'Internet ou un Intranet. Cependant, ces technologies ne tiennent pas compte d'un certain nombre de caractéristiques importantes, comme les propriétés non-fonctionnelles, même si de nombreuses spécifications ont été proposées. Elles sont d'ailleurs tellement nombreuses qu'il est difficile de toutes les maîtriser. On peut penser qu'une stabilisation interviendra, mais ce n'est pas encore le cas aujourd'hui.

Avec la spécification WS-BPEL, les services Web font une avancée en ce qui concerne la composition des services. Cette composition est une orchestration de services qui a les avantages des compositions par procédés mais aussi les inconvénients comme le problème du moteur d'exécution centralisé. Cependant, les services Web ne supportent pas les mécanismes du dynamisme de l'architecture à services.

Le Tableau 1 récapitule les informations concernant la technologie des services Web en fonction des caractéristiques importantes de l'approche à services.

[23] http://orchestra.objectweb.org/xwiki/bin/view/Main/WebHome
[24] http://www.activevos.com/community-open-source.php
[25] http://www.oracle.com/technology/products/ias/bpel/index.html
[26] https://open-esb.dev.java.net/kb/preview3/ep-bpel-se.html

	Services Web
Spécification	– Interface WSDL – Des propriétés non-fonctionnelles
Implantation	– Nombreux langages – Génération des talons clients
Découverte	– UDDI – Découverte active
Composition	– Oui, orchestration WS-BPEL
Communication	– SOAP sur HTTP – Synchrone
Liaison	– Type : direct – Statique

Tableau 1 : Récapitulatif de la technologie des services Web

Pour conclure, nous pouvons noter que les services Web représentent une technologie très répandue pour l'intégration d'applications au sein d'une entreprise, comme présentée dans la section 7 page 65. Ils seront aussi de plus en plus utilisés pour la communication entre différentes entreprises grâce à des annuaires distribués (*Business to Business* (B2B)). Cependant, les services disponibles sur le Web restent encore une vue de l'esprit, en attendant que Microsoft lance le *Windows Cloud* qui se base sur le concept de *Cloud Computing*, c'est-à-dire que des applications sur serveurs Web sont accessibles par des connexions Internet au lieu d'être installées sur des postes clients. Ce concept fait partie de ce qui est appelé *Software as a service* (SaaS), mais chez Microsoft, on parle plutôt de *software plus service*.

4. Composants orientés service

4.1. Principes

L'approche à composants est une approche antérieure à l'approche à services. Elle préconise l'assemblage de composants pour réaliser des applications [36]. Un composant peut être vu comme une brique logicielle qui est faite pour être assemblée avec d'autres composants [37]. Les composants tentent de répondre à deux objectifs :

- le *megaprogramming*, c'est-à-dire la programmation à partir de modules, logiciels gros grains pour améliorer la productivité et faire face à la taille toujours croissante des logiciels. Il s'agit également de favoriser la réutilisation.
- l'amélioration de la modularité des applications qui n'est pas fournit de façon suffisante avec des approches à objets par exemple.

Les composants ont été conçus principalement pour être réutilisés dans différentes applications, sans que l'implantation du composant soit connue. Il suffit d'avoir une connaissance des fonctionnalités du composant et des moyens de l'assembler. Dans la suite, nous détaillons les caractéristiques d'un composant ainsi que les bénéfices qui ont été tirés de l'approche à composants et de l'approche à services pour ce que l'on appelle l'approche à composants orientés service.

4.1.1. Les composants

Il n'existe pas de définition consensuelle de la notion de composant. Cependant, l'une des plus citées et connues est celle de Szyperski :

> *"A software component is a unit of composition which contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [38]*

Cette définition met en évidence les concepts importants qui caractérisent un composant :

- **la spécification d'interfaces :** un composant expose les fonctionnalités qu'il fournit ;
- **les dépendances explicites :** un composant présente les besoins qu'il requiert pour réaliser ses fonctionnalités. Il existe en fait deux types de dépendances :
  - la dépendance « logique » sur des fonctionnalités d'autres composants ;
  - la dépendance « physique » sur du code, par exemple des bibliothèques.

- **l'instanciation :** un composant fournit un type et peut être instancié plusieurs fois ;
- **l'indépendance de déploiement :** un composant est une unité de déploiement à part entière. Toutefois cette caractéristique est controversée.

Les composants ayant été créés pour être assemblés, il faut ajouter à la notion de composant un mécanisme de composition illustré dans la figure 2.11. En fait, chaque approche à composants définit un langage de description d'architecture d'application à partir de composants, appelé ADL[27]. Les ADL, qui n'ont pas été faits à l'origine pour les composants, sont basés sur deux éléments :

- **les instances :** les instances de composants forment le logique métier de l'application, ce sont les éléments requis à la réalisation des fonctionnalités de l'application ;
- **les connecteurs :** ce sont les liaisons qui sont réalisées entre les différentes instances de composants de l'application. Ces liaisons sont déterminées en fonction des propriétés fournies et requises par les différents types de composants utilisés.

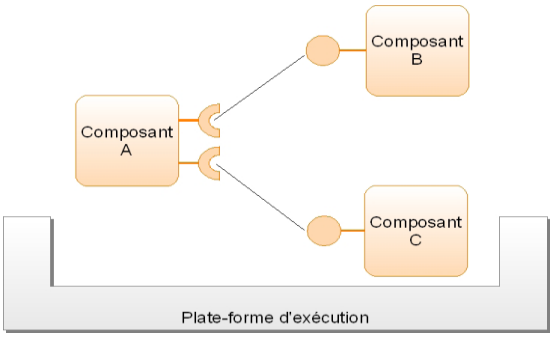


Figure 2.11. Exemple d'assemblage de composants

[27] ADL: Architecture Description Language

A cette approche, qui définit les caractéristiques de composants et leur composition, est associé un environnement d'exécution qui prend en charge la réalisation de l'application résultante. L'environnement d'exécution est parfois comparé à un mini-système d'exploitation [39] car il est chargé de gérer des aspects divers tels que le cycle de vie des instances ou bien les propriétés non-fonctionnelles.

D'après les principes de l'approche à composants qui ont été cités précédemment, nous pouvons noter que son point fort est la gestion complète du cycle de vie des composants : en particulier, le déploiement et l'instanciation ne sont que peu traités par l'approche à services. Par contre, dans cette approche, la méthode d'implantation du composant n'est pas du tout abordée.

Toutefois, il est intéressant de noter que, au sein de la communauté scientifique des approches à composants[28], une approche particulière a été proposée en se basant sur le principe de la séparation des préoccupations. La structure des composants suit ce principe, c'est-à-dire que le code fonctionnel du composant est clairement identifié et séparé du code non-fonctionnel, comme par exemple les transactions, la persistance ou la sécurité. Une des approches les plus utilisées pour appliquer la séparation des préoccupations est l'utilisation des conteneurs. Un conteneur est illustré ci-après :

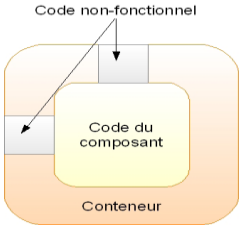


Figure 2.12. Structure d'un composant

Le code métier du composant est englobé dans un conteneur qui prend en charge certains aspects non-fonctionnels en suivant les spécifications données par l'utilisateur ainsi que la gestion du cycle de vie du composant. Il existe de nombreuses implantations de modèles à composants détaillées et comparées dans [40]. Ces implantations se focalisent en général sur un unique aspect non-fonctionnel, par exemple, la communication par événements des JavaBeans [41], ou encore les transactions et la persistance des données gérée par les EJB[29] [41]. Comme un certain nombre d'aspects non-fonctionnels sont récurrents dans les approches à composants, il existe des bibliothèques standards de code non-fonctionnel. Cependant, certaines approches laissent l'utilisateur libre de définir ses propres aspects non-fonctionnels ; ces approches sont dites approches à composants extensibles, Fractal [48] en est un exemple.

4.1.2. Approche à composants orientés service

L'approche à composants orientés service, proposée en 2004 dans [42], combine les avantages de l'approche à composants et de l'approche à services. Les concepts de l'approche à services enrichissent le modèle à composants. A ceux-ci, s'ajoutent des mécanismes permettant aux applications de s'adapter pour supporter la disponibilité dynamique des composants à l'exécution.

[28] CBSE: Component-Based Software Engineering
[29] EJB: Enterprise JavaBeans

Cette approche est basée sur les principes suivants :

- **un service fournit une fonctionnalité.** Un service propose un ensemble d'opérations réutilisables.
- **un service est décrit par une spécification de service** qui contient des informations syntaxiques et, éventuellement, comportementales et sémantiques. Cette spécification est utile pour la composition de services, les interactions et la découverte. Dans la partie syntaxique de cette spécification, les dépendances éventuelles avec d'autres services sont annoncées.
- **un composant implante une spécification de service.** Le composant doit respecter les contraintes qui ont été définies dans la spécification. Si l'implantation impose d'autres contraintes, celles-ci doivent être énoncées.
- **le modèle d'interaction de l'approche à services dynamique est utilisé pour résoudre l'ensemble des dépendances du système.** Les services sont sous forme d'instances de composants qui sont enregistrées dans un annuaire. Cet annuaire est utilisé lors de l'exécution pour découvrir les instances qui résolvent les dépendances des services.
- **les compositions sont décrites sous forme de spécifications de services.** La composition est une spécification de services qui permet de choisir les composants à instancier. La composition devient alors concrète à partir du moment où les composants sont découverts et instanciés. Les liaisons ne sont pas définies de manière explicite, elles sont inférées à partir des dépendances des services.
- l**es spécifications de services sont les fondements de la substituabilité.** Dans une composition, un composant peut être remplacé par n'importe quel autre composant pourvu qu'il respecte la même spécification de service.

Il est clair qu'avec cette approche à composants orienté service, on peut bénéficier des avantages de l'approche à composants : un modèle de développement simple et une description de la composition ; ainsi que des avantages de l'approche à services : un faible couplage et le dynamisme. Dans la suite, nous présentons deux technologies OSGi et iPOJO qui mettent en œuvre cette approche.

4.2. OSGi

OSGi [24] est une spécification de plate-forme à services qui a été proposée par le consortium OSGi Alliance qui réunit de nombreux acteurs tels que IBM, Motorola, Nokia,...La spécification OSGi avait pour but à l'origine de proposer une plate-forme pour les passerelles réseaux et résidentielles. Ces passerelles sont limitées en taille mémoire et en capacité d'exécution. Elles ont en plus des contraintes fortes en ce qui concerne le cycle de vie des composants logiciels ; puisque, dans ces domaines, de nouveaux composants logiciels doivent pouvoir être déployés et assemblés dynamiquement sur la passerelle, sans interrompre l'exécution des autres composants logiciels installés. Aujourd'hui avec l'émergence de nouveaux domaines d'application pour l'informatique embarquée, tels que la téléphonie ou encore l'automobile, les plates-formes OSGi deviennent de fait des standards pour le développement d'applications dynamiquement reconfigurables. OSGi devient aussi un standard puisqu'il est utilisé pour des serveurs d'applications tels qu'OW2 JOnAS [43] ou Sun Glassfish[44] et pour des applications à *plug-ins* telles que l'IDE Eclipse [45].

Un des points forts de la spécification OSGi est de prendre en compte le cycle de vie complet du logiciel, c'est-à-dire de son conditionnement jusqu'à sa désinstallation. Pour cela, OSGi propose un

système de gestion de déploiement d'applications spécifique. Ce système permet de déployer indépendamment différents morceaux d'une application, l'unité de déploiement utilisée est le *bundle*. Ces unités de déploiement 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. Le cycle de vie d'un *bundle* est représenté par la Figure 2.13. Une des caractéristiques des *bundles* OSGi est qu'ils spécifient leurs dépendances qu'ils ont avec les autres unités de déploiement au niveau des *paquetages* fournis et requis. Ceci permet de contrôler au démarrage d'une application que toutes les dépendances de code sont résolues.

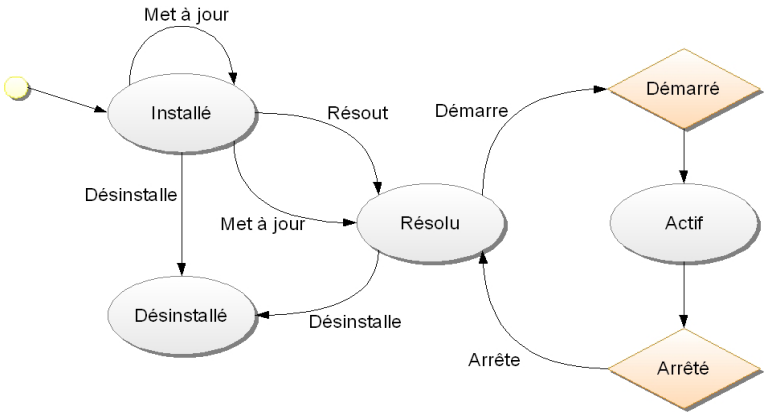


Figure 2.13. Cycle de vie d'un bundle OSGi

La spécification OSGi spécifie une architecture orientée service dynamique. Elle s'appuie sur la couche de déploiement précédemment présentée mais aussi sur le langage Java. OSGi est une plate-forme centralisée avec un registre de services. Les services sont décrits à l'aide d'interface Java et d'un ensemble de propriétés dont la sémantique est laissée libre aux utilisateurs. L'invocation d'un service se fait par l'intermédiaire d'un appel de méthode, identique aux appels de méthodes de l'approche objet de Java. La Figure 2.14 illustre l'architecture orientée service de la spécification OSGi.

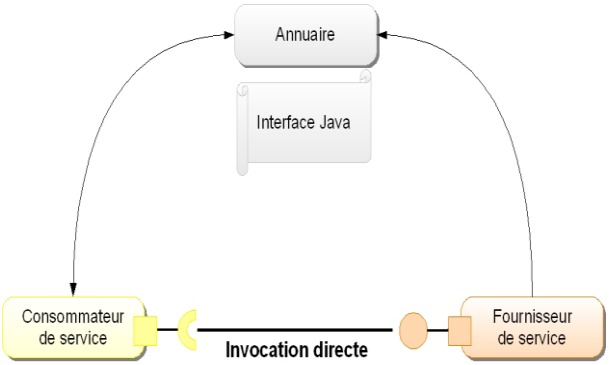


Figure 2.14. Architecture de OSGi.

Le principe de fonctionnement de l'architecture à services proposée par la spécification est le suivant :

- un fournisseur de service publie son service au niveau du registre de services fournit par OSGi. Le service est décrit par une interface Java qui définit sa partie fonctionnelle ainsi que la syntaxe pour l'appeler. Cette interface est complétée par un ensemble de propriétés qui sont à l'appréciation du fournisseur.
- un consommateur de service interroge l'annuaire de services pour rechercher le service qui convient à ses besoins qui peuvent être sur la partie fonctionnelle du service ou bien sur ses propriétés complémentaires. La requête est traitée par l'annuaire qui liste quels sont les fournisseurs disponibles répondant aux besoins.
- le consommateur invoque directement un des fournisseurs de services proposés par l'annuaire.

La spécification OSGi procure une architecture orientée service dynamique centralisée. Le dynamisme de cette architecture est supporté en grande partie par l'annuaire de service qui permet de retrouver à tout moment l'ensemble des fournisseurs de services disponibles. Il maintient une correspondance entre les descriptions de services et les fournisseurs de services disponibles. A cela, la plate-forme OSGi ajoute un ensemble de primitives et de mécanismes pour créer des applications capables d'être informées dynamiquement de la disponibilité des services. Un fournisseur de service peut alors être retiré et, par conséquent, les services qu'il fournit deviennent indisponibles. Aucun consommateur ne peut alors l'utiliser.

Ils existent plusieurs implantations de la plate-forme OSGi autant commerciales que *open source* telles que Apache Felix[30] de la communauté Apache, Equinox[31] de IBM ou encore Knopflerfish[32] maintenu par Makewave.

La spécification permet les interactions prévues par l'architecture à services mais celles-ci doivent toutes être gérées manuellement par le développeur. Cette tâche est délicate et demande une grande connaissance des mécanismes OSGi pour bien traiter tous les cas possibles afin d'éviter les erreurs. Pour palier à ce problème de gestion manuelle des interactions, la technologie iPOJO, présentée dans la section suivante, propose de les gérer automatiquement.

4.3.iPOJO

La technologie Apache iPOJO [47, 46], acronyme de *injected Plain Old Java Object*, est une implantation des principes de l'approche à composants orientés service. Elle a été développée au sein de l'équipe Adèle du Laboratoire d'Informatique de Grenoble (LIG). Elle est un sous-projet du projet Apache Felix, qui est lui-même une implantation *open source* de la spécification OSGi R4.

L'objectif principal d'iPOJO est de fournir une plate-forme d'exécution facilitant le développement d'applications basées sur le principe de l'architecture à services dynamique. Cette plate-forme repose sur la spécification OSGi, et par conséquent, elle hérite d'une partie de ses caractéristiques : elle est en Java, centralisée et supporte le dynamisme. Dans l'optique de simplifier le développement des applications à services dynamiques, iPOJO propose d'appliquer le principe de l'approche à composants orientés service, c'est-à-dire que les services sont implantés sous forme de composants.

[30] Apache Felix: http://felix.apache.org/site/index.html
[31] Equinox : http://www.eclipse.org/equinox/
[32] http://www.knopflerfish.org/

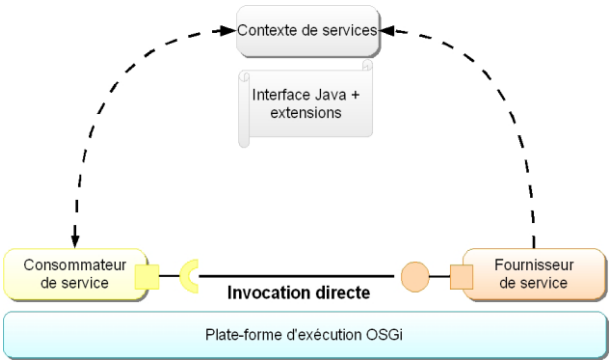


Figure 2.15. Architecture de iPOJO

La technologie iPOJO propose :

- **un modèle et une machine d'exécution fortement liés :** tous les éléments du modèle existent au moment de l'exécution, ceci afin de rendre plus compréhensible la structure de l'application.
- **une architecture à services dynamique avec isolation :** les interactions de l'architecture à services dynamique sont issues de l'architecture de OSGi. Cette architecture propose aussi un principe d'isolation des services pour que certains services d'une application puissent rester privés.
- **un modèle de développement simple :** il permet de cacher toute la complexité de l'architecture SOA en utilisant les idées du modèle de développement des POJO [49] et surtout il cache toute la complexité du dynamisme de OSGi. Pour cela, la machine d'exécution d'iPOJO fournit une machine d'injection et d'introspection permettant une gestion transparente de ces préoccupations pour le développeur.
- **un langage de composition structurelle :** iPOJO permet de construire des applications à partir d'une composition structurelle de services. La composition est modélisée avec des spécifications de services indépendamment de l'implémentation. Ce découplage est un point fort de iPOJO puisqu'ainsi à l'exécution l'infrastructure choisit une implémentation disponible. Les applications conçues ainsi sont gérées de manière dynamique.
- **des fonctionnalités d'introspection et de reconfiguration dynamique :** ces mécanismes permettent d'avoir un retour sur la structure du système qui change grâce au dynamisme.
- **des mécanismes d'extension :** ils permettent d'ajouter facilement des propriétés non fonctionnelles comme la persistance, la sécurité ou la qualité de service.

Grâce à ces mécanismes, iPOJO permet de construire plus facilement qu'auparavant des applications à services dynamiques tout en respectant les principes de l'architecture à services dynamique. La simplification de la gestion du dynamisme est un apport très important qui aide considérablement le développeur. De plus, la possibilité de gérer uniquement les propriétés non-fonctionnelles que souhaite l'utilisateur est un atout.

La technologie iPOJO est aujourd'hui utilisée dans un nombre croissant de produits, comme JOnAS [43] de Bull et les passerelles domotiques.

4.4. Synthèse

Les deux technologies OSGi et iPOJO sont deux technologies qui sont des implantations des principes de l'approche à composants orientée services. Ces deux technologies respectent les mécanismes de l'approche à composants mais aussi ceux de l'approche à services. Le Tableau 2 récapitule les principales fonctionnalités de ces deux technologies.

	OSGi	iPOJO
Spécification	– Java – Aucune propriété non-fonctionnelle	– Java – Propriétés non-fonctionnelles – Extensions possibles
Implantation	– Java – Conditionnée dans des *bundles* – Deux niveaux de dépendances	– Java, composite – Conditionnée dans des *bundles* – Trois niveaux de dépendances
Découverte	– Annuaire de services OSGi – Découverte active et passive	– Contexte, pas d'annuaire global – Découverte active et passive
Composition	– Structurelle – Sans ADL	– Structurelle – ADL implicite
Communication	– Java – Synchrone, centralisée	– Java – Synchrone, centralisée
Liaison	– Type : indirect – Autonomique	– Type : indirect – Dynamique

Tableau 2 : Comparatif des technologies OSGi et iPOJO

Nous pouvons constater que iPOJO hérite de tous les avantages de OSGi, comme le dynamisme, mais étend une partie des fonctionnalités, par exemple, il est possible de gérer les propriétés non-fonctionnelles. Le principal avantage de iPOJO est l'automatisation de la gestion du dynamisme qui est une tâche très difficile pour un développeur s'il souhaite réaliser correctement une application. OSGi et iPOJO sont cependant des technologies centralisées qui supportent uniquement le langage Java.

OSGi et les composants orientés service tels qu'iPOJO deviennent très populaires. OSGi est par exemple devenu incontournable dans le domaine des serveurs d'applications (JOnAS, Websphere, Glassfish...) et iPOJO est aujourd'hui intégrer dans JOnAS et bientôt dans Glassfish.

5. Autres technologies à services

Dans les sections précédentes, nous nous sommes focalisés sur les technologies à services qui sont en plein essor grâce à leurs avancées et leur réactivité face aux différentes demandes du marché. Cependant, les services Web avec WS-BPEL pour leurs compositions, OSGi et iPOJO ne sont pas les seuls standards existants pour mettre en oeuvre l'architecture à services.

Dans cette partie, nous allons détailler CORBA et Jini qui font partie des premières spécifications de cette architecture. Puis, nous nous intéresserons aux technologies UPnP et DPWS qui voient le jour dans le domaine de la domotique. C'est un environnement qui s'appuie sur une grande partie des caractéristiques du SOA, telles que le faible couplage, l'hétérogénéité des équipements, le dynamisme... Finalement, nous présenterons la spécification la plus connue pour les composants qui est le modèle SCA.

5.1. CORBA et Jini

5.1.1. CORBA

CORBA [50], acronyme de *Common Object Request Broker Architecture*, est une norme qui a été rédigée par un groupe de travail nommé *Object Management Group* (OMG). Dans ce groupe, on retrouve les principaux acteurs informatiques tels que Sun Microsystems, Oracle ou encore IBM. L'OMG travaille à la définition de « l'architecture distribuée idéale », c'est-à-dire une architecture distribuée dans laquelle des clients émettent des requêtes à destination d'objets, qui s'exécutent dans des processus serveurs. Ces objets répondent aux clients par la transmission d'informations bien que ces deux éléments (client et serveur) soient localisés dans des espaces mémoire différents, généralement situés sur des machines distantes.

La norme CORBA a été créée en 1992 et supporte l'hétérogénéité et l'interopérabilité entre les différents langages de programmation et les environnements informatiques. Ceci est dû en grande partie au principe du langage de description d'interface[33] (IDL). En 1996, la deuxième version de CORBA sort avec comme innovation le protocole de communication IIOP27et des améliorations pour la conversion des langages de programmation en IDL et inversement. La dernière version de CORBA, qui date de 2002, s'enrichit avec l'ajout de seize types de services tels que celui de nommage, l'annuaire des objets, le cycle de vie, la notification d'événements, la sécurité... Parmi ces services, le plus important est le service d'annuaire sur lequel repose la mise en place d'applications dynamiques avec l'infrastructure CORBA. Avec le service d'annuaire, CORBA suit les principes de l'architecture orientée service comme le montre la Figure 2.16.

[33] IDL : Interface Description Language

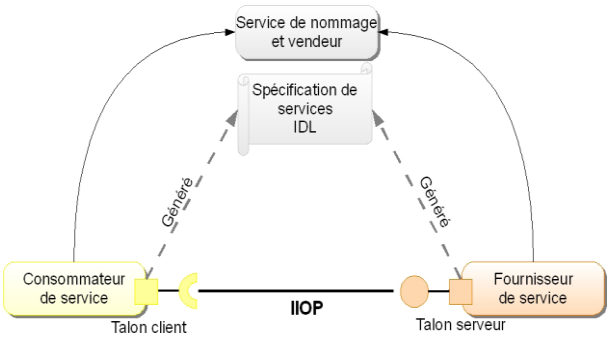


Figure 2.16. Architecture de CORBA

Un service est décrit avec le langage de description d'interfaces, cette description contient une référence vers l'interface du service ainsi que les propriétés qui le caractérisent. Les informations contenues dans les descriptions sont uniquement syntaxiques et restreintes en nombre. Le fournisseur enregistre la description de service, et ensuite celle-ci est publiée. Le service est alors disponible pour les clients.

Un consommateur de service peut rechercher un service dans un annuaire. La requête d'interrogation de l'annuaire peut être plus ou moins complexe en fonction des besoins du client. Parmi les services disponibles retournés par l'annuaire, le client en choisit un et il peut alors l'invoquer en utilisant les méthodes de son interface. En général, le protocole IIOP[34] est utilisé pour interroger un service.

Dans la spécification CORBA, un client peut interroger un annuaire ou une fédération d'annuaires. En effet, pour augmenter les chances d'obtenir un service qui corresponde parfaitement aux attentes du client, un annuaire peut interagir avec d'autres annuaires tout en restant transparent lors de son utilisation.

CORBA suit les principes de l'architecture orientée service mais ne supporte pas la notification de l'arrivée ou du départ d'un service comme dans l'architecture à service dynamique. Un consommateur doit toujours s'assurer de la disponibilité du service avant de l'utiliser et le libérer sitôt qu'il n'en a plus l'utilité. Cependant, la composition de services CORBA n'est pas fournie dans la spécification, certains travaux proposent des compositions comportementales [52] ou structurelles [51].

5.1.2. Jini

Jini [22] est une spécification proposée à l'origine par Sun Microsystems. Actuellement, cette spécification fait partie du projet River[35], géré par Apache. Elle permet de construire un réseau de dispositifs communicants capables de fonctionner avec une machine virtuelle Java (JVM). L'utilisateur a ainsi la possibilité d'accéder facilement à distance à ce réseau et d'utiliser tous les dispositifs qui y sont branchés. Jini est indépendant de tout système d'exploitation.

[34] IIOP: Internet Inter-Orb Protocol
[35] http://incubator.apache.org/projects/river.html

Les dispositifs communicants supportés par Jini peuvent être autant matériels que logiciels, ils sont considérés comme des objets indépendants qui peuvent communiquer. Ainsi, Jini peut les réunir en *fédération* d'objets qui s'installent automatiquement et qui fonctionnent dès qu'ils sont branchés dans un réseau local dynamique.

Jini définit un service comme une entité logicielle qui peut être utilisée par l'ensemble des acteurs du réseau, c'est-à-dire une personne, un programme ou un autre service. Un service est une fonctionnalité utilisable à distance, comme par exemple, un service d'impression offert par une imprimante. Le service ainsi que ses propriétés sont décrits avec une interface Java. Cette définition n'est que syntaxique.

Jini suit les principes de l'architecture à services ; nous retrouvons donc dans son architecture le fournisseur de service, le service de recherche et le consommateur de service en tant qu'acteurs, comme représentés dans la Figure 2.17.

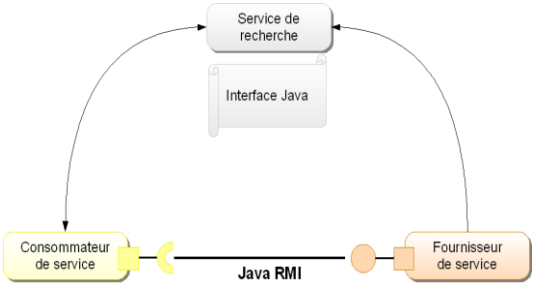


Figure 2.17. Architecture de Jini

Un service est enregistré dans un service de recherche par son fournisseur. Le service de recherche peut être contacté à une adresse fixe ou bien découvert. Une fois que le service est enregistré et publié par le service de recherche, le client doit découvrir le service de recherche. Ensuite, il peut rechercher et utiliser le service. A ces mécanismes de l'architecture à services, Jini en ajoute un pour gérer de façon explicite les politiques de libération des instances des services avec un système de bail. Lors de la publication du service par le fournisseur dans le registre, il est possible d'indiquer la durée du bail. Lorsque le bail se termine, le fournisseur doit renouveler son bail, sinon le service est supprimé du registre. Ce mécanisme de bail est aussi utilisé pour la découverte passive de services par le consommateur.

Toutefois avec ces briques de base de l'architecture à service, Jini ne propose pas de modèle de composition de services. Le développeur est libre de gérer la composition de services ainsi que le dynamisme dû aux arrivées et aux départs des services. Cependant, un langage de composition a été proposé par Huang [111] dans le but d'exécuter des compositions comportementales au-dessus de Jini. Ceci consiste simplement à traduire en Java la composition écrite dans ce langage.

5.1.3. Synthèse

CORBA et Jini font partie des plus anciennes technologies qui implantent les principes de l'architecture orientée service. Le Tableau 3 récapitule les informations de ces technologies en fonction des éléments intéressants de l'architecture à services.

	Corba	Jini
Spécification	– IDL – Aucune propriété non-fonctionnelle	– Interface Java – Aucune propriété non-fonctionnelle
Implantation	– Nombreux langages – Génération des talons client et serveur – Déploiement manuel des *stubs* et *skeletons*	– Java
Découverte	– Service de nommage et vendeur – Découverte active	– Service de recherche – Découverte active et passive
Composition	– Non (Client/Serveur)	– Non
Communication	– Principalement IIOP – Synchrone, asynchrone, publication/souscription	– Principalement RMI – Synchrone, asynchrone, décentralisée
Liaison	– Type : indirect – Dynamique	– Type : indirect – Dynamique

Tableau 3 : Comparatif des technologies Corba et Jini

Ces deux technologies supportent par défaut les principes de base de l'architecture : les services sont spécifiés et implantés, ils peuvent être découverts. Jini est restreint l'implantation des services à unique langage alors que CORBA permet d'intégrer plusieurs langages de programmation. Cependant, la composition de services n'est pas prise en considération par ces technologies, c'est une limite à leur utilisation aujourd'hui.

5.2. UPnP et DPWS

La popularité des objets communicants comme, par exemple, les téléphones portables, les assistants personnels, etc , ont poussé le développement des technologies de communication comme le Bluetooth[36], UPnP et DPWS. Ce sont des protocoles de haut niveau, qui cachent la complexité des technologies utilisées tout en permettant non seulement des communications sans fils entre différents équipements hétérogènes de réseaux domestiques ou d'entreprise mais aussi, pour UPnP et DPWS, leur découverte. Dans cette partie, nous allons détailler UPnP et DPWS qui sont deux spécifications basées sur le principe de l'architecture orientée service dynamique et dédiées aux équipements des réseaux domestiques ou d'entreprise.

[36] Bluetooth: http://www.bluetooth.com/

5.2.1. UPnP

UPnP [23], acronyme de *Universal Plug and Play*, est une spécification issue d'une initiative industrielle et gérée par l'UPnP Forum. L'objectif de cette spécification est de pouvoir simplifier au maximum la connexion de dispositifs communicants hétérogènes et la mise en place de réseaux domotiques. UPnP est inspiré et dérivé de la technologie *Plug and Play*, qui permet d'attacher de manière dynamique de nouveaux périphériques à un ordinateur.

La spécification UPnP propose un ensemble de protocoles, basés sur des protocoles standards de l'Internet, pour mettre en place une architecture orientée service dynamique spécialisée dans la domotique. La Figure 18 illustre cette architecture de UPnP :

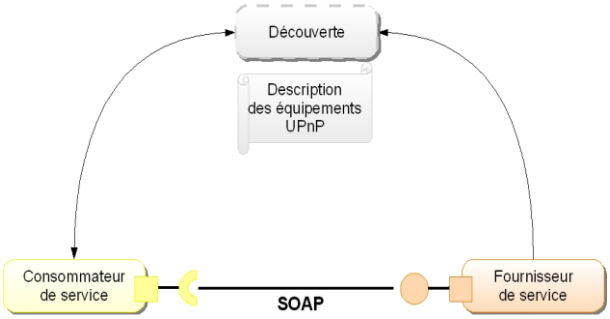


Figure 2.18. Architecture de UPnP

Le but de UPnP est de pouvoir découvrir de nouveaux équipements dans un réseau, c'est pourquoi les équipements ainsi que leurs services sont décrits dans un format propre à UPnP et basé sur XML. Lorsqu'un équipement rejoint un réseau, il s'annonce auprès de tous les autres intervenants présents dans ce réseau. Pour qu'un équipement ait une connaissance des autres équipements du réseau, il peut émettre une requête pour récupérer les informations des différents équipements. Comme nous sommes dans une architecture à services dynamique, il est aussi possible qu'un équipement soit arrêté. Dans ce cas, cet équipement est capable d'envoyer un message pour notifier de son départ aux autres équipements.

Finalement, UPnP se propose de fournir une spécification pour la communication et la découverte d'équipements et de leurs services dans des réseaux domestiques ou d'entreprise. L'évolution des technologies dans le domaine de la domotique nécessite de pouvoir mettre en place des applications en fonction des équipements découverts dans les maisons, c'est pourquoi UPnP commence à devenir un standard de plus en plus populaire.

Toutefois, il est clair que UPnP est centré sur les équipements plutôt que sur les applications que l'on peut réaliser avec ces équipements, ce qui est un obstacle à l'intégration de ces équipements dans les applications. Mais, il manque aussi des fonctionnalités importantes à UPnP. En particulier, la sécurité n'est pas du tout abordée, alors qu'il est essentiel dans des réseaux domotiques de pouvoir restreindre certains équipements et/ou services avec des authentifications par exemple.

5.2.2. DPWS

DPWS [54, 53] est l'acronyme de *Device Profile for Web Services*. C'est une spécification [55] proposée et supportée par Microsoft. Windows Vista, qui est la dernière génération de système d'exploitation de Microsoft, intègre nativement cette spécification. Cette dernière permet de simplifier la mise en place d'appareils dans un réseau domestique ou dans une entreprise. La politique affichée par Microsoft avec cette stratégie monopolistique est de remplacer UPnP.

La spécification DPWS se base sur la technologie des services Web. Elle part du constat que les services Web sont aujourd'hui une technologie très riche en standards et en fonctionnalités qui répondent à de nombreux besoins. Cependant, pour être adaptable à tous les services Web, Microsoft propose un noyau de fonctionnalités restreintes basées sur des spécifications des services Web :

- **l'envoi et la réception** de messages sécurisés pour un service Web avec la spécification WS-Security [33],
- **la découverte dynamique** de serviceWeb avec la spécification WS-Discovery [56],
- **la description** d'un serviceWeb avec la spécification WS-MetadataExchange [57],
- **la souscription et la réception** d'événements pour un service Web avec la spécification WS-Eventing [58].

L'architecture de DPWS suit le principe de l'architecture orientée service, en particulier celle des services Web, elle est illustrée dans la Figure 2.19 ci-après :

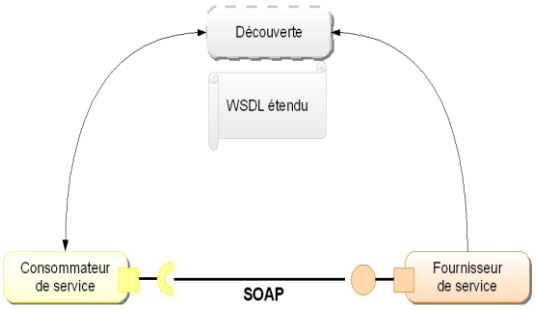


Figure 2.19. Architecture de DPWS

Après avoir obtenu une adresse IP [37] sur le réseau, l'équipement doit découvrir tous les équipements du réseau et signaler son existence ainsi que les services qu'il propose. En général, ces communications sont réalisées en mode *multicast*. Les fonctionnalités d'un équipement sont données dans un fichier de description WSDL étendu. Ensuite, il est possible de demander à un équipement un service. Cette communication se fait avec le protocole SOAP et peut être asynchrone.

Pour conclure, la spécification de DPWS est fortement inspirée de celle de UPnP ; elle se veut d'ailleurs être la nouvelle version de UPnP. Les fonctionnalités offertes sont semblables à celles d'UPnP, à la différence qu'elles sont fournies de façon plus propre et plus précise, grâce, entre autres, aux spécifications des services Web. Un autre avantage de cette spécification est l'ajout de la sécurité.

[37] WS-Addressing [59]

Cependant, malgré les améliorations apportées et les efforts de Microsoft, DPWS n'est implanté que sur très peu d'équipements et il est limité aux des réseaux locaux ; par conséquent, l'adoption de ce standard risque d'être un peu longue.

5.2.3. Synthèse

Les deux technologies UPnP et DPWS sont en train de devenir des standards pour la domotique. Elles s'appuient sur l'architecture orientée service dynamique. Ainsi les équipements sont décrits et disponibles pour être utilisés par des clients. Nous présentons dans le Tableau 4 un récapitulatif des points importants de l'architecture à services pour UPnP et DPWS.

	UPnP	DPWS
Spécification	– Description d'équipement UPnP – Aucune propriété non-fonctionnelle	– WSDL étendu – Propriété non-fonctionnelle : sécurité
Implantation	– Nombreux langages – Serveur HTTP côté équipement – Proxy généré côté client	– Nombreux langages – Serveur HTTP côté équipement – Proxy généré côté client
Découverte	– Déclaration *multicast* – Découverte active et passive	– Déclaration *multicast* – Découverte active et passive
Composition	– Client/Serveur	– Client/Serveur
Communication	– SOAP – Synchrone, à événements	– SOAP – Synchrone, à événements
Liaison	– Type : direct – Dynamique	– Type : direct – Dynamique

Tableau 4 : Comparatif des technologies UPnP et DPWS

Le Tableau 4 montre que les deux technologies UPnP et DPWS sont très proches. La spécification de DPWS a montré un effort de standardisation au niveau du fichier de description. La mise en place d'un standard commun à tous les équipements facilite leur utilisation. Un autre atout très important dans DPWS est la gestion de messages sécurisés, ceci permet de restreindre l'accès à certains équipements et ainsi de réaliser des applications plus sûres.

Le point le plus important pour ces technologies est le dynamisme issu de l'architecture orientée service dynamique. Le dynamisme permet en effet d'envisager des applications réactives à l'ajout ou au retrait d'équipements dans un réseau domestique ou d'entreprise. Par exemple, dans une maison, on peut créer une application qui utilise une télévision et un assistant personnel, il est clair que la télévision restera dans la maison alors que l'assistant personnel risque de suivre les déplacements de son propriétaire à l'extérieur de la maison.

Enfin, on peut regretter le manque de maturité logicielle de ces deux technologies. L'expérience montre que la mise en place d'applications avec UPnP et/ou DPWS requiert un développement logiciel et, surtout, une phase de tests et de robustesse très coûteuse. On peut également douter de la capacité de ces technologies à passer à l'échelle. UPnP, par exemple, est très verbeux et émet de nombreux messages sur le réseau.

5.3.SCA

Le consortium OSOA[38] a été à l'initiative de la spécification SCA, acronyme de *Service Component Architecture*, pour définir une architecture orientée service. Aujourd'hui SCA est un ensemble de spécifications gérées par le consortium OASIS, qui a pour but de le standardiser. L'objectif de SCA est de simplifier l'écriture d'applications à base de composants sans se soucier du langage d'implantation du composant. Il existe plusieurs implantations de la spécification SCA telles que Websphere proposé par IBM [60], Aqualogic de BEA Systems [61] ou encore Tuscany d'Apache Software Foundation [62].

Une application est définie comme un ensemble de composants logiciels qui interagissent. Un composant SCA implante le logique métier dans un langage de programmation quelconque. Il est composé de trois parties :

- les **services** qui sont les fonctionnalités du logique métier rendu accessibles aux autres composants. La description des services est faite selon la technologie utilisée pour réaliser la logique métier. Par exemple, on a une interface Java pour un composant implanté en Java ; pour des procédés BPEL, un fichier WSDL donne la description du service.
- les **références** qui sont les fonctionnalités requises pour le bon fonctionnement du composant. Ce sont des dépendances à d'autres services fournis par d'autres composants.
- les **propriétés** qui sont des informations de configuration nécessaires lors de l'instanciation du composant.

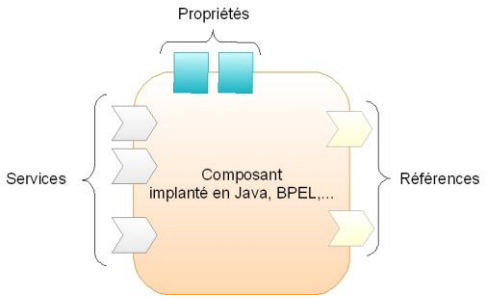


Figure 2.20. Structure d'un composant SCA

La spécification SCA a été proposée pour créer des applications ; ces dernières sont alors le résultat de l'assemblage de composants SCA. Cet assemblage suit le modèle de la composition structurelle qui permet d'obtenir une fonctionnalité plus complexe à partir des services fournis par différents composants. Un tel assemblage est un composite qui contient des composants et des liaisons qui représentent les liens entre composants. La nature des composants dans un composite peut être ou non hétérogène. La composition hiérarchique est aussi possible : un composite peut être lui-même utilisé dans un autre composite.

[38]OSOA : Open Service Oriented Architecture

L'architecture obtenue par assemblage de composants est donnée dans un langage de description appelé SCDL[39]. Ce langage permet de décrire les composites à différents niveaux de granularité et d'abstraction. Un composite est une unité d'encapsulation et de visibilité. En effet, les composants qui sont à l'intérieur d'un composite, ne sont pas visibles de l'extérieur du composite et ne peuvent pas être liés à des services provenant de l'extérieur. Tout comme pour un composant, un composite peut exposer ses services, ses références et ses propriétés. Ces informations sont issues des composants du composite. La Figure 2.21 est une illustration du modèle de composition de SCA.

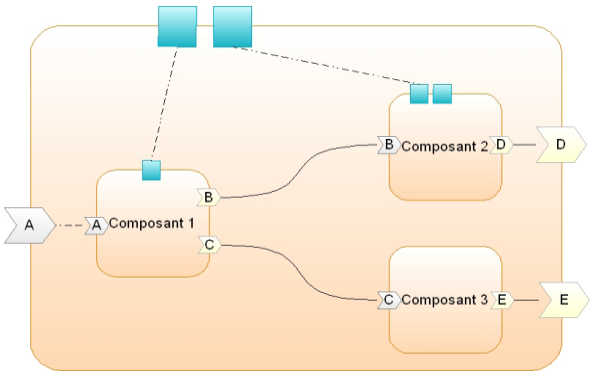


Figure 2.21. Exemple de composition de composants SCA

En conclusion, la spécification SCA se base sur un ensemble de concepts intéressants pour la réalisation d'applications à base de composants. La composition est définie clairement dans un langage de description d'architecture qui sépare les services des liaisons. Un autre point fort de SCA est qu'il est possible de créer des compositions structurelles hiérarchiques en imbriquant des composants composites implantés dans n'importe quel langage. Cependant, la composition n'est faite qu'au niveau conceptuel et non au niveau exécution, par conséquent le dynamisme et la reconfiguration à l'exécution ne sont pas traités.

6. Conclusion

L'approche à service définit un nouveau paradigme de programmation plaçant le service au centre de l'approche. Un service est défini comme une entité logicielle rendant accessible certaine de ces fonctionnalités au monde extérieur. Cette approche est basée sur l'utilisation de trois acteurs : le fournisseur de service fournissant les fonctionnalités, l'annuaire qui enregistre les services disponibles à un moment donné, et le consommateur qui utilisent les fonctionnalités. Ce chapitre a permis de mettre en évidence la propriété principale de l'approche à service à savoir le faible couplage entre les entités qui compose une application. De plus, l'approche à service dynamique a permis d'ajouter à cette propriété de faible couplage, la prise en compte de l'évolution indépendante de certaines parties de l'application. L'étude des différentes mises en œuvre de l'approche à service a néanmoins montré qu'il était particulièrement complexe de développer ce type d'application. L'approche à composant orienté service qui vise à mettre en œuvre l'approche à service en utilisant les concepts des composants tend à simplifier la mise en œuvre de ce genre d'application.

[39] SCDL: Service Component Definition Language

La gestion de la dynamicité et de l'hétérogénéité est d'une telle complexité que la mise en œuvre d'applications robustes et répondant aux besoins est un véritable défi. Dans de nombreux cas, les possibilités de l'approche ne sont en fait pas utilisées faute de pouvoir ou de savoir les maîtriser. Il apparaît clairement à nos yeux qu'un cadre de développement de telles applications est aujourd'hui nécessaire.

Chapitre 3 Lignes de produits

Ce chapitre présente un état de l'art sur l'ingénierie des lignes de produits, qui est un paradigme de développement de systèmes logiciels issu des approches de réutilisation logicielle. L'objectif des lignes de produits est de réduire l'effort et le coût de développement ainsi que le temps de mise sur le marché de produits à forte composante logicielle. Il s'agit également d'améliorer la qualité générale des produits fournis.

Ce paradigme de développement se focalise sur la production de familles de produits logiciels plutôt que sur la fourniture d'un produit unique. La première section commence par donner une définition d'une ligne de produits logiciels. Elle détaille ensuite le développement des lignes de produits logiciels qui est structuré en deux processus – l'ingénierie du domaine et l'ingénierie des applications.

La conception d'une architecture commune, partagée par un ensemble de produits logiciels est un aspect fondamental dans les lignes de produits. Une telle architecture permet de se focaliser sur le développement de familles de produits logiciels dans le cadre d'un domaine spécifique. Cette architecture est appelée architecture de la ligne de produits, ou *architecture de référence*. Elle augmente non seulement le niveau d'abstraction de la conception de systèmes mais aussi la granularité de la réutilisation. Elle sert de base à la conception et au développement d'une ligne de produits. Nous présentons donc dans la section 3 un ensemble de thèmes de recherche concernant l'architecture logicielle, tels que les langages de description d'architectures et des méthodologies de développement de l'architecture de référence. La notion de variabilité, qui est au cœur des lignes de produits, est également présentée dans ce chapitre. Cette notion a pour but de permettre des variations au niveau des produits individuels appartenant à une famille, à partir de l'architecture de référence et d'autres éléments réutilisables.

La section 4 étudie les enjeux et les défis auxquels il est nécessaire de faire face lors du développement de lignes de produits.

1. Définitions

Nous avons parcouru dans l'introduction générale différentes technologies permettant ou visant à améliorer la réutilisation logicielle. Il apparaît que la réutilisation, pour être efficace, ne peut se reposer uniquement sur une technologie mais doit être préparée, planifiée, organisée. Les lignes de produits, précédemment introduites, visent justement la réutilisation par l'utilisation de composants et de modèle mais aussi par la mise en place de processus précis. Nous nous concentrons ici sur ce paradigme.

Dans les domaines d'ingénierie traditionnelle, la création d'une ligne de produits en utilisant un ensemble de pièces réutilisables n'est pas une idée nouvelle. La société Ford, célèbre fabricant automobile, a créé la première ligne d'assemblage d'automobiles en utilisant des pièces interchangeables en 1908 [16]. Cette innovation a permis à Ford d'atteindre des objectifs économiques ambitieux : un coût de production bas, une productivité importante et des voitures de qualité. Depuis les années 1990, les lignes de produits sont fortement considérées en informatique, aussi bien par les académiques que les industriels tels que Hewlett-Packard, Nokia ou Motorola [63]. Ceci peut être considéré comme une des transitions les plus marquantes parmi les paradigmes du développement logiciel depuis les langages de haut niveau [15].

Considérons tout d'abord la définition proposée par Krueger :

> *"A software product line refers to engineering techniques for creating a portfolio of similar software systems from a shared set of software assets using a common means of production." [64]*

Cette définition présente une ligne de produits logiciels comme un ensemble de techniques d'ingénierie permettant la création de systèmes logiciels similaires s'appuyant sur un ensemble d'artéfacts logiciels partagés. Dans cette définition, l'auteur introduit de façon indirecte le concept de la portée (*portfolio*) d'une ligne de produits. La portée est une notion fondamentale : elle permet de déterminer les produits qui sont inclus dans une ligne de produits et ceux qui n'y appartiennent pas, au moment courant ou dans le futur. Cependant, cette définition ne précise pas la manière de produire ces systèmes similaires.

Une seconde définition est apportée par Lai et Weiss :

> *"A software product line is a family of products designed to take advantage of their common aspects and predicted variabilities" [65]*

Cette définition met en avant la finalité d'une ligne de produits, qui est bien de « profiter » d'un ensemble d'artéfacts communs réutilisables. Dans leur proposition, Lai et Weiss mentionnent explicitement la notion de variabilité dans une ligne de produits. Ils expriment le fait que certains aspects sont effectivement communs mais qu'il existe également des aspects variables au sein des différents artéfacts réutilisables. Enfin, cette définition souligne que les aspects communs et variables sont utilisés de façon explicite.

Jan Bosch propose, quant à lui, la définition suivante :

> *"A software product line consists of a product line architecture and a set of reusable components that are designed for incorporation into the product*

line architecture. In addition, the product consists of the software products that are developed using the mentioned reusable assets." [66]

Dans cette définition, Jan Bosch met en évidence l'importance de la notion d'architecture logicielle dans une ligne de produits. La ligne de produit est ainsi définie par cette architecture, appelée *architecture de référence*, et par un ensemble de composants réutilisables au sein de cette architecture. L'architecture de référence fournit le cadre de développement des composants réutilisable et garantit leur incorporation appropriée. Elle apparaît ainsi comme un guide d'assemblage des artéfacts réutilisables pendant la phase de conception du développement d'un produit. De ce fait, elle permet de conduire à la création de produits individuels par la réutilisation d'artéfacts prévus à cet effet. Cependant, cette définition n'évoque pas le concept de variabilité au sein d'une ligne de produits.

La dernière définition que nous évoquons provient du SEI (*Software Engineering Institute*). Le SEI contribue fortement à la définition et à la promotion des lignes de produits en logiciel. Le SEI organise de nombreuses formations, ainsi que des séminaires où des industriels présentent des retours d'expérience argumentés. Ces présentations confirment régulièrement les bénéfices apportés par les lignes de produits logiciels. La définition proposée par le SEI est la suivante [63]:

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [67]

Cette définition s'applique aux systèmes à forte composante logicielle. Ceci fait référence à des systèmes incluant des aspects matériels mais où le logiciel a un impact majeur sur toutes les étapes du développement, telles que la conception, la mise en œuvre, le déploiement et l'évolution [68]. Cette définition précise que les systèmes répondent aux besoins d'un marché existant et s'inscrivent dans une vision globale. Ceci provient du fait qu'une ligne de produit est coûteuse et qu'elle ne peut être mise en place que pour répondre à un réel enjeu économique. Enfin, cette définition souligne le fait que les produits individuels sont réalisés de manière prescrite, en utilisant des artéfacts communs. La manière exacte de construire une famille de produits reste encore floue dans cette définition, qui se focalise plutôt sur la perspective économique.

En conclusion, nous considérons une ligne de produits comme un ensemble de techniques d'ingénierie permettant la construction de façon prescrite de produits individuels au sein d'une famille basée sur la réutilisation d'artéfacts partagée par la famille de produits. Une ligne de produits se concentre sur marché existant, bien défini. La personnalisation des membres de la famille peut s'effectuer grâce à la prise en compte de variabilités à différents niveaux d'abstraction et tout au long du cycle de vie de développement.

L'objectif des lignes de produits est de promouvoir une réutilisation logicielle effective et systématique. Pour ce faire, la mise en œuvre d'une ligne de produits est divisée en deux processus de développement parallèles. Le premier, appelé ingénierie du domaine ou « *Domain Engineering* », a pour but le développement des artéfacts réutilisables. C'est un processus de développement *pour la réutilisation* en ce sens qu'il prépare les éléments qui seront plus tard réutilisés. Le second processus, appelé ingénierie des applications ou « *Application Engineering* », a pour but de construire des nouvelles applications spécifiques. C'est un processus de développement *par la réutilisation* puisqu'il se base sur

les artéfacts logiciels réutilisables développés lors de l'ingénierie du domaine. La figure 3.1 illustre précisément ces deux processus de développement en précisant certaines de leurs activités.

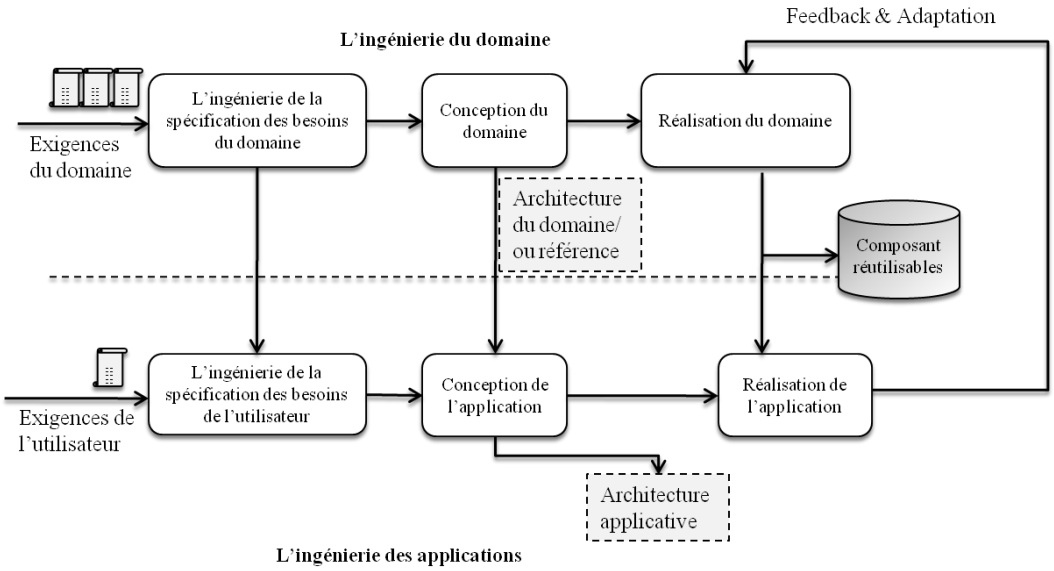


Figure 3.1. Modèle des deux cycles de vie de l'ingénierie de lignes de produits

Le but de l'ingénierie du domaine est de mettre en place une plate-forme de réutilisation. Celle-ci comprend tous les artéfacts logiciels pouvant être partagés par des produits appartenant à la ligne de produits. Les artefacts peuvent être de différentes natures : ils couvrent en fait toutes les activités d'une production logicielle allant de l'analyse des besoins du domaine, de la conception jusqu'à l'implémentation et les phases de test. Ils peuvent correspondre à des documents (une spécification textuelle d'exigences), des modèles (une architecture spécifiée dans un outil adapté), du code, etc. Ils doivent tous être testés et leur fiabilité doit être avérée. C'est ainsi que l'ingénierie du domaine peut être la base de la production de produits de haute qualité. Enfin, la plate-forme comprend également des stratégies et des procédures de production systématique des produits individuels.

Le but de l'ingénierie des applications est de développer les produits ou systèmes finaux s'appuyant sur la plate-forme établie lors de l'ingénierie du domaine. Dans cet esprit, la construction d'un produit logiciel spécifique est un processus de configuration et d'assemblage d'artéfacts réutilisables. Les artéfacts réutilisables possèdent en effet des parties variables qui doivent être configurées lors d'une réutilisation. Cela permet, par exemple, de ne retenir que certaines parties des exigences d'un domaine ou d'adapter l'architecture de référence aux besoins spécifiques d'un produit. La gestion des variabilités (les différences entre produits) doit être gérée et planifiée : l'ingénierie du domaine fournit pour cela des outils, comme indiqué précédemment. De nombreux mécanismes ont été proposés [69, 66] pour utiliser des points de variation explicites de manière à configurer ou paramétrer des artéfacts lors du processus de la dérivation de produits.

Lors du processus d'ingénierie des applications, plus de 90% des artéfacts constituant un produit peut être obtenu par configuration/assemblage d'artéfacts réutilisables. Ainsi, un objectif raisonnable est de viser la production de moins de 10% de l'ensemble des artéfacts spécifiques pour un produit. Ces nouveaux développements sont évalués et peuvent rejoindre la plate-forme de réutilisation s'ils sont jugés

suffisamment génériques au sein de la ligne de produits. Cette activité de développement permet de réunir les deux processus du développement de lignes de produits. Dans le cas idéal, les deux processus ont un couplage faible et peuvent être synchronisés à l'aide de la plate-forme [15].

Les études du SEI montrent que la gestion explicite de lignes de produits permet de réutiliser la quantité prévue d'artéfacts logiciels et, ainsi, d'obtenir de réels gains économiques. Plus précisément, des améliorations sont obtenues en termes de réduction du temps de sortie sur le marché, de l'effort et du coût de développement et d'amélioration de la qualité des systèmes. Les lignes de produits permettent en particulier de s'adapter rapidement à des marchés concurrentiels qui ont des fréquences élevées de sortie de nouveaux produits (comme les téléphones portables par exemple). L'organisation des entreprises bénéficie également de la mise en place d'une ligne de produits : il apparaît que des structures spécifiques sont créées et facilitent la discussion entre les équipes de conception, de développement, d'implantation et les équipes en prise avec le business. D'un point de vue technique, la prise en compte de la notion de variabilité tout au long du cycle de vie différencie nettement l'approche des lignes de produits des autres approches de réutilisation. Cette caractéristique permet de satisfaire plus facilement les besoins de personnalisation des clients finaux. Les différences possibles entre produits sont en effet explicites, ainsi que les liens entre la spécification des besoins d'un produit et les impacts sur les éléments réutilisables. L'architecture de référence est conçue pour structurer les points communs et identifier des points de variations. Cela permet d'augmenter le niveau d'abstraction lors de l'ingénierie applicative : les artéfacts réutilisables sont assemblés et configurés au niveau des points de variations.

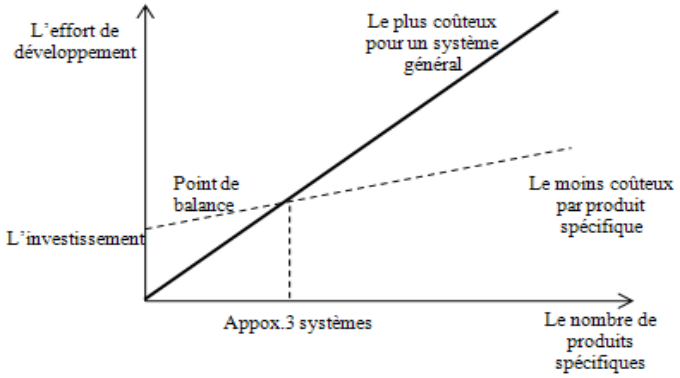


Figure 3.2. Aspects économiques de l'ingénierie de lignes de produits (adapté de [15])

Cependant, il est clair que la mise en place d'une ligne de produits demande un investissement initial important. Développer une ligne de produits est complexe pour de multiples raisons. Tout d'abord, l'identification des points de variabilité sur les différents artéfacts logiciels est difficile et coûteux. Cela demande une analyse minutieuse des différentes déclinaisons possibles d'un artéfact et de maîtriser les techniques de modélisation et de codage permettant d'introduire un certain niveau de variabilité. Cela ne peut être réalisé que par des ingénieurs et architectes ayant des connaissances très larges. Par ailleurs, la phase de maintenance est rendue plus difficile. Il ne s'agit plus de gérer les versions successives d'un même produit mais, maintenant, de gérer de façon conjointe l'évolution de plusieurs produits similaires et de leurs artéfacts réutilisables. Il faut ainsi gérer l'évolution des éléments du domaine ainsi que les liens de dépendances avec les différents produits déjà construits. La figure 3.2 illustre l'aspect économique lié aux lignes de produits. Il apparaît que le développement des premiers produits est très coûteux et que l'investissement n'est amorti qu'au-delà de trois systèmes.

2. L'ingénierie des domaines

2.1. Définitions

Comme nous l'avons indiqué précédemment, l'objectif principal de l'ingénierie du domaine est de produire des artéfacts réutilisables et de fournir les moyens permettant leur utilisation effective pour construire un nouveau produit au sein d'une ligne de produits. L'ingénierie du domaine cherche ainsi à identifier, formaliser, préparer tous les artéfacts logiciels d'une ligne de produit pouvant être réutilisés lors des phases d'analyse, de conception, d'implémentation, de test, de maintenance, etc. Chaque phase produit, bien sûr, des artéfacts de nature différente, mais pareillement caractérisés par des invariants et des aspects modulables. On peut voir ces artéfacts comme le résultat du transfert de connaissances avancées sur un domaine (celui de la ligne de produits) dans un monde informatique, sous une forme configurable.

Le SEI définit les artéfacts réutilisables comme suit :

> *"An artifact or resource that is used in the production of more than one product in a software product line. Core assets often include, but are not limited to, the architecture, requirements statements, reference architecture, and reusable software components. More documents can be also included, such as domain models, documentation and specifications, performance models, schedules, budgets estimation, test plans, test cases, work plans, process descriptions." [70]*

Chaque artéfact réutilisable est lié à une activité précise du développement de l'ingénierie du domaine. Il apparaît clairement dans cette définition que les artéfacts réutilisables de lignes de produits ne limitent pas seulement à du code, comme un composant logiciel par exemple. Un artéfact réutilisable peut prendre la forme d'un document, décrivant des exigences en langage naturel par exemple, d'un modèle, décrivant par exemple des concepts du domaine (appelés le vocabulaire du domaine) et leurs associations, d'une conception, comme l'architecture de référence de la ligne de produits. Un artéfact peut également correspondre à un plan de production, un plan de test, une description de processus, etc.

La figure 3.3 présente les principales activités du processus d'ingénierie du domaine ainsi que leurs liens en termes d'entrées requises et de résultats fournis. Cette figure met également en évidence la dépendance entre les processus domaine et applicatif. Le développement d'un nouveau produit requiert parfois l'implantation d'un nouvel artéfact pour répondre à certains besoins spécifiques. Dans ce cas, ce nouvel artéfact peut intégrer la plate-forme de la ligne de produits. Il s'agit d'un *feedback* du processus applicatif vers le processus de gestion du domaine.

Nous détaillons dans les sections suivantes les tâches principales, à savoir l'analyse du domaine, la conception du domaine et l'implantation du domaine.

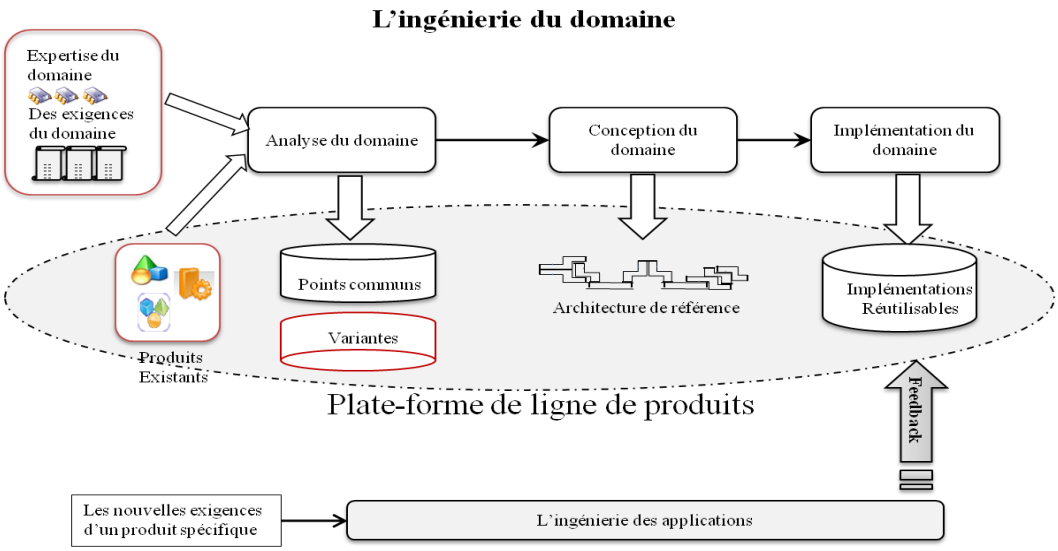


Figure 3.3. Le processus de développement de l'ingénierie des domaines

2.2. Analyse du domaine métier

De façon générale, l'analyse du domaine métier est une activité visant à définir un domaine et à capturer et formaliser les connaissances dans ce domaine. Il s'agit ainsi d'une activité de transformation de connaissances métier vers des représentations informatiques. Cette activité inclut la définition du domaine en lui-même, la collecte d'informations, le tri et le regroupement des informations, la modélisation des informations. La modélisation produit une description normalisée des connaissances du domaine et, si possible, un ensemble de procédures systématiques afin de les implémenter [71]. L'analyse du domaine s'appuie grandement sur les éléments existants dans le domaine, tels que des matériels, des logiciels ou des documents. Les produits existants dans un domaine sont donc clairement des entrées pour l'analyse. Les expériences et compétences des experts du domaine (techniques et métier) représentent également des ressources implicites qui impactent fortement le résultat de cette activité. L'étude des exigences du domaine est une partie fondamentale de l'analyse. Elle doit identifier les fonctionnalités du domaine mais aussi les contraintes, les règles métier, les niveaux de qualité attendus. Les exigences sont généralement considérées comme des informations acquises.

Comme nous l'avons indiqué précédemment, certains aspects sont partagés par tous les produits, c'est-à-dire qu'ils sont nécessaires à tous les produits appartenant au portefeuille de la ligne de produits. Ces aspects sont les points communs, ou *commonalities* en anglais. D'autres aspects sont variables et servent à respecter certaines exigences spécifiques. Ce sont les variations, ou *variabilities*. Des aspects communs et variables apparaissent naturellement dans les différents artéfacts réutilisables. Il convient donc de fournir des mécanismes de configuration des artéfacts réutilisables pour agir sur les points de variations. La capture des points de variation n'est pas le but final. Il faut en effet identifier et classifier les liaisons possibles entre les variantes de façon à complètement illustrer les solutions alternatives dans le domaine. Cela conduit à la définition d'ensemble de variantes en relation. Chaque ensemble est un point de variation (défini par la suite dans 2.2.2). Au niveau du domaine, les différences entre produits d'une même ligne s'expriment à l'aide des points de variation au niveau du modèle de spécification des

exigences. Cela permet d'expliciter les différences entre produits à un haut niveau d'abstraction, souvent dans un langage proche des considérations métier.

Une étape fondamentale de l'analyse du domaine est de déterminer le périmètre fonctionnel ; il s'agit de la portée de la ligne de produits [72]. Celle-ci définit explicitement l'ensemble des produits à étudier et implicitement les futurs produits qui pourront entrer dans la ligne de produits. Au cours de cette étape, il faut trouver un bon compromis entre coût, effort et qualité de développement de l'ingénierie des domaines et de l'ingénierie des applications. Plus précisément, l'activité déterminant la portée d'une ligne de produits est réalisée à trois niveaux différents en termes de la granularité de la réutilisation logicielle [73] :

- Déterminer le domaine : le but ici est d'analyser systématiquement les besoins (plutôt les fonctions) sur un segment de marché. Cela permet de guider les décisions d'investissement sur une ligne de produits dans le cadre du marché analysé.
- Déterminer le portefeuille de la ligne de produits : le but ici est d'identifier les produits à considérer dans la ligne de produits et leurs fonctionnalités supportées par la plate-forme de la ligne de produits. Ceci dépend en partie des besoins du marché.
- Déterminer le portefeuille des artéfacts réutilisables : le but ici est de sélectionner les artéfacts abstraits (comme la spécification des besoins) et concrets (comme des composants implémentés) servant de base à la ligne de produits dans un domaine spécifique.

La définition de la portée d'une ligne de produits influence plusieurs d'activités au sein des deux processus du développement de la ligne de produits. Une mauvaise définition de la portée conduira à l'échec de la construction de la ligne de produits. Les deux cas extrêmes suivants doivent être évités :

- Définition d'un cadre trop limité : de nombreux points communs sont définis dans le troisième niveau (le portefeuille des artéfacts réutilisables). Par contre, peu de variabilités sont identifiées parmi les produits de la famille. Les artéfacts réutilisables (comme l'architecture de référence) manquent alors de flexibilité afin d'adapter des besoins spécifiques croissants au cours de nouveaux développements.
- Définition d'un cadre trop large : de nombreux points variables sont déterminés dans l'ensemble des artéfacts réutilisables. Les produits inclus dans le portefeuille de la ligne de produits sont très variés. La construction d'un nouveau produit spécifique est donc très difficile car elle ne repose pas sur suffisamment de points communs. Les développeurs ne peuvent profiter des avantages d'une réutilisation planifiée.

En résumé, le résultat de la définition et de la modélisation du domaine est formalisé et documenté comme une spécification contenant les éléments fondamentaux ci-dessous :

- Le vocabulaire du domaine : il définit les notions du domaine exprimant le comportement fonctionnel et/ou les attributs non fonctionnels ainsi que les contraintes.
- L'ensemble des produits concernés du domaine : la portée de la ligne de produits identifie le périmètre du domaine, les produits et fonctionnalités qui appartiennent au domaine et ceux qui n'y appartiennent pas, le portefeuille des artéfacts réutilisables.
- L'identification des points communs et variables : les points communs permettent de capitaliser sur des développements alors que les variantes permettent la configuration pour différencier les applications/produits spécifiques.

L'analyse d'un domaine est une activité plus complexe que la « simple » analyse des besoins lors du développement d'une application individuelle. Cette différence n'est pas toujours perceptible à première vue, car le périmètre du domaine dans le cadre de l'ingénierie des domaines est en général flou au démarrage et, souvent, trop limité. L'analyse du domaine est ainsi considérée comme un processus progressif et relativement non structuré d'apprentissage et d'acquisition d'expérience qui permet au fur et à mesure de clarifier les concepts du domaine jusqu'à ce qu'ils deviennent suffisamment concrets pour pouvoir être transformés en artéfacts réutilisables. De nombreuses méthodes ont été proposées pour guider l'analyse de domaine, notamment au sein de méthodes telles que FODA (*Feature-Oriented Domain Analysis*) [71], SCV (*Scope, Commonality and Variability*) ou FAST (*Family-Oriented Abstraction, Specification, and Translation*) [74]. Ces méthodes servent à découvrir et exploiter des points communs et variables de façon systématique lors de la modélisation du domaine.

2.3. Conception de l'architecture de référence

Comme illustré par la figure 3.3, la conception domaine apparaît à la suite de l'analyse du domaine et repose sur les résultats produits. Son but est de concevoir les éléments logiciels qui feront partie de la plate-forme de la ligne de produits, et qui seront donc réutilisables. Deux types d'éléments particulièrement importants sont définis à ce niveau : l'architecture de référence et les composants logiciels pouvant être réutilisés au sein de la ligne de produits. Ces composants sont en général configurables ou extensibles, ce qui est une façon d'apporter des points de variation.

L'architecture de référence qui définit de façon abstraite les composants structurels des produits et leurs relations. Nous l'avons déjà indiqué : l'architecture de référence est un élément clé dans une plate-forme de ligne de produits. Généralement, les experts du domaine se focalisent en premier sur ce point. Ils s'appuient sur l'ensemble des modules communs identifiés lors de l'analyse du domaine pour obtenir une première structuration de l'architecture. Cette structuration met en évidence les composants importants et leurs relations. Ensuite, l'architecture est successivement affinée de façon à faire apparaître des points de variation explicites. La création d'un point de variabilité est une décision importante : rappelons qu'un trop grand nombre de points de variabilité conduit à une ligne de produit inefficace car demandant trop d'effort lors de la phase d'ingénierie applicative. Dans certains cas, l'architecture de référence permet d'automatiser le développement des produits finaux en définissant de façon systématique comment les artéfacts réutilisables doivent être assemblés. Toutes les architectures des produits spécifiques dans le portefeuille de produits doivent être « conformes » à l'architecture de référence de la ligne de produits.

De façon générale, la *variabilité* [75] est une capacité des artéfacts logiciels leur permettant d'être étendus, modifiés, personnalisés ou configurés sur mesure pour répondre à des besoins particuliers. Au niveau d'une architecture de référence, un *point de variation* [75] identifie précisément en endroit dans l'architecture où des changements/variations peuvent apparaître dans le temps. Un point de variation concrétise une décision de conception retardée. Afin d'obtenir un produit particulier dans une ligne de produits, il est nécessaire de prendre cette décision de conception. Un point de variation peut être caractérisé par un ou plusieurs choix, appelés des *variantes*. Les variantes représentent des choix architecturaux possibles sur le point de variation. On peut envisager deux situations

- Le point de variation est « ouvert. » Dans ce cas, il est possible lors de l'ingénierie applicative d'ajouter une nouvelle variante ou de modifier des variantes existantes.

- Le point de variation est « fermé. » Dans ce cas, il n'est pas possible de sortir des choix proposés par le point de variation.

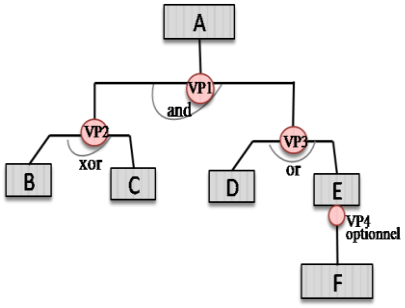


Figure 3.4. Le schéma des dépendances des points de variation

Les points de variation apparaissent souvent sur les liaisons entre composants. Ces liaisons peuvent mettre en jeu deux ou plusieurs composants. Dans ce cas, les points de variation peuvent prendre les formes suivantes :

- Optionnel : la liaison architecturale ainsi définie n'est pas indispensable pour le système. Par exemple, l'utilisation d'un composant spécifique permettant l'envoi de messages sécurisés peut être optionnelle.
- Alternatif ou « xor » : la liaison architecturale dans ce cas est nécessaire. Le système spécifie un ensemble de composants possibles à relier. Par exemple, on peut spécifier qu'un composant de visualisation est nécessaire mais qu'il peut prendre la forme d'une télévision ou d'un téléphone portable. Les deux équipements électroniques fournissent un ensemble de fonctionnalités informatiques communes.
- Multiples ou « or » : la liaison architecturale dans ce cas est également nécessaire. Le système spécifie un ensemble de composants possibles à relier et plusieurs variantes peuvent être sélectionnées à la fois. Par exemple, un système peut inclure un composant de messagerie électronique, un composant de traitement de SMS, un composant de liaison à un fax comme mode de communication entre des différents systèmes ou applications logicielles.

Lorsqu'une variante est choisie sur un point de variation, nous dirons qu'une décision de conception est prise à ce point de variation. Dans la grande majorité des cas, cette décision est prise lors de la conception d'un produit et n'est pas remise en cause ensuite, notamment durant à l'exécution. Un des apports de cette thèse est justement de retarder la prise de décision jusqu'à l'exécution.

La résolution d'un point de variation n'est pas toujours indépendante des autres points de variation de l'architecture. En effet, des dépendances entre points de variabilité peuvent exister. En particulier [15], on peut rencontrer les cas suivants :

- La sélection d'une variante donnée à un point de variation entraîne la sélection obligatoire d'une variante spécifique sur un autre point de variation (peut être pour le même point de variation)
- La sélection d'une variante donnée à un point de variation entraîne l'impossibilité de choisir certaines variantes sur un autre point de variation (peut être pour le même point de variation)

L'architecture de référence de la ligne de produits a ainsi pour but d'intégrer explicitement des points de variations, caractérisés par d'éventuelles variantes, et des liens entre ces points de variation. La gestion de cette variabilité est une problématique plus complexe. Le mécanisme de gestion de variabilités de lignes de produits demande de considérer de multiples aspects (*i.e.* la conception des variabilités, l'implantation de variabilités ainsi que la maintenance et l'évolution de la plate-forme de l'ingénierie des domaines). L'introduction d'un mécanisme de gestion des variabilités représente une forte différence entre les approches de lignes de produits logiciels et les autres approches de réutilisations conventionnelles. Nous reviendrons plus en détail sur ce point dans la section 3.4 - Architecture de référence en terme de la gestion de variabilités.

2.4. Conception et implantation des composants

Dès que la description de l'architecture est créée, des développeurs peuvent commencer à implémenter les composants, principaux constituants de l'architecture. Comme indiqué, il est important que ces composants soit adaptables à différents contextes d'utilisation. Il existe plusieurs façons de créer, lors de l'ingénierie du domaine, des composants adaptables ou extensibles. Les plus connues sont les techniques suivantes [69, 66] :

- Utilisation de l'héritage en orienté objet. Un composant peut être implanté sous la forme d'un *framework* orienté objet, permettant des modifications ou des ajouts de comportement afin de s'adapter à des besoins spécifiques. Comme pour les architectures de référence, des variantes peuvent être mises à disposition des développeurs et, dans ce cas, les points d'extension peuvent être ouverts ou fermés selon la latitude offerte au développeur.
- Utilisation d'une interface de configuration. Un composant peut, dans sa logique interne, prévoir des comportements différents et fournir une interface permettant de choisir le comportement souhaité. On peut également mettre à disposition des développeurs un fichier de configuration pour décrire les variantes souhaitées et, éventuellement, les liens entre les variations. Un mécanisme similaire est la paramétrisation lors des appels des procédures ou des fonctions ;
- Configuration au moment de la compilation. Les compilateurs peuvent fournir des mécanismes pour adapter un composant lors de la compilation en utilisant un pré-processeur et des macros. Par exemple, un « *makefiles* » peut compiler un composant dans les plusieurs variantes binaires, ou sélectionner différents binaires pour résoudre des dépendances lors de la création d'application exécutable.
- Utilisation d'un générateur de code. Le principe d'un générateur de code est de prendre des spécifications en entrée et de générer un module logiciel, un composant dans ce cas, répondant aux spécifications. Le langage de spécification est plus ou moins ouvert, laissant une place plus ou moins grande aux variations dans le comportement du composant généré. Dans certains cas, le générateur de code devient très compliqué et son développement ainsi que sa maintenance sont donc très coûteux.
- Utilisation de mécanismes d'extension. L'adaptation d'un composant peut être définie par addition dynamique de code comme c'est le cas, par exemple, avec les mécanismes de type « *plug-ins* ». Les *plug-ins* sont parfois communs, ou peuvent aussi servir à une application spécifique (il s'agit des variantes).

2.5. Conclusion

L'activité d'ingénierie du domaine fournit ainsi les éléments réutilisables du domaine. Nous nous sommes particulièrement intéressés dans ce chapitre aux notions de composants adaptables et d'architecture de référence. Cette phase d'ingénierie du domaine peut également fournir un environnement de spécification d'architecture de produits individuels utilisable durant la phase suivante, à savoir la phase d'ingénierie applicative. Un tel environnement peut s'appuyer sur un langage spécifique au domaine, ce qui facilité grandement le travail des ingénieurs du domaine. Un tel langage porte le nom de *langage spécifique au domaine (Domain Specific Language)* [76].

En conclusion, le développement de l'ingénierie des domaines est un processus itératif qui vise à la construction des artefacts réutilisables et des outils qui vont être mis à disposition des développeurs d'applications du domaine. Il est important d'apporter de la flexibilité au niveau des artéfacts réutilisables tout en ne sacrifiant pas la facilité d'utilisation et de maintenance de ces artefacts. Il est également important de fournir des procédures, des guides, permettant d'étendre ou d'adapter les artéfacts réutilisables. Sans de tels supports, le danger est de retomber dans les travers de la réutilisation *ad-hoc*.

3. L'ingénierie des applications

L'ingénierie des applications est le processus de création et de mise à disposition des logiciels de la ligne de produits. Ce processus de développement se base sur les artéfacts réutilisables produits lors de la phase d'ingénierie du domaine. Plus précisément, la construction d'un produit logiciel spécifique est un processus d'assemblage et de configuration des artéfacts réutilisables en conformité avec l'architecture de référence. L'architecture d'un produit est une instanciation particulière de cette architecture de référence (qui intègre intentionnellement des points de variabilité) qui répond aux exigences spécifiques du produit. L'application construite s'inscrit automatiquement dans la portée de la ligne de produits.

Nous rappelons ici que ce processus est une activité de développement *par la réutilisation.* Lors de ce processus du développement, la réutilisation logicielle se situe au niveau d'abstraction architectural.

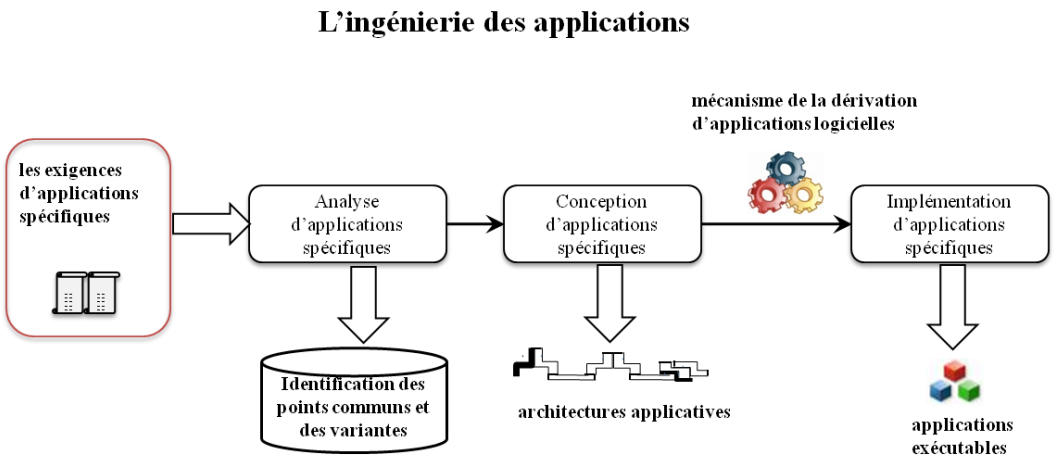


Figure 3.5. Processus de développement de l'ingénierie des applications

Comme dans l'ingénierie du domaine, la mise en œuvre de l'ingénierie des applications est divisée en trois activités principales. Ces trois activités du développement sont l'analyse des exigences spécifiques, la conception d'applications spécifiques et, finalement, l'implémentation d'applications logicielles spécifiques. La figure 3.5 illustre de processus et met en évidence les entrées requises correspondant à chaque activité du développement ainsi que le résultat apporté par chaque activité.

Ces trois activités majeures de l'ingénierie des applications sont présentées plus en détail dans la suite de ce rapport.

3.1. L'analyse des exigences d'applications

La première activité de l'ingénierie des applications est l'analyse des exigences du produit visé. Les exigences spécifiques [77] sont obtenues à partir des interactions avec les différents intervenants du développement de produits, tels que des utilisateurs finaux, les manageurs de produits, les clients ou les techniciens de maintenance. Cette étape d'analyse est essentielle et doit apparaître explicitement. C'est elle, en effet, qui met en évidence les besoins spécifiques et justifie les futurs travaux d'adaptation ou de développement.

On voit apparaître trois types d'exigences :

- Les exigences qui sont inclus dans les exigences du domaine répertoriées lors de l'ingénierie du domaine. Celles-ci sont connues et sont généralement reliés à des décisions architecturales et d'implantation des composants.
- Les exigences spécifiques qui sont prévues dans l'analyse du domaine sous forme de variabilités. Celles-ci donnent lieu à des configurations particulières de l'architecture, via les points de variabilité, et des composants réutilisables.
- Les exigences spécifiques qui n'ont pas été prévues dans l'analyse du domaine. Ces exigences posent généralement problème. Elles constituent des demandes ou des contraintes qui donnent lieu à des configurations particulières, non guidées, des artefacts réutilisables ou à de nouvelles implantations.

La différence entre les exigences prévues et les exigences inattendues doit être soigneusement étudiée. Il s'agit en effet de bien vérifier que la portée de la ligne de produits est bien respectée et de ne pas se lancer dans des développements coûteux non réutilisables. Les nouvelles exigences [77] demandent de modifier et d'étendre les fonctionnalités des artéfacts réutilisables. Ces évolutions doivent également être analysées pour décider si elles doivent intégrées la plate-forme de la ligne de produits. Il est également possible de reporter la mise en place des nouvelles exigences spécifiques à l'aide de l'implémentation de l'application sur la version prochaine.

3.2. La conception d'applications et l'implémentation

A partir du résultat de l'activité de l'analyse des exigences spécifiques, l'architecture du produit visé est dérivée de l'architecture de référence conçue lors de l'ingénierie du domaine. Les points de variation définis et planifiés dans l'architecture de référence sont instanciés et configurés durant cette étape du développement pour créer une architecture applicative. Celle-ci est constituée de concepts abstraits issus de l'architecture de référence. Il faut ensuite faire la correspondance avec les artéfacts d'implémentation, typiquement des artéfacts logiciels. Des outils peuvent être fournis pour faciliter et guider la gestion et la résolution des points de variation et le traitement de certains aspects des points communs. Les architectures des produits individuels se différencient à l'aide des points de variation planifiés dans l'architecture de référence. Rappelons que ces points de variation déterminent les endroits précis où des variations sont anticipées et que chaque point de variation représente une décision de conception retardée. Dès qu'une décision est prise lors du processus d'instanciation de l'architecture de référence, la variabilité correspondante est retirée.

Le moment exact où les décisions de conception retardées sont prises est variable. Ce moment, généralement appelé « *Binding Time* », peut survenir tout au long du cycle de vie du développement d'une application spécifique. Si nous reprenons la classification proposée par [78], une décision de conception peut s'effectuer :

- Au moment de la décision de réutiliser un composant donné. Lorsqu'un composant est choisi pour être intégré dans une architecture, il est de suite adapté au contexte d'utilisation.
- Au moment du développement de l'application spécifique, c'est-à-dire lors de l'assemblage des composants au sein de l'architecture.
- Au moment de l'instanciation du code source, juste avant la compilation.

- Au moment de la compilation.
- Au moment de l'emballage (ou « packaging »), c'est-à-dire lors de l'assemblage de code binaire et des exécutables ;
- Au moment de la personnalisation coté client, c'est-à-dire lors du processus d'implémentation chez un client spécifique ;
- Au moment de l'installation.
- Au moment du démarrage.
- Au moment de l'exécution.

Pour un même produit spécifique, il est possible d'utiliser plusieurs *binding times* afin de traiter les différentes décisions de conception retardées. Cela permet de prendre certaines décisions plutôt au début du cycle de vie, lors de la conception architecturale par exemple, et de prendre d'autres décisions plus tard, jusqu'au l'exécution. Cette approche permet d'intégrer les compétences des différentes personnes intervenant dans un projet. Certaines décisions, par exemple, peuvent être prises par développeur assemblant divers composants métier alors que d'autres décisions peuvent être prises par des utilisateurs finaux pouvant configurer l'application au cours de l'exécution. Lorsque l'on utilise plusieurs « *Binding Times* », le processus de développement devient très incrémental. Les décisions prises à certains points de variations deviennent les entrées d'artéfacts logiciels partiellement instanciés par rapport à la prochaine phase de production.

Il faut néanmoins reconnaitre que, dans la plupart des cas, les décisions se prennent assez tôt. Le « binding time » correspond rarement à l'exécution dans les lignes de produits actuelles.

Bien que les décisions de conception se prennent tout au long du cycle de développement, toutes les décisions doivent être conservées et tracées dans le cadre de la ligne de produits. Certaines exigences spécifiques, qui ne pas inclues dans la spécification des exigences du domaine, donnent lieu à des modifications d'artéfacts existants ou à de nouveaux développements. Ces implémentations produisent des artéfacts logiciels potentiellement réutilisables, qui seront évalués et, éventuellement, intégrés à la plate-forme de la ligne de produits. Cela correspond à la notion de « *feedback* » au sein d'une ligne de produit. Les artéfacts réutilisables communs sont donc complétés par le biais des implémentations de variantes.

4. Architecture

Nous avons mis en évidence dans la section précédente l'importance que revêt l'architecture logicielle dans une approche ligne de produit. L'architecture logicielle est d'ailleurs devenue un élément essentiel dans tous développements de systèmes logiciels. Elle apporte un niveau d'abstraction suffisamment élevé pour réunir les différents « stakeholders » d'un projet et permet d'aider efficacement à la structuration des développements. Le but de cette section est de fournir au lecteur les éléments théoriques et pratiques concernant les architectures logicielles nécessaires à la compréhension de la suite de cette thèse.

4.1. Définitions

De nombreuses définitions ont été avancées pour caractériser une architecture logicielle. L'une des premières a été apportée par Mary Shaw et David Garlan :

> *"Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns." [79]*

La définition de Shaw et Garlan présente une architecture logicielle comme la description abstraite des éléments constituant un système logiciel, de leurs interactions, des patrons de conception utilisés, ainsi que des contraintes associés.

Franck Buschmann fournit la définition suivante :

> *"Software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system." [80]*

Dans cette définition, l'architecture décrit également la structure d'un système, mais, cette fois, sous la forme explicite de sous-systèmes et de composants. La structure est définie suivant différentes vues, incluant des aspects fonctionnels et non fonctionnels – la qualité de système étant prise en compte dans l'architecture logicielle. Les caractéristiques de qualité du système peuvent être introduites sous la forme d'attributs dans les sous-systèmes ou les composants. De tels attributs peuvent être de plus présentés explicitement dans les différents points de vue de l'architecture logicielle. Ces attributs sont ainsi appelés propriétés externes dans d'autres définitions [81]. Les attributs concernant la qualité du système sont pris en compte et classés en deux catégories générales. Certains types d'attributs sont observables au cours de l'exécution de systèmes logiciels (tels que la performance, la sécurité, la disponibilité, la fonctionnalité et l'utilité). Au contraire, d'autres types d'attributs sont invisibles au cours de l'exécution de systèmes logiciels (tels que la modifiabilité, la portabilité, la réutilisabilité, l'intégrabilité et la testabilité) et s'adressent plus aux développeurs ou ingénieurs chargés de la maintenance.

Booch propose la définition suivante :

> "An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition." *[82]*

Cette définition exprime d'abord que l'architecture logicielle est un ensemble de décisions de conception importantes concernant l'organisation d'un logiciel. Plus précisément, ces décisions concernent la sélection des éléments structuraux constituant un système, leur composition, leur comportement collaboratif, les patterns ou styles qui sont mis en œuvre. Cette définition mentionne explicitement les styles d'architectures qui servent à guider la composition des éléments cités ci-dessus. Pour aller plus loin, une méthodologie de conception de système logiciel, nommée « *4+1View Model* » et développée par *Rational Software,* est proposée dans [82]. Cette méthodologie permet de développer l'architecture d'un système logiciel selon quatre points de vue, incluant la structuration logique, l'implémentation, le comportement dynamique et le déploiement. Par ailleurs, l'annotation « +1 » fait référence à une vue de cas d'utilisation qui spécifient les besoins des clients et qui regroupe les autres vues.

L'organisation IEEE fournit la définition suivante :

> *"Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." [68]*

L'organisation *IEEE* propose cette définition dans *Recommanded Practice for architectural Description of Software-Intensive Systems.* Cette définition prend en compte plusieurs aspects pour formaliser les différentes utilisations d'une architecture logicielle. D'abord, l'architecture est l'organisation fondamentale d'un système permettant de structurer un ensemble des composants (soit des éléments implémentés, soit des concepts abstraits) et de décrire la relation entre le système et l'environnement. De même, l'architecture d'un système englobe les aspects principaux servant à guider la modélisation de la conception de logiciels et gérer l'évolution du système logiciel.

Nous constatons qu'il n'existe pas un consensus à ce jour sur la définition de la notion d'architecture logicielle. Néanmoins, toutes les définitions s'accordent sur certaines notions et certaines théories fondamentales acceptées communément par les architectes. L'architecture logicielle présente une vision structurelle de systèmes logiciels sous la forme d'assemblage des composants et de leurs connecteurs. Les composants encapsulent les fonctionnalités cohérentes et les connecteurs réalisent les interactions entre les composants. De même, chaque définition admet la séparation au niveau de l'architecture de différentes préoccupations liées à divers aspects du développement logiciel. Enfin, l'architecture doit inclure les contraintes et principes sous-jacents à sa construction de façon, entre autres choses, à permettre son évolution.

4.2.ADLs

Une architecture peut être représentée à l'aide d'un ADL, pour *Architecture Description Languages* [83]. Celui-ci peut être informel (utilisation de schémas par exemple), semi-formel (en UML par exemple) ou formel. La description d'une architecture par un ADL a pour but d'améliorer la compréhension et la conception des systèmes complexes, favoriser leur évolution et leur réutilisation, procéder à diverses analyses, faciliter la construction et le déploiement, ou encore assister la gestion de configuration d'applications logicielles. Un langage de description d'architecture fournit les éléments pour représenter une ou plusieurs vues architecturale se focalisant sur une préoccupation particulière. De plus, la description d'une architecture à l'aide d'un ADL peut être compréhensible par une machine, ou transformable en un format compréhensible. Ce format peut être analysé et utilisé afin d'automatiser l'implémentation de certains aspects des systèmes logiciels.

Tous les langages doivent permettre de spécifier les concepts de base de l'architecture tels que, typiquement, les composants du système, leurs interfaces, les connecteurs, et la configuration architecturale [83]. Par ailleurs, chaque langage de description d'architecture se distingue par ses capacités de modélisation qui proviennent directement des objectifs poursuivis par cet ADL [83]. Par exemple, *Aesop* [86] spécifie les architectures d'applications en se basant sur un ensemble de styles architecturaux. *SADL* [85] est un langage de description d'architecture structural. Il se concentre sur le raffinement des architectures du système sur différentes couches d'abstraction de façon formelle. Cela permet la traçabilité des variantes et des évolutions du système. *UniCon* [84] permet de générer le *glue* code afin de construire et vérifier les architectures à partir d'éléments architecturaux prédéfinis utilisant les protocoles de communication communs. *MetaH* [87] se concentre sur la conception, la vérification et la validation de l'assemblage et du déploiement de systèmes temps réel critique. Il est utilisé pour modéliser les architectures en particulier dans le domaine de la navigation et du contrôle.

Certains langages de description d'architecture permettent de décrire des interactions collaboratives entre les éléments structuraux du système à l'exécution. En d'autres termes, ils permettent de modéliser, simuler et analyser le comportement dynamique d'un système au cours de son exécution. *C2, Darwin, Rapide* et *Wright* constituent des exemples de tels ADLs. Dans ce cas là, une description d'architecture spécifiée en utilisant ce type *d'ADLs* est une spécification dynamique. ***C2*** [88] se focalise sur la description des architectures d'application répartis et dynamiques. ***Darwin*** [89] se consacre à la configuration et l'instanciation des systèmes distribués et dynamique de façon formelle. ***Rapide*** [90, 91] est un langage de la modélisation et simulation du comportement dynamique d'un système dans son architecture conceptuelle. Rapide se fonde sur le concept de *posets* (*Partial Ordered event Sets*) [92]. Il permet de vérifier des propriétés de type synchronisation, concurrence et flots de données sur des assemblages de composants. ***Wright*** [93] est pareillement utilisé comme ***Rapide*** pour modéliser et analyser formellement le comportement dynamique des systèmes concurrents. Mais, ***Wright*** se concentre plus sur la vérification de conformité de l'assemblage des éléments architecturaux et la détection d'interblocages des systèmes concurrents au cours de l'exécution. Il emploie CSP (*Communicating Sequential Process*) [94] pour décrire l'architecture du système logiciel.

4.3.Styles d'architecture

Certains langages de description d'architecture permettent également de décrire des patterns particuliers ou des styles d'architectures. Les styles architecturaux sont des techniques facilitant l'identification et la définition des éléments structuraux et des connecteurs. L'un des piliers principaux de l'architecture logicielle moderne est l'utilisation des styles architecturaux [95].

La notion de style d'architecture est définie comme suit par le SEI :

> *"A specialization of element and relation types, together with a set of constraints on how they can be used." [96]*

Cette définition exprime clairement le fait qu'un style architectural spécialise les notions d'éléments et de relations entre éléments permettant de structurer un système à un haut niveau d'abstraction, et ajoute des contraintes d'utilisation. Ces éléments sont communément exprimés sous la forme de composants, de connecteurs et de ***contraintes*** délimitant la manière dont les composants et les connecteurs interagissent les styles architecturaux permettent d'améliorer la réutilisabilité de la conception logicielle, car chaque style architectural synthétise un ensemble de décisions de conception. Ils sont formulés et développés de façon à réutiliser des conceptions éprouvées et, ainsi, à éviter des échecs apparus dans d'anciens travaux. Un patron de conception fournit un ensemble de modèles décrivant de façon plus ou moins formelle les solutions correspondant à un ensemble de problèmes sous un contexte donné. Les styles architecturaux ont été utilisés avec succès depuis plusieurs années. Certains styles architecturaux sont aujourd'hui communs et très utilisés[80]. Par exemple, le style architectural « *Pipe-and-Filter* » est utilisé dans le cas où les systèmes logiciels demandent de nombreuses transformations de données et lorsque l'utilisation des données doit être flexible. Il facilite la mise en œuvre de tâches d'*ordonnancement, de synchronisation, de communications via des pipes.* Les styles architecturaux sont considérés comme la dénotation des langages de la modélisation des systèmes logiciels [69]. En outre, les styles architecturaux communs peuvent être classifiés simplement en fonction des types de systèmes logiciels, tels que les systèmes de flux de données, les systèmes d'appels et de retours, et les systèmes se basant sur un courtier.

Dans certains cas, des styles architecturaux se chargent plutôt de fournir un framework de solutions[95]. Ils sont décrits en utilisant les termes plus génériques comme *composants et connecteurs, boîtes noires et lignes.* Un style architectural est une spécification de la structure générique des systèmes/applications logiciels à un très haut niveau d'abstraction. De plus, il permet de décrire les contraintes des règles/conditions topologiques en termes de structures des systèmes. Ils se focalisent plus sur la présentation d'une topologie des éléments architecturaux sans informations concernant la logique du domaine métier. Dans certains cas, le style architectural à l'aide de l'outillage associé permet d'automatiser une partie de la mise en œuvre d'applications logicielles. Par exemple, J2EE[40] [97], CORBA [50] sont deux styles architecturaux à composants largement utilisés. J2EE fournit un framework d'intergiciels supportant l'implémentation d'applications d'entreprises. CORBA se focalise plus sur l'interopérabilité des éléments d'applications se basant sur l'architecture orientée objets. Le framework J2EE ou le framework CORBA permettent aux développeurs de se détacher des détails du développement

[40] J2EE: Java Platform Enterprise Edition

des protocoles de communications et du développement des parties techniques. La mise en œuvre d'applications se conformant aux deux frameworks est semi-automatique. En autres termes, la programmation de la partie technique d'applications est générée automatiquement à l'aide des plates-formes associées. Cela améliore la productivité de développement de systèmes logiciels et leur qualité.

4.4. Architectures de référence

Une architecture de référence est avant tout une architecture logicielle. C'est ainsi un modèle, une abstraction apportant une solution satisfaisant des exigences et qui permet aux différents intervenants de communiquer de façon précise. Elle repose bien souvent sur des patrons éprouvés et peut être décrite à l'aide d'***ADLs***. Mais, une architecture de référence a un autre rôle. Elle a pour but de mettre en évidence les aspects architecturaux communs et ariables au sein d'une famille de produits.

4.4.1. Définition

Clements définit une architecture de référence de la façon suivante [70] :

> *"Description of the structural properties for building a group of related systems (i.e., product line), typically the components and their interrelationships. The inherent guidelines about the use of components must capture the means for handling required variability among the systems. (This is called a reference architecture)"*

Cette définition considère qu'une architecture de référence est la description d'un ensemble de composants et de leurs relations permettant de construire des systèmes logiciels appartenant à une ligne de produits. De plus, l'architecture de référence doit contenir les informations nécessaires pour gérer la variabilité au sein des systèmes. Andersson adopte un point de vue similaire [98]. Il met en évidence que l'architecture de référence comporte un processus d'instanciation d'une nouvelle application particulière. Ce processus repose sur l'architecture de référence via les points communs et des points de variabilités.

On peut ainsi considérer qu'une architecture de référence est la généralisation de toutes les architectures de produits individuels dans une ligne de produits. Plusieurs problématiques peuvent alors être distinguées : 1) comment représenter les points communs ; 2) où et quand des variations peuvent être apportées pour traiter les cas particuliers ; 3) comment les variations peuvent être présentées de façon explicite et complète au sein de l'architecture de référence.

De nombreuses méthodologies se consacrant à l'analyse des points communs sont aujourd'hui proposées. Les plus abouties sont celles se fondant sur la modélisation de *features* comme FODA [71] ainsi que ses extensions FORM [99] et RSEB [100], et celle utilisant le concept de services sous la forme du diagramme de service [101].

Ces méthodologies peuvent être classifiées en trois classes [102] :

- Une approche *proactive* est utilisée pour exploiter et créer une ligne de produits à partir de zéro, c'est-à-dire, qu'elle ne peut pas s'appuyer sur des produits existants ou des artéfacts déjà implémentés ;

- Une approche *reactive* est utilisée pour créer une ligne de produits en se basant sur des artéfacts et des systèmes patrimoniaux pour ensuite les intégrer dans la ligne de produits ;
- Une approche *extractive* est utilisée pour créer une ligne de produits en s'appuyant sur des systèmes existants. Les méthodologies de l'analyse des points communs appartiennent aux approches du développement de ligne de produits.

4.4.2. Evaluation et évolution

L'évaluation des architectures de référence est une tâche délicate. Elle a pour but de garantir la qualité de la famille de produits. Elle garantit aussi que les applications spécifiques pourront être dérivées correctement à partir de l'architecture de référence. Durant l'évaluation de l'architecture de référence, il est nécessaire de bien prendre en compte le mécanisme de la planification systématique des variabilités de la ligne de produits. Les deux précautions suivantes sont à considérer pour évaluer l'architecture de référence :

- L'architecture de référence a pour but d'être commune et partagée pour tous les produits spécifiques encadrés dans la portée de la ligne de produits. L'architecture de référence doit être assez générique afin d'avoir une vision globale de la ligne de produits.
- L'autre objectif de l'architecture de référence est de diriger le processus de la dérivation de produits spécifiques. Ainsi, l'architecture de référence doit être assez flexible afin d'avoir assez indices aux seins de la production des produits personnalisés.

Les architectures applicatives correspondantes aux applications spécifiques doivent être évaluées conformément à leurs exigences spécifiques. Dans certains cas, des exigences spécifiques s'avèrent insolubles lors de la conception de l'architecture applicative. Au contraire, elles se lient directement à la conception de l'architecture de référence. Des évaluations informelles [15] sont souvent menées, par exemple à l'aide des scénarios des cas d'utilisation, en faisant un calcul approximatif de la performance de système ou en développant une partie de l'architecture afin de tester certains morceaux de conceptions. Les évaluations plus formelles sont effectuées moins fréquemment. Les technologies servant à estimer et à évaluer une architecture sont divisées en deux catégories : *qualitative measures (e.g. scenarios, checklists, questionnaires)* et *quantitative measures (e.g.metrics, simulations, prototypes, experiments).* Une manière plus formelle de l'évaluation de l'architecture intitulée *ATAM (Architecture Tradeoff Analysis Method)* est introduite dans [81].

Ainsi, une architecture de référence de la ligne de produits est désirée essentiellement pour traiter des exigences spécifiques imprévues ou inattendues. Une architecture de référence de haute qualité est en effet évaluée sur sa capacité à traiter ce besoin. Elle a besoin d'être mise à jour continuellement. L'évolution de l'architecture de référence peut être provoquée par plusieurs raisons. Premièrement, les connaissances et la compréhension des experts et des architectes du domaine s'améliorent et demandent des changements. Deuxièmement, l'évolution des demandes du marché ciblé rend directement nécessaire l'évolution de l'architecture de référence. De plus, le *feedback* des utilisateurs par rapport aux systèmes livrés peut apporter de nouvelles exigences. Enfin, les évolutions technologiques (e.g. l'évolution du langage d'implémentation, l'évolution de la plateforme ou du paradigme de développement, et l'évolution de la bibliothèque des composants), les évolutions de versions des composants utilisés ou la fourniture de nouveaux composants par une partie tierce ont des impacts sur l'architecture de référence.

Quand des exigences imprévues apparaissent, elles ne sont pas prises en compte dans l'architecture de la ligne de produits ou dans une architecture applicative dans la plupart des cas. Ces nouvelles exigences sont normalement mises en œuvre par l'implémentation des composants existants de différente manière (e.g, par l'extension des composants existants, par l'héritage des composants existants, par la modification des composants existants). Toutes les modifications peuvent causer un problème de conformité entre la spécification de la conception de l'architecture et l'implémentation des systèmes. L'évolution de l'architecture de référence doit alors prendre du compte toutes les modifications du niveau de l'implémentation au niveau de l'architecture de référence de la ligne de produits.

4.4.3. Gestion de la variabilité

Comme nous l'avons vu précédemment, les lignes de produits permettent la réutilisation logicielle planifiée et stratégique grâce à l'introduction des mécanismes de la gestion de variabilités. Un point de variation montre une localisation dans l'architecture de référence, où des changements peuvent apparaître lors du processus de la dérivation de produits logiciels. Par ailleurs, au cours de la mise en œuvre d'un produit spécifique, nous avons présenté des mécanismes de gestion de variabilités aux différents moments de liaison (*binding times*). En effet, la gestion de variabilités de la ligne de produits doit être prise en compte selon les deux axes : la localisation et le moment de liaison [78, 103]. Le premier axe prévoit et planifie les changements potentiels par rapport aux différents produits dans la même famille. L'autre axe considère plutôt le moment où sont prises des décisions liées aux localisations ayant une variation. Un produit logiciel spécifique est constitué dès que toutes les décisions aux points de variations sont prises, à un moment de liaison.

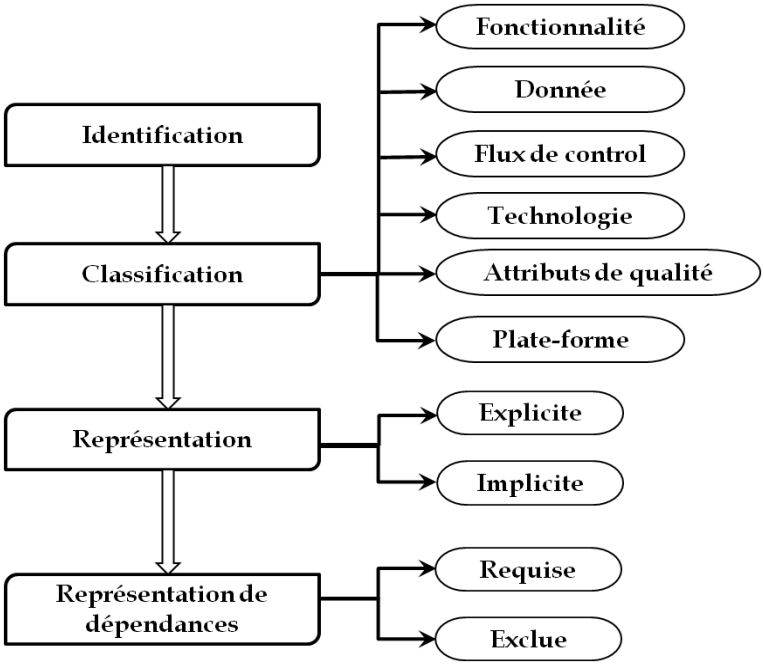


Figure 3.6. la gestion de variabilités dans l'architecture de référence

L'architecture de référence est une abstraction générique des solutions couvrant toutes les structures abstraites des produits logiciels spécifiques. La structure architecturale est une présentation

sous la forme de topologie de système logiciel. Les localisations correspondantes aux points de variation sont plus faciles à être explicitement identifiées et planifiées dans la topologie structurelle. Enfin, l'architecture de référence prend en charge non seulement la structure des fonctionnalités de la famille, mais aussi les attributs de la qualité des produits logiciels spécifiques. Cela réduit les risques du développement, puisque l'évaluation des attributs de la qualité de systèmes peut être mise en œuvre au cours de la conception au lieu de l'implémentation. Alors, les attributs de la qualité de systèmes sont considérés comme les éléments variables. Par conséquence, la gestion de variabilités de la ligne de produits dans l'architecture de référence est une manière efficace de traiter la diversité des produits logiciels.

L'architecture de conception est souvent considérée comme un modèle décrivant la structure de système sur un haut niveau d'abstraction. L'activité de planification de variabilités dans l'architecture de référence est donc un processus de la modélisation. La création de la structure de l'architecture de référence est mise en place lors de la conception du domaine. Elle prend en entrée l'ensemble des points communs, qui sont le résultat de l'activité de l'analyse des besoins du domaine. De même, l'activité de la modélisation de variabilités dans l'architecture commence lors de l'analyse des besoins du domaine pour identifier les variations et ensuite les classifier. Au cours de la conception de l'architecture de référence, l'activité de la modélisation de variabilités prend en charge la représentation des points de variation et leurs dépendances ainsi que les contraintes relatives. La figure 3.6 illustre les sous-activités principales de la modélisation de variabilités.

L'activité d'identification a pour but de répertorier les variantes à partir de l'ensemble des artéfacts réutilisables fournis par la plate-forme de la ligne de produits. Les points communs sont le pivot de l'architecture de référence, alors que les variations ajoutent les différences potentielles.

La deuxième activité est de reprendre les variantes et de les classifier. Des produits spécifiques peuvent être différenciés par des artéfacts fonctionnels et des attributs de la qualité de systèmes sous le contexte de la ligne de produits. Or, ils peuvent être aussi distingués par d'autres attributs spéciaux. Les catégories de variabilités des attributs en termes des sources [103] sont illustrées et fléchées à partir du bloc nommé « classification » dans la figure 3.6 :

- Variabilités de fonctionnalité : une fonctionnalité particulière peut exister dans un ou plusieurs produits spécifiques ;
- Variabilités de donnée : un type de donnée peut varier d'un produit personnalisé à l'autre ;
- Variabilités de flux de contrôle : un patron d'interaction peut varier de l'un produit à l'autre ;
- Variabilités de technologie : cela concerne plusieurs aspects, incluant les systèmes d'exploitation, les matériels, les intergiciels, les interfaces d'utilisateur et les plates-formes d'exécution ;
- Variabilités des objectifs de qualité : un objectif de qualité peut varier d'un produit à l'autre, en fonction des exigences spécifiques ;

En résumé, les variantes correspondantes à la technologie ou à l'environnement spécifique sont plutôt prises en compte dans l'architecture du point de vue de l'implémentation et du déploiement d'applications. Par contre, la communication entre l'ingénieur et le client se focalise plutôt sur les fonctionnalités fournies ainsi que la qualité du produit logiciel représenté dans l'architecture de point de vue de la logique. Les activités à la suite se concentrent donc sur la manière de représenter les variations à partir de ce dernier point de vue. Enfin, l'activité de la classification rassemble les variantes relatives

selon les différentes fonctionnalités ou attributs de qualité dans les différents ensembles, qui se lient aux points de variation.

La représentation de la variabilité dans l'architecture de référence se réfère à la définition des localisations où des variations peuvent potentiellement apparaître, il s'agit de définir les points de variation. Ensuite, chaque point de variation doit attacher un ensemble approprié, qui est issu de la classification de variabilité. À la fin, il faut déterminer un type (défini dans la section 2.3 comme *Indispensable, Optionnel, Alternatif et Multiples*) de la dépendance entre les variantes associées. La représentation des points de variation peut être argumentée de façon implicite dans la documentation de la description de l'architecture. Au contraire, les points de variation peuvent être explicitement présentés dans la topologie de l'architecture de référence à l'aide de certaine méthodologie ainsi que ses outils associés.

5. Conclusion

Les lignes de produits visent une réutilisation logicielle planifiée et stratégique. Elles se caractérisent par la mise en place d'un double processus de développement, *pour* et *par* la réutilisation. Les deux processus incluent du reste trois activités de développement similaires concernant l'analyse des exigences, la conception et l'implémentation des artéfacts logiciels. Le but du premier processus, l'ingénierie du domaine, est de spécifier et d'implanter des artefacts logiciels communs à tous les logiciels de la ligne de produits, tout en identifiant des points d'adaptation. Le but du second processus, l'ingénierie applicative, est de produire des applications spécifiques en s'appuyant sur les artefacts mis à disposition par le processus précédent. Ces artefacts sont adaptés en fonction des besoins spécifiques. De nouveaux développements sont envisageables et peuvent, éventuellement, intégrer la base de réutilisation. Ainsi, l'approche lignes de produits correspond typiquement à une approche de réutilisation anticipée et planifiée. Les applications spécifiques sont construites en utilisant les mécanismes de variabilités mis en place lors de l'ingénierie domaine.

L'architecture de référence joue un rôle fondamental. De sa qualité, qui est liée au savoir faire et à l'expertise des architectes métier, dépend la réussite de la ligne de produits. Une architecture de référence de qualité doit permettre aux développeurs applicatifs de se focaliser sur l'automatisation de l'assemblage et la personnalisation des composants du domaine spécifiques et non plus sur la mise en œuvre manuelle des applications [70]. Les architectures de référence, qui servent à anticiper les spécificités applicatives ou les évolutions, doivent elles-mêmes évolué. Cela est nécessaire pour intégrer les changements qui surviennent inévitablement et qui concernent toute une ligne de produits. La maintenance de l'architecture de référence de la ligne de produits est une tâche complexe.

Les lignes de produits sont aujourd'hui mises en place avec succès. Elles ont néanmoins un coût, organisationnel et technique, qui effraie nombre d'entreprises. Il est intéressant de noter que des approches très bénéficiaires ont souvent été mises en place dans des entreprises au bord de la faillite qui ne pouvaient survivre sans une réutilisation effective dans leurs atouts.

Cette approche à d'autres limites que son coût. En particulier, les prescriptions issues du SEI par exemple n'abordent pas suffisamment les aspects techniques qui, pourtant, sont difficiles à mettre en place. En particulier, la gestion du dynamisme à l'exécution (et, ainsi, l'affinement très retardé d'une architecture applicative) est un réel défi.

Enfin, la séparation de l'approche en deux processus n'apporte pas que des avantages. Il est en effet difficile de synchroniser les deux processus. Une modification dans une architecture applicative est difficilement reportée dans l'architecture de référence. Pareillement, une modification de l'architecture de référence pose immédiatement le problème de la compatibilité avec les applications antérieures.

Enfin, malgré de récentes tentatives [104], les lignes de produits ne proposent pas de dynamisme à l'exécution. Elles ne conviennent pas très bien à des environnements dynamiques, changeants.

Deuxième partie

Contribution

Chapitre 4 Proposition

La première partie de cette thèse a présenté un état de l'art sur deux approches majeures de réutilisation, à savoir les lignes de produits et l'informatique à services. Elle a mis en évidence les avantages mais aussi les limites de ces approches. Cette seconde partie aborde notre proposition qui vise à concilier les approches à services et à base de ligne de produits. Elle est structurée en cinq chapitres qui détaillent peu à peu notre travail et donnent des éléments de validation.

Ce chapitre 4 rappelle notre contexte de travail et présente une vision globale de notre approche. Celle-ci repose sur la définition de trois phases successive pour permettre le développement d'applications à services distribués et hétérogènes. Les détails sur l'approche sont donnés dans les chapitres suivants.

1. Rappel du contexte

Les travaux de cette thèse visent à simplifier le développement d'applications à services logicielles dans des environnements distribués, dynamiques et hétérogènes. Comme nous l'avons présenté dans la première partie de l'état de l'art, il existe actuellement de nombreux modèles à services et de multiples technologies d'implémentation. Une application basée sur les services peut être composée de services distribués et hétérogènes, qui peuvent être implémentés par différentes technologies à services, telles que les Services Web, les services iPOJO, les services UPnP ou encore les services DPWS.

En général, une technologie est choisie pour servir de pivot lors du développement d'applications. Des mécanismes de communication, plus ou moins transparents, sont alors développés pour permettre l'interaction entre les différentes technologies (chaque technologie étant liée à une plate-forme). Dans notre cas, la technologie pivot est le modèle à composant iPOJO, fonctionnant au dessus d'OSGi, et nous avons développé les ponts vers d'autres technologies telles que la technologie UPnP ou les services Web.

Au cours du développement d'applications pervasives, nous nous sommes peu à peu rendu compte que le développement d'applications basées sur services dans un environnement distribué, dynamique et hétérogène est une tâche techniquement complexe. Cela est dû, en particulier, aux aspects suivants:

- **Gestion du dynamisme.** Le dynamisme est une caractéristique importante des architectures orientées services. Un service peut dynamiquement apparaitre ou disparaitre sur le réseau sans aucun avertissement préalable. Lors de l'exécution d'une application, il faut prendre en charge la vérification de la disponibilité des services hétérogènes sur le réseau. Cela n'est pas une tâche simple. Le dynamisme se en fait caractérise par deux aspects. Le premier aspect correspond aux changements du contexte informatique, à savoir certains équipements à services comme UPnP ou certains services Web se connectent ou déconnectent du réseau. Un autre aspect est lié aux changements des besoins des utilisateurs pendant l'exécution d'une application. Evidemment, la gestion de ces deux types de dynamisme des technologies à services induit une forte complexité au niveau de la de compositions de services.
- **Gestion de l'hétérogénéité et de la distribution.** Les applications à services fonctionnent généralement dans des environnements distribués et hétérogènes. Lors de la construction d'une application, on cherche à utiliser tous services satisfaisant les contraintes architecturales, quels que soient leur distance et leur technologie d'implantation. La distribution pose le problème toujours délicat de la gestion des communications. L'hétérogénéité, quant à elle, pose le problème de la maîtrise de technologies souvent très spécifiques pour la découverte et l'invocation de services.
- **Gestion de la conformité des assemblages**. Il est très difficile de garantir, et de vérifier, que le comportement d'une application composite est conforme aux attentes. La plupart des approches à services actuelles « se contentent » de vérifier les conformités syntaxiques. Les conformités sémantiques ou les variations possibles dans les compositions sont peu abordées [105]. Par ailleurs, la vérification de la conformité devient extrêmement ardue lorsqu'on doit intégrer diverses technologies. L'utilisation d'une technologie particulière pour implanter un service n'est pas une décision anodine.

Nous avons également vu dans l'état de l'art que la mise en place de lignes de produits logicielles permettait la réutilisation effective de composants logiciels, qui sont relativement proches technologiquement des services. Pour autant, les lignes de produits demeurent relativement statiques et ne peuvent, aujourd'hui, facilement traiter des applications très dynamiques telles que celles rencontrées en informatique ambiante.

2. Proposition

La proposition défendue dans cette thèse pour faciliter le développement d'applications à services est de concilier les approches à services et les approches « lignes de produits » en proposant une démarche outillée de composition de services structurée en trois phases, à savoir :

- **Définition d'un domaine autour de la notion de services.**
  L'objectif de cette première phase est de spécifier des architectures à service de référence et un ensemble de services réutilisables au sein d'un domaine. Les services sont spécifiés de façon abstraite, et de façon plus ou moins précise, de façon à rester indépendants de toutes technologies à services. Ces services peuvent être implantés suivant diverses technologies et donner lieu à plusieurs versions en fonction des plates-formes d'exécution ou des fonctionnalités exactes qui sont fournies. Les architectures de référence sont constituées de ces spécifications de services, des règles d'assemblage de ces services et de variabilité sous diverses formes. L'objectif de cette architecture est de fournir un cadre architectural commun à toutes les compositions de services au sein du domaine visé. Cette phase est assimilable à la phase d'ingénierie du domaine des lignes de produits, dans le cadre conceptuel du paradigme à service.
- **Définition d'une architecture applicative sous la forme d'une composition de services.**
  L'objectif de cette seconde phase est de définir une architecture applicative répondant à un besoin particulier. Cette architecture, conforme à l'architecture de référence, est constituée de services, qui peuvent être abstraits ou concrets. Cela signifie que certains services sont clairement identifiés (technologie d'implantation spécifique, version) alors que d'autres restent sous la forme de spécifications. L'architecture applicative peut inclure de la variabilité, au même titre que l'architecture de référence. Garder un niveau de variabilité dans cette architecture permet de s'adapter à l'exécution à un environnement changeant et imprévisible. Cette phase est assimilable à la phase d'ingénierie applicative des lignes de produits, dans le cadre conceptuel des approches orientées service. Elle se conclut par le déploiement de l'architecture applicative sur une plate-forme d'exécution à services.
- **Exécution autonomique de l'application à service.**
  L'objectif de cette troisième phase est de lancer l'exécution de l'application à services en conformité avec son architecture applicative. Au cours de l'exécution, les dernières décisions de conception laissées libres dans l'architecture applicative sont prises. En particulier, des instances de service sont sélectionnées ou créées lors de l'exécution, au dernier moment possible. Le modèle de l'architecture applicative est utilisé pour diriger le processus de composition des services. Par conséquent, toutes les compositions de services doivent se conformer à ce modèle de l'architecture applicative. Dans le cadre de cette thèse, les applications sont exécutées sur une plate-forme OSGi/iPOJO étendue avec des fonctionnalités de découverte et de réification de services distribués et hétérogènes (UPnP et WS). Les aspects techniques d'intégration de technologies à services hétérogènes sont pris en compte par la plate-forme d'exécution.

Nous avons suivi une approche dirigée par les modèles pour mettre en place notre approche. Cela nous a amené à définir des méta-modèles (langages) pour la définition de spécifications de services, des implantations de services et pour les architectures, de référence et applicatives. A partir des méta-modèles et des modèles associés nous avons créé des outils de génie logiciel. Le premier outil est un atelier, intégré sous Eclipse, permet la définition des services et des architectures de référence. Le deuxième outil, également intégré sous Eclipse, permet la définition des architectures applicatives. Le deuxième outil est dérivé à partir du résultat du premier outil de façon automatique. Enfin, le dernier outil prend la forme d'une machine d'exécution de service capable de gérer l'arrivée et la départ de services obéissant à différentes technologies (dans notre cas, iPOJO, UPnP, et WS), de sélectionner les services appropriés en fonction d'informations architecturales, et de réaliser les liaisons entre services.

Notre proposition est résumée de façon schématique par la figure 4.1, ci-après.

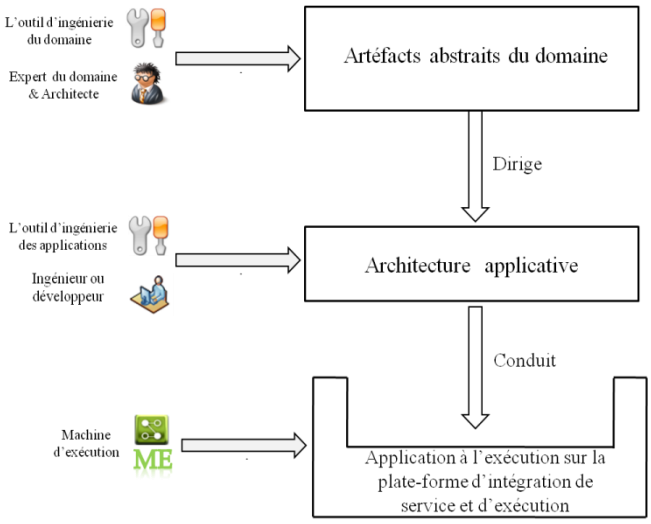


Figure 4.1. Composition de services en trois phases

Le but de notre approche est de réunir les points forts des approches à services et des approches lignes de produits. Ainsi, elle emprunte aux lignes de produits les notions d'ingénierie du domaine et applicative et, de façon générale, la volonté de mettre en place une réutilisation planifiée, de granularité élevée (niveau composant) au sein d'un domaine. Elle intègre également à tous les niveaux des notions de variabilité permettant de retarder des décisions de conception. Notre approche s'appuie aussi sur la propriété de dynamisme des approches à services de façon à retarder le plus tard possible à l'exécution les choix des services réels à utiliser, en fonction de leur disponibilité et autres propriétés au moment où l'invocation doit se faire. De façon plus précise, notre approche apporte les avantages suivants :

- **Elle met en œuvre une réutilisation planifiée et systématique.** Le développement d'un environnement domaine, constitué de services assemblés au sein d'une architecture de référence, permet de fournir un cadre précis de réutilisation. Les experts du domaine définissent ce cadre et les ingénieurs de développement l'emploient pour réutiliser de façon contrôlée les services métier mis à leur disposition. La réutilisation, ainsi planifiée, apporte de meilleurs résultats
- **Elle garantit la correction des applications à service. Les dé**veloppeurs bénéficient d'un cadre d'assemblage des services, à savoir l'architecture de référence. Ainsi, ils n'ont pas besoin de maîtriser toutes les spécificités d'un domaine pour faire une composition. Par ailleurs, les services, au sein du domaine, ont été choisis ou implantés suivant des spécifications garantissant leur compatibilité. Les services qui sont réellement utilisés pour une application donnée ont des types en accord avec ceux prévus au sein de l'architecture de référence.
- **Elle simplifie le développement des applications à service.** La mise en œuvre des applications à services est beaucoup plus accessible pour des développeurs non experts en services. La machine d'exécution se charge de sélectionner les services en fonction de critères décidés par les développeurs et se charge de générer, ou de paramétrer, le code de gestion et d'invocation des services. Ainsi, les développeurs peuvent se concentrer sur le logique métier de l'application et ne se connaissent pas tous les détails des différentes technologies à services à intégrer.
- **Elle profite pleinement du dynamisme inhérent à l'approche à services.** Les instances de service devant être liées sont choisies, autonomiquement, au dernier moment. Elles peuvent également être modifiées autonomiquement de façon à prendre en compte des conditions d'exécution nouvelles (nouvel environnement, nouveaux besoins utilisateur). Cette adaptation dynamique est dirigée par le modèle que constitue l'architecture de référence.

3. La phase d'ingénierie du domaine

La première phase de notre approche a pour objectif de définir les éléments réutilisables au sein d'un domaine en termes de services. Nous avons pour cela défini de façon précise les notions de spécifications de services, d'implantation de services, et d'architectures à services. Les services sont en effet les éléments fondamentaux de notre approche en ce sens qu'une application est composée par l'assemblage de telles unités. Une architecture de référence est le canevas de composition des services au sein d'une famille d'applications. Nous n'abordons pas dans cette thèse les aspects liés à la gestion des exigences du domaine spécifique.

Les **spécifications de service** définissent les blocs fonctionnels utilisables dans le domaine pour répondre aux diverses exigences. Elles sont ainsi construites à partir de l'analyse du domaine. Les spécifications de service sont décrites suivant un langage bien défini, structuré autour des notions d'interfaces fonctionnelles et de propriétés. Plus précisément, une spécification de service est un modèle conforme à un méta-modèle que nous avons défini (voir chapitre 5). Les spécifications de service constituent les éléments de base pour définir des architectures de référence.

Une **implantation de service** est la description d'une implantation suivant le modèle à composant orienté service iPOJO. Une implantation de service est également un modèle conforme à un méta-modèle que nous avons défini (voir chapitre 5). Nous avons choisi iPOJO comme modèle d'implantation pivot pour ses qualités de dynamisme à l'exécution. Utiliser un modèle pivot est souvent bien accepté par les développeurs puisqu'ils ne manipulent qu'une technologie lors de la construction d'applications. Cela a néanmoins un coût en terme d'ingénierie car il est nécessaire d'implanter des ponts entre le pivot et les autres technologies. La description d'une implantation de service reprend au minimum les propriétés et les interfaces d'une spécification de service et peut ajouter des informations propres à une technologie d'implantation.

Nous avons défini une **relation de référence explicite** entre spécifications de service et implantations de service ; une spécification de service pouvant être réalisée par une ou plusieurs implantations. Les références servent à l'établissement de la correspondance entre des spécifications de services et des implantations utilisant des formats de description différents.

L'architecture de référence est la description structurelle d'une famille d'applications à services. Elle est constituée de spécifications de service, de relations entre ces services, de règles d'assemblage, et de contraintes architecturales. Par le biais de la relation précédente, on peut considérer qu'une architecture de référence contient également des ensembles possibles d'implantations de services. Une architecture de référence est également un modèle conforme à un méta-modèle que nous avons défini (voir chapitre 5). L'objectif d'une architecture de référence est de fournir une structure abstraite et générique capable de répondre aux exigences des applications du domaine. Pour cela, elle doit comporter des éléments communs à toutes les applications du domaine (services, patterns d'assemblage, règles, etc.), tout en conservant suffisamment de flexibilité pour répondre aux besoins personnalisés de certaines applications. La flexibilité de l'architecture repose sur l'abstraction et des points de variation explicites.

Par le jeu des formes diverses de variation, on peut définir des architectures de référence plus ou moins précises. Dans un premier lieu, il existe certains domaines, tel que celui des applications pervasives par exemple, où une grande flexibilité est requise pour faire face à des environnements très dynamiques et peu prévisibles. Cela conduit à des architectures très flexibles, incluant de nombreuses structures optionnelles ou configurables. L'inconvénient, bien sûr, est que ces architectures ne fournissent finalement que peu de guidage lors du développement d'une application. Pour remédier à cette limitation, nous proposons la création de plusieurs architectures de référence, liées par des relations bien définies, au sein d'un même domaine.

4. La phase d'ingénierie applicative

L'objectif de la deuxième phase est de définir des applications sous la forme de compositions de spécification de services conforme à une architecture de référence. Cette composition de service est appelée architecture applicative. L'architecture applicative servira de guide lors de la composition de services concrets ayant lieu lors de l'exécution. Pour permettre une certaine adaptation au contexte, elle peut conserver des points de variabilité, tout comme l'architecture de référence. Elle est néanmoins beaucoup plus spécifique qu'une architecture de référence.

Une **architecture applicative** est constituée de spécifications de service, de relations entre ces services, de règles d'assemblage, et de contraintes architecturales. Une architecture applicative est un modèle conforme à un méta-modèle que nous avons défini (voir chapitre 5). L'ensemble est conforme à une architecture de référence en ce sens qu'elle doit respecter la topologie et les services spécifiés par l'architecture de référence. La notion de conformité est reprise plus en détail dans le chapitre suivant.

Nous avons mis en place une approche dirigée par les modèles de façon à garantir la conformité en architectures applicatives et architectures de référence. Cette approche repose sur transformations de modèles à partir du modèle de l'architecture de référence et du modèle de spécifications de service du domaine. La transformation de modèles permet de définir la relation correspondante entre les éléments dans le modèle de l'architecture de référence et le métamodèle d'architecture métier comme la relation d'héritage. Ainsi, les architectures applicatives définies par ce langage sont naturellement conformes à l'architecture de référence. De plus, cette transformation de modèles automatise le passage entre les deux phases de développement afin de construire l'outil d'ingénierie des applications de façon générative. Nous étudions précisément cette transformation de modèles dans le chapitre 6.

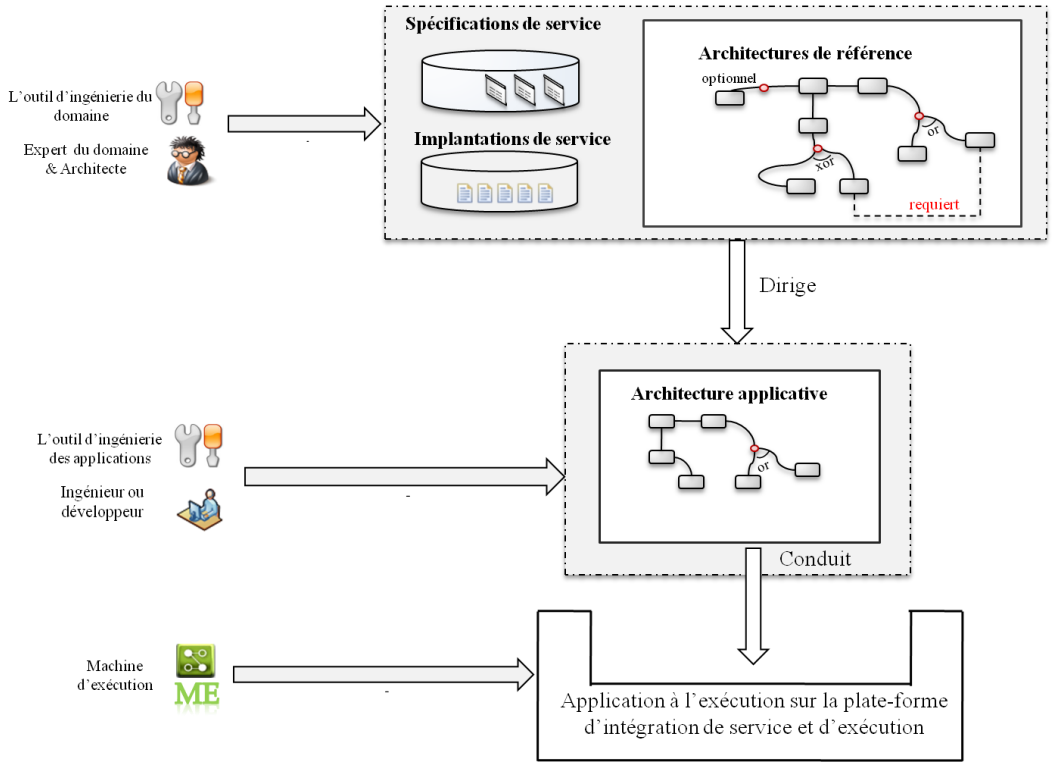


Figure 4.2. Liens détaillés entre ingénierie du domaine et ingénierie applicative.

5. La phase d'exécution

La troisième phase de notre approche a pour but de gérer de façon autonomique une application à services à partir de l'architecture applicative précédemment définie et des services disponibles sur la plate-forme d'exécution.

Cette phase de développement est soutenue par une machine d'exécution que nous avons développée. Elle est composée de deux composants logiciels importants: une plate-forme d'intégration de service et un moteur d'interprétation. Le rôle de la plate-forme d'intégration de service est de surveiller la disponibilité de services sur le réseau et d'exécuter l'application elle-même. Elle est également chargée d'inspecter continuellement l'état de l'application et l'état de la plate-forme d'exécution. Le rôle du moteur d'interprétation est d'interpréter l'architecture applicative afin de créer une application par l'assemblage des instances de services sélectionnées. Ce moteur d'interprétation gère également de façon autonomique les applications afin de s'adapter aux changements du contexte lors de leur exécution. La machine d'exécution repose sur une plate-forme cible en iPOJO au-dessus de OSGi. Le schéma ci-après montre la vision globale de la machine d'exécution.

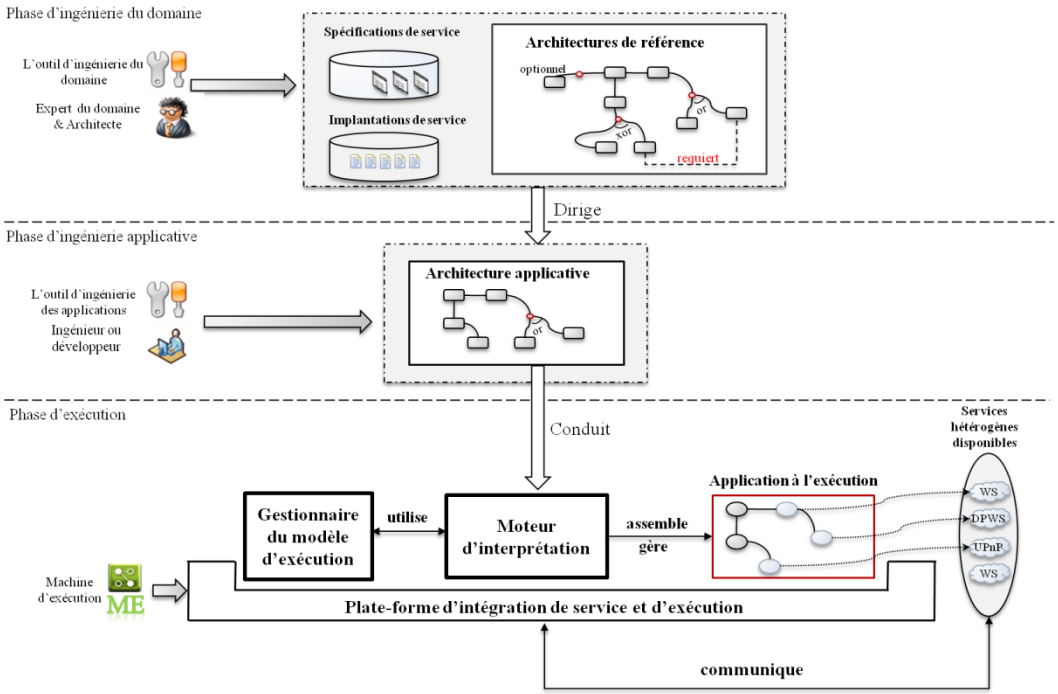


Figure 4.3. Vue détaillée de l'approche.

Pour gérer une application à services, le moteur d'interprétation considère l'architecture applicative ainsi que le modèle d'exécution afin de connaître la disponibilité des services sur la plate-forme d'exécution. Il s'appuie alors sur une stratégie de sélection de services afin de sélectionner les services disponibles répondant aux spécifications définies au sein de l'architecture applicative. Dès que le moteur d'interprétation a résolu toutes les variations au sein de l'architecture applicative, l'application est finalement créée par l'assemblage des services sélectionnés suivant son architecture. Cette stratégie est également valable pour la mise à jour d'une application dans le cas d'un changement de disponibilité des services.

6. Synthèse

Le but de ce chapitre était de présenter de façon synthétique la proposition constituant le cœur de notre thèse. Notre approche est d'étendre les notions définies par les lignes de produits pour s'adapter à la problématique des applications à services. En particulier, les phases d'ingénierie du domaine et d'ingénierie applicative sont structurées autour de la notion de services. Par ailleurs, la phase d'exécution est beaucoup complexe dans notre approche puisqu'elle est gérée de façon autonomique de façon à apporter la flexibilité nécessaire aux environnements dynamiques.

Comme indiqué, nous avons choisi d'établir notre proposition suivant une approche dirigée par les modèles. Cela nous a amené à construire différents méta-modèles définissant de façon précise la sémantique associée aux notions de services et d'architectures.

Le but des chapitres suivants est de présenter plus en détail les trois phases constituant notre approche, les méta-modèles définissant les éléments majeurs et les outils générés à partir de ces méta-modèles.

Chapitre 5 Services et architectures

L'approche proposée dans cette thèse est structurée en trois phases de développement. La première phase, ou ingénierie du domaine, vise à mettre en place au sein d'un domaine d'un ensemble d'éléments réutilisables fondés autour des notions de services et d'architectures à services. La seconde phase, ou ingénierie applicative, concerne l'exploitation des éléments communs précédemment définis. Son objectif est de spécifier, de façon plus précise, les applications à réaliser. Enfin, la troisième phase, a pour but d'exécuter l'application spécifiée en identifiant et utilisant au dernier moment les services appropriés.

Ce chapitre se concentre sur les notions de services et d'architectures à services en les définissant de façon précise à l'aide de méta-modèles.

1. Introduction

Contrairement à de nombreuses technologies actuelles, nous pensons qu'il est important de nettement distinguer les notions de spécification de service, d'implantation de service et d'instance de service. Les Web services, par exemple, tendent à effacer la notion d'instance de service et à confondre les notions de spécification et d'implantation. Les fichiers WSDL en effet contiennent des informations liées à la description de service et d'autres directement concernées par l'implantation du service, son adresse par exemple. Ceci complique la compréhension et l'appréhension des notions de service et d'applications à service. Distinguer clairement spécification, implantation et instance nous semble nécessaire pour gérer explicitement les différentes phases du cycle de vie des applications à services, notamment celles incluant des technologies différentes comme c'est le cas dans le domaine pervasif.

Plus précisément, nous définissons les notions de spécifications, d'implantation et d'instance de service de la façon suivante :

- Les spécifications de service décrivent les fonctionnalités fournies et parfois requises par les services ainsi qu'un ensemble de propriétés les caractérisant. Nous verrons par la suite que les spécifications de service sont, dans notre modèle, indépendantes des technologies d'implantation.
- Les implantations de services décrivent la façon dont une spécification de service est réalisée dans une technologie particulière. Les implantations peuvent se faire en utilisant diverses technologies incluant les services Web, UPnP, DPWS et iPOJO. Cependant, dans notre approche, toutes les implantations non iPOJO donnent lieu à la création d'un pont (proxy) iPOJO vers cette implantation. Un développeur ne manipule ainsi qu'un seul modèle de développement.
- Les instances de services sont la matérialisation à l'exécution des implantations de services. Elles correspondent à du code binaire. Des fabriques peuvent être définies de façon à faciliter la création de ces instances. Contrairement aux implantations, celles-ci peuvent évoluer au cours du temps : les valeurs des attributs peuvent changer et le comportement de certaines fonctionnalités se modifier en fonction du contexte.

Ces trois notions sont très liées, comme l'indique la figure 5.1, ci-après :

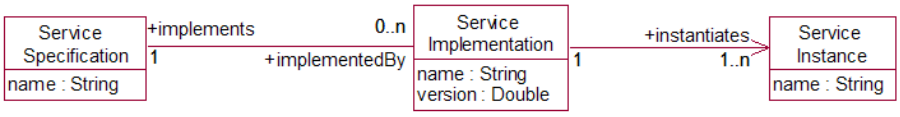


Figure 5.1. Spécification, implantation et instance de service.

Comme indiqué par la figure 5.1, une spécification de service peut donner lieu à différentes implantations de services. Ces implantations peuvent se distinguer au niveau de leurs propriétés non fonctionnelles, de leur version, des technologies employées, etc. Les implantations de services peuvent également permettre la création de multiples instances qui peuvent être caractérisées par des configurations initiales différentes.

Les spécifications, implantations et instances de services sont explicitement manipulées dans notre approche. Ils forment les éléments essentiels des architectures de référence et des architectures applicatives que nous proposons de construire. Ces différents éléments sont décrits plus en détail dans la suite de ce chapitre.

2. Spécification de service

2.1. Définition

Comme précédemment introduit, une spécification de service décrit les fonctionnalités fournies et parfois requises par un service ainsi qu'un ensemble de propriétés le caractérisant.

De façon à gérer l'hétérogénéité des technologies à service, nous avons défini un langage de spécification de service indépendant de toute technologie. Ce langage se concentre sur la description des fonctionnalités et propriétés des services et n'aborde en aucune façon des aspects propres aux implémentations ou instances de service. Il abstrait les détails propres aux technologies à services existantes et cache leur complexité. Ce langage s'adresse aux architectes et aux concepteurs désireux de travailler au niveau des modèles de spécification et soucieux, à ce stade du développement, de prendre de la distance par rapport aux technologies d'implantation.

Plus précisément, une spécification de service est définie de la façon suivante :

- **des interfaces fonctionnelles** de type Java spécifiant les fonctionnalités fournies et requises par le service. Une interface peut être définie comme un ensemble d'opérations ou comme un ensemble de types de message pour le transfert de flux de données.
- **des propriétés,** identifiées par leur type ainsi que par leur nom, caractérisant le service. Nous divisons les propriétés en deux types :

Les *propriétés de service* définissent des attributs permettant de spécifier les aspects fonctionnels des services tels que les formats manipulés par le service par exemple. Les *propriétés de qualité* définissent des attributs permettant de qualifier certains aspects non fonctionnels des services tels que la sécurité, le coût ou la persistance. Par exemple, on peut définir un attribut *traçabilité* de type booléen. Dès qu'il devient *vrai*, toutes les opérations de ce service doivent être tracées. On peut également définir des attributs de qualité de façon plus fine. Un tel attribut peut concerner, pour la traçabilité par exemple, une interface ou une opération précise.

Les propriétés de services peuvent être statiques ou configurables. Les propriétés statiques ne peuvent pas être modifiées dès qu'elles sont définies. Il peut s'agir, par exemple, de la définition d'un format de message de la spécification de service. Les propriétés configurables, quant à elles, peuvent être affectées ou modifiées lors de la spécialisation de l'architecture d'une application au cours de l'ingénierie applicative. En particulier, elles sont utilisées pour configurer la spécification de service lors du processus de personnalisation d'application à services. Cela peut concerner, par exemple, la destination d'envoi d'un message ou le protocole utilisé pour envoyer un message.

La qualité de service peut donner lieu à une spécification plus complexe qu'un simple attribut. Il s'agit d'encapsuler au sein d'un ou plusieurs concepts un ensemble de caractéristiques liées à la qualité d'un service. On peut, par exemple, spécifier la traçabilité de façon plus fine en donnant les informations devant être conservées en fonction de l'état du service ou, même, du contexte d'exécution au sens large. Dans par exemple, la sécurité de service est modélisée de cette façon par un ensemble de concepts abordant les problématiques d'authentification, de confidentialité, etc.

Nous avons également introduit une notion d'héritage entre spécifications de service afin de pouvoir regrouper au sein d'une même hiérarchie certains services proches et, ainsi, construire des architectures de référence de façon plus synthétique. Plus précisément, les fonctionnalités fournies et requises d'un service peuvent être raffinées et enrichies par spécialisation.

Cette relation peut être assimilée à l'héritage de classes en Java et en reprend les principes :

- Une spécification de service héritant d'une autre spécification de service peut non seulement fournir toutes les fonctionnalités du service ***A***, exposer les propriétés qui y sont définies, mais aussi modifier certaines des ces fonctionnalités et ajouter de nouvelles fonctionnalités ou de nouvelles propriétés ;
- L'héritage est simple en ce sens qu'une spécification de service ne peut pas hériter de plusieurs spécifications de services ;
- La propriété de transitivité indique que si la spécification de service ***A2*** hérite de la spécification de service ***A1*** qui hérite de la spécification de service ***A***, alors ***A2*** hérite de ***A***. Cette propriété implique également que les dépendances d'une spécification de service sont héritées par ses spécialisations.

Dans le méta-modèle présenté par le figure 5.2, la relation d'héritage entre deux spécifications de service est définie concrètement à l'aide d'une relation appelée *extends*.

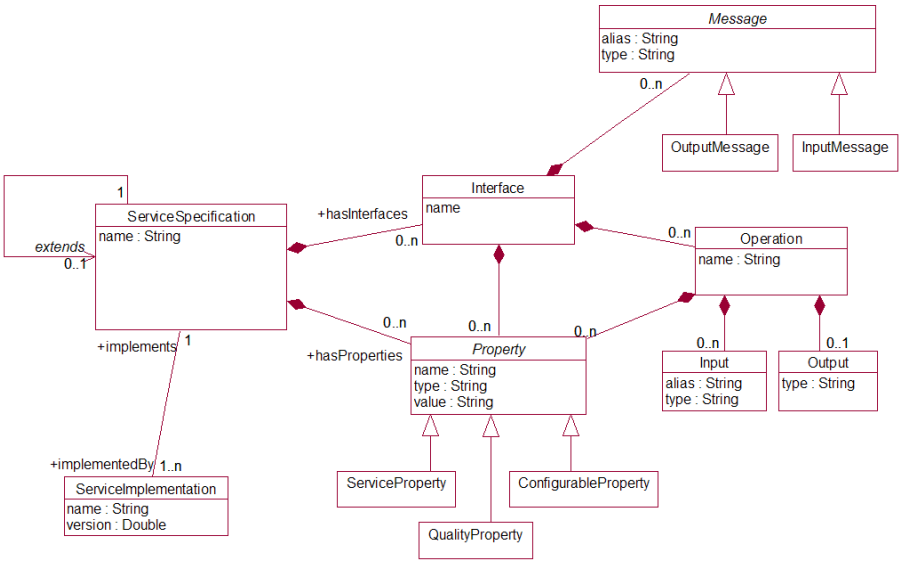


Figure 5.2. Méta-modèle pour la spécification de service

Nous verrons que l'introduction de la relation d'héritage entre les spécifications de services augmente l'expressivité et la flexibilité des architectures de référence et des architectures applicatives. Au sein d'une architecture de référence, on peut par exemple créer une relation potentielle entre une spécification de service et un ensemble de spécifications de service par la création d'une spécification de service généralisée (sommet de la hiérarchie). Cela permet de modéliser plus facilement et plus clairement des familles d'applications. Ceci est, bien sûr, particulièrement utile lors de la phase d'ingénierie applicative où l'on est amené à réduire la variabilité de l'architecture de référence et donc choisir certaines spécifications de service au sein d'une hiérarchie. Grâce aux hiérarchies d'héritage, on pourra également faire évoluer de façon localisée et maîtrisée les architectures de référence en ajoutant, par exemple, un nouveau sous-type de service. Ce type d'extension doit néanmoins rester limité, sous peine d'avoir une dérive forte de l'architecture qui empêcherait une réelle réutilisation planifiée. En général, les spécifications de service ajoutées dans l'architecture applicative sont non réutilisables et ne sont pas répercutées dans l'architecture de référence.

2.2. Exemple

Nous illustrons ici la notion de spécification de service avec des exemples issus de notre domaine d'applications, les applications d'hospitalisation à domicile. Ces applications, généralement rattachées à l'informatique ambiante utilisent des capteurs dispersés dans un bâtiment pour apporter une aide aux occupants du bâtiment. La figure 5.3 présente un ensemble de spécification de services que nous avons identifié. Les relations d'héritage entre ces spécifications de service sont matérialisées par des flèches. Les flèches pointent vers les spécifications de service généralisées alors que les spécialisations des spécifications de service sont à l'origine des flèches.

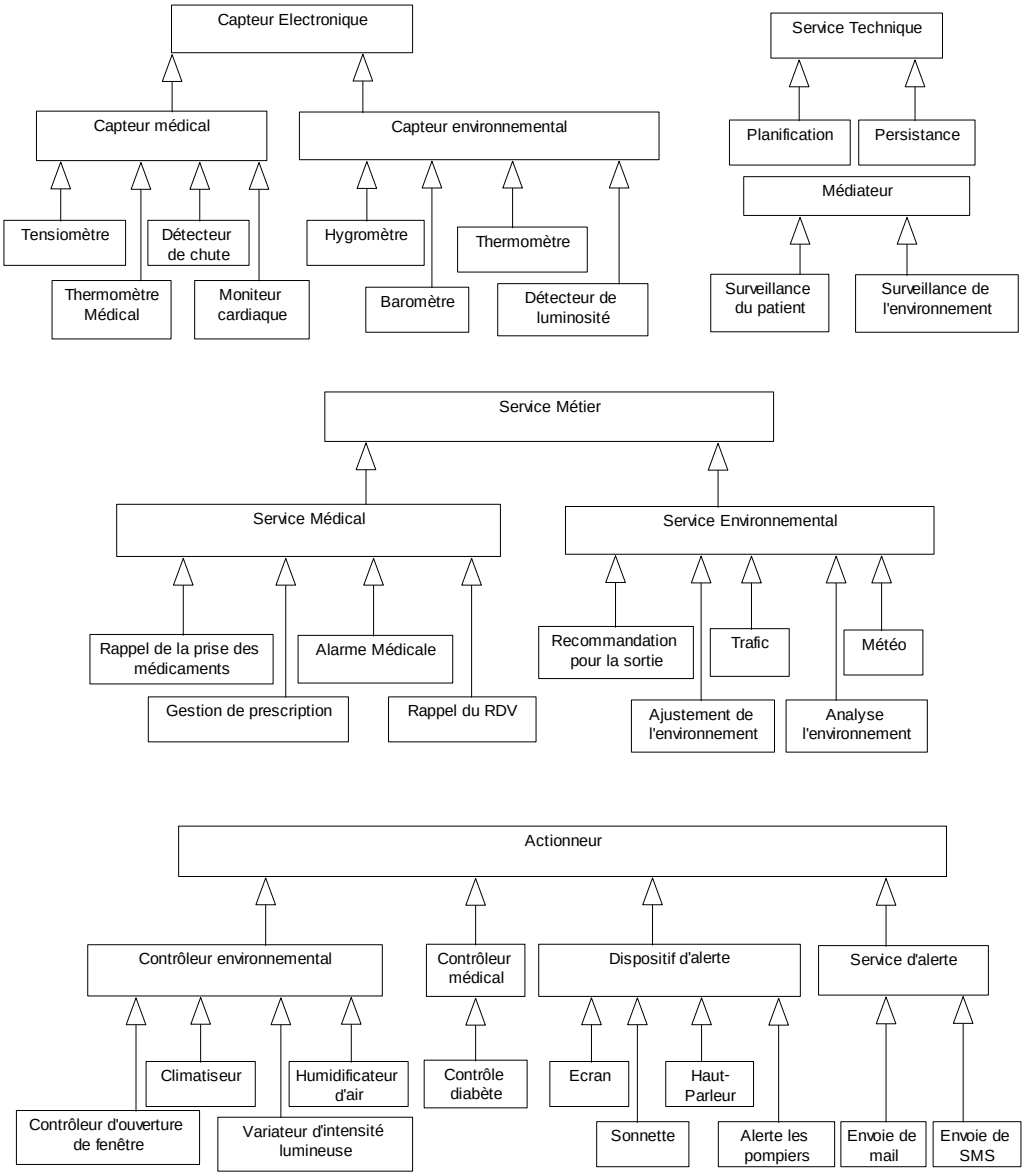


Figure 5.3. Les spécifications de service et la relation d'héritage entre eux

Cet exemple montre bien l'importance de la notion d'héritage entre les spécifications de services. Dans un domaine, il existe en effet de très nombreuses spécifications de services. Sans des mécanismes de regroupement de ces spécifications de services, il serait très difficile d'exprimer simplement des

architectures de référence (on aurait rapidement des architectures présentant de forts couplages). Une spécification de service peut ainsi être définie sans aucune interface fonctionnelle ou aucune propriété de façon à représenter une catégorie. Ceci permet d'organiser et de hiérarchiser les spécifications de services issues de l'analyse du domaine.

Les hiérarchies de spécifications de services créent des catégories de services, regroupant et positionnant des éléments métier proches les uns par rapport aux autres. La spécification de service représentant une catégorie se trouve au plus haut niveau de la hiérarchie des artéfacts réutilisables.

La figure 5.3 met en évidence des services que l'on retrouve fréquemment dans les bâtiments intelligents :

- Les capteurs environnementaux fournissent des informations sur les aspects physiques liés au bâtiment telles que la température ou le taux d'humidité. Ils sont en général paramétrables et présentent des interfaces spécifiques aux paramètres physiques qu'ils mesurent.
- Les capteurs médicaux fournissent des informations physiologiques sur les personnes assistées médicalement. Ils peuvent parfois inclure des effecteurs pour agir directement sur les patients (ceci demande la mise en place de précautions importantes). Le moniteur cardiaque permet de contrôler régulièrement les battements du cœur. Le tensiomètre fournit régulièrement la tension du patient. Le détecteur de chute permet de repérer une chute du patient et, éventuellement, d'émettre une alerte.
- Les médiateurs correspondent à des éléments purement logiciels qui se chargent du traitement des données issues des capteurs électroniques. Ces traitements peuvent inclure des transformations, des filtrages, des stockages, ou des opérations fonctionnelles. Par exemple, « Surveiller le patient » peut servir à stocker et vérifier les valeurs de pression artérielle du patient remontées par le pilote du tensiomètre. Il peut également assister la production des rapports basés sur les données récoltées pendant la journée. Il peut aussi recevoir des alertes de chutes depuis un détecteur de chute et créer et propager les alarmes adéquates. La spécification du service de médiation comprend une interface fonctionnelle faite de trois opérations : récolterDonnées(), transférerDonnées() et traiterDonnées(). Les sous types de service peuvent redéfinir chacune des opérations en fonction des types de données qui sont reçus, traités et transférés par chaque service de médiation spécifique.
- Les services techniques fournissent des fonctions traverses à d'autres services. Par exemple, les planificateurs fournissent des fonctions de gestion du temps. Ils permettent de définir de façon statique ou dynamique la réalisation des actions, telles que le rappel d'un rendez-vous, le rappel de la prise des médicaments pour le patient et l'envoi d'un rapport médical.
- Les services métier permettent de fournir les services dédiés aux applications médicales concernant au patient surveillé. Par exemple, « AlarmeMédical » est chargée de centraliser les événements importants, de détecter de possibles alarmes, de leur donner des niveaux d'importance et de décider d'éventuelles actions à effectuer. Ce service permet de superviser l'état de ses occupants à l'aide des services de types capteur et médiation (température médicale, tension, etc.). Ce service est défini par l'interface incluant les opérations – recevoirMessage() et envoyerMessage() .
- Les actionneurs permettent de pouvoir influer sur les valeurs physiques de l'environnement ou sur l'état du patient surveillé. Ils permettent également de réagir avec l'environnement ou l'utilisateur final. Par exemple, les services d'alerte et les dispositifs d'alerte permettent d'avertir les informations aux utilisateurs finaux, à savoir l'occupant du bâtiment. On peut citer par exemple des services d'émission de mail, d'émission de SMS, d'affichage sur écran ou d'utilisation d'un haut-parleur.

2.3. Conclusion

Le méta-modèle de spécification de service est un des méta-modèles constituant le langage pour modéliser les artéfacts abstraits de domaine métier. Il permet de formaliser la description de type de composant abstrait métier sous la forme de spécification de service, qui représente une exigence de base ou une entité de connaissances métier. Ce méta-modèle permet de définir en deux formes de spécification de service visant à l'aspect fonctionnel et non-fonctionnel. De plus, ce méta-modèle permet de définir également la relation d'héritage entre des services afin de représenter telle relation entre des types de composants abstraits métier dans le domaine spécifique. Toutes spécifications de service restent générique et indépendant d'aucune technologie à services spécifique. Les différents types de spécifications de service fonctionnel ou non-fonctionnel sont enregistrés dans leur modèle de registre particulier, à savoir le registre de spécifications de service et le registre de spécifications de qualité de service.

3. Implantation de service

3.1. Définition

Une implantation de service est la description d'une implantation suivant le modèle à composant orienté service iPOJO. Nous avons en effet choisi iPOJO comme modèle d'implantation de référence pour ses propriétés de dynamisme et parce que iPOJO, tout comme notre modèle de spécification de service, est fondé sur des interfaces Java et des propriétés (spécifiées dans un fichier contenant des meta-données et nommé *metafile*). L'implantation d'une spécification de service est ainsi facilitée ; en particulier les expressions de dépendances entre services se font en utilisant des interfaces de type Java.

Nous aurions pu ne pas définir de modèle d'implantation pivot. En effet, une autre solution aurait été de décrire les spécifications de service en fonction de la technologie d'implantation cible. Les interfaces auraient été décrites différemment et des informations différentes seraient apparues (des points de contrôle en UPnP par exemple). Nous avons préféré garder un niveau de généricité important au niveau de l'architecture car, de fait, il importe peu à un développeur de savoir si un service est disponible en WS ou en UPnP. Seul le service rendu lui importe réellement dans son travail de composition.

Comme spécifié dans le méta-modèle ci-dessous, une implantation de service est définie par les éléments suivants :

- L'ensemble des ressources nécessaires à la description du service iPOJO, incluant les fichiers InterfaceFile, Manifest, BuildFile et PropertyFile,
- Le mode de communication avec le service iPOJO,
- Les propriétés nécessaires à la création et la configuration initiale des instances de ce service iPOJO,
- Eventuellement, la description de l'implantation du service dans une technologie différente de iPOJO.

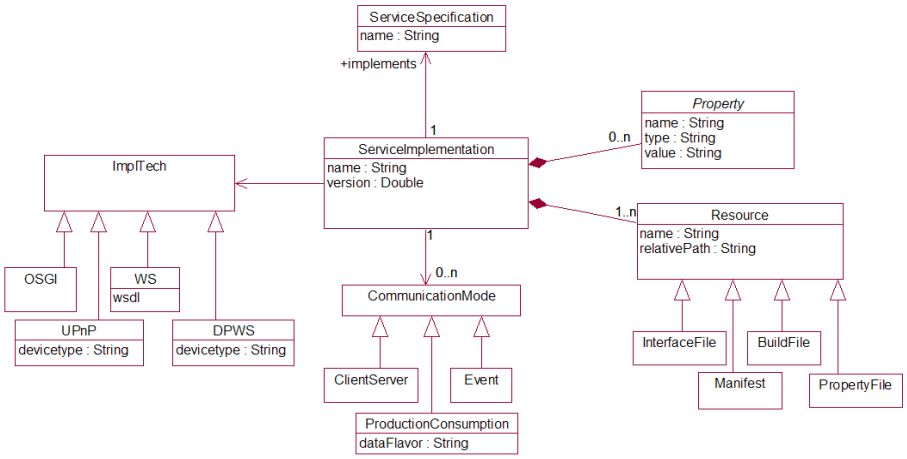


Figure 5.4. Méta-Modèle pour l'implantation de service

Les ressources manipulées par le modèle d'exécution iPOJO sont les suivantes :

- Le fichier *Interface* définit les interfaces effectives du composant iPOJO. Celles-ci doivent être en ligne avec les interfaces définies dans la spécification de services. Le fichier est en fait constitué de deux parties : les interfaces fonctionnelles et les interfaces s'adaptant au mode de communication choisi.
- Le fichier *Manifeste*. Ce fichier contient toutes les informations nécessaires à la configuration des instances. Par exemple, il peut définir les noms des méthodes permettant de démarrer ou de reprendre l'exécution du composant. Le fichier comprend également la définition des filtres régissant les dépendances du composant.
- Le fichier *PropertyFile*. Ce fichier décrit des propriétés spécifiques à iPOJO. Par exemple, l'implantation de service s'adaptant au mode de communication via l'événement par l'utilisation « Event Admin handler » doit spécifier la propriété de « topic » (voir la spécification de ce handler [106]) représentant une liste de messages transférés et d'autres propriétés nécessaires;
- Le fichier *BuildFile*. Celui-ci est généré automatiquement. pour gérer les dépendances de cette implantation de service au moment de la compilation de son développement.

Il est impératif à noter que les propriétés des technologies non iPOJO, par exemple, le profile standardisé d'un équipement UPnP, ne peuvent pas être dans la description de l'implantation de service iPOJO. Ces types des informations sont essentiels à manager par l'outil d'ingénierie du domaine métier.

Le mode de communication précise la façon d'interagir avec un service. C'est une information qui peut être importante lors de la sélection d'un service. Trois modes d'interaction sont possibles :

- *client/serveur* : l'interaction se fait au travers de l'invocation d'interfaces proposées par le service,
- *événement* : l'interaction se fait suivant le modèle de publication/souscription. Ce mode de communication est mis en application à l'aide du service Event Admin fourni par OSGi et du « Event Admin handler » en iPOJO [106] ;
- *production/consommation* : l'interaction se fait également suivant le modèle de publication/souscription mais, cette fois, les consommateurs de services publient des messages dans un canal spécifique. Celui-ci se charge de créer la liaison entre les consommateurs et les fournisseurs de services et d'administrer ces messages. Ce canal de communication est réalisé en

conformité avec la spécification du service « Wire Admin[44] ». Ce service permet de créer, détruire, recevoir et mettre à jour des liaisons de services de façon programmatique. Dès qu'une liaison de services est établie, un producteur peut envoyer des données à des consommateurs de services.

Le fait qu'iPOJO soit notre modèle pivot d'implantation ne signifie pas que tous les services sont implantés exclusivement en iPOJO. En effet, un service peut être effectivement implanté en utilisant les Services Web, UPnP, DPWS ou OSGi. Il s'agit alors de développer des ponts (*proxies*) faisant le lien entre le modèle iPOJO et les implantations effectives. Les *proxies* sont constitués d'une partie générique configurable, correspondant aux alignements techniques entre technologies, et d'une partie spécifique correspondant aux alignements syntaxiques. Un proxy spécifique est utilisé pour invoquer les interfaces d'un service particulier en fonction d'une technologie à services spécifique. Un proxy générique est en mesure d'interagir avec tous les services en fonction de la technologie à services donnée, alors qu'il ne peut pas invoquer une interface propre d'une implantation de service suivant cette technologie spécifique. Ce dernier est plutôt qu'un pont entre une technologie à services hétérogène et la technologie pivot iPOJO.

Pour les instances de service répondant à la même implantation de service dans une technologie donnée (des versions par exemple), le même *proxy* est utilisé.

Pour l'assemblage et la communication, tout service est vu comme un service iPOJO. Ils agissent comme si les autres services étaient tous implantés comme des services iPOJO. Cela signifie que, dans le code, les appels vers d'autres services sont des appels vers le registre iPOJO. En pratique, un service est implanté avec la technologie souhaitée. Le registre d'implantations de service manipule en fait deux types de services iPOJO :

- les services iPOJO fournissant une fonctionnalité bien définie. Ces services se retrouvent, en particulier, au cœur des architectures. Ce sont eux qui implantent les éléments de contrôle régissant le comportement des applications.
- les services iPOJO fournissant des ponts vers des services implantés avec d'autres technologies. Ces services sont des *proxies* spécifiques ou génériques, qui permettent d'intégrer des services basés sur des descriptions hétérogènes.

3.2. Exemple

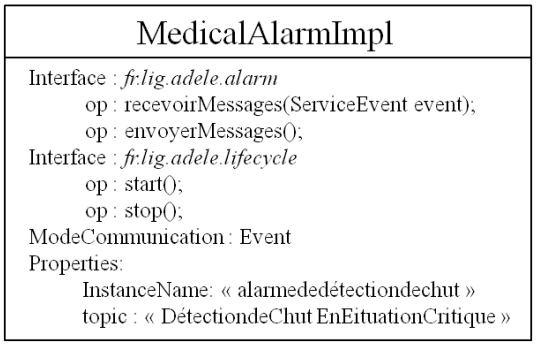


Figure 5.5. L'exemple de la description d'une implantation de service

[44] OSGi Service Platform Release 4 V.4.1-108.h ttp://www.osgi.org/download/r4v41/

La spécification de service « *Alarme* » introduit précédemment est développé par le composant orienté service iPOJO « AlarmServiceImpl » s'adaptant au mode de communication de l'événement à l'aide de « Event Admin Handler ». La description de l'implantation de « AlarmServiceImpl » a développé une interface avec le nom qualifié « fr.lig.adele.alarm » réalisant les opérations recevoirMessages(ServiceEvent event) et envoyerMessages() et une interface liant à technologie iPOJO avec les opérations pour gérer le cycle de vie de service, telles que start(), stop(). Cette implantation a spécifié la propriété « InstanceName » avec la valeur « alarmededétectiondechut» et la propriété « topic » avec la valeur « DétectiondeChut EnSituationCritique ». Il signifie qu'une instance de service d'alarme en tant que consommateur de service est identifiée par le nom « alarmededétectiondechut » et recevait seulement les messages envoyés dans le topic « DétectiondeChut EnSituationCritique ».

3.3. Conclusion

Le métamodèle d'implantation de service est utilisé pour décrire les entités logicielles existantes dans le domaine métier. Une description d'implantation de service permet de spécifier des propriétés servant à la configuration lors de la création de ses instances de service. Elle donne également les informations permettant à la machine d'exécution de gérer concrètement le cycle de vie de ses instances de service lors de l'exécution.

Chaque spécification de service peut être réalisée par une ou plusieurs implantations de service s'adaptant aux technologies à services hétérogènes ou aux différents modes de communication. Dans notre cas, toutes les implantations de service sont mises en œuvre en iPOJO - la technologie pivot - comme le pont d'invocation de services exécutables hétérogènes.

Enfin, le modèle du registre d'implantations de service peut être référencé par le modèle d'architectures de référence du domaine, tout en référant au modèle du registre de spécifications de service. De plus, toutes les entités logicielles des implantations de service réelles sont déposées dans une base de donnée distribuée – laquelle peut être accédée de façon locale ou à distance via une adresse ***URI*** du registre d'implantations de service.

4. Instance de service

Les instances de service sont les éléments concrets utilisés pour constituer les applications à services sur la plate-forme cible lors de l'exécution. Les instances de service sont réifiées uniquement lors de l'exécution. Une instance de service est créée par l'implantation de service déployée sur la plate-forme d'exécution. Il est important de noter que le cycle de vie d'une instance de service ne dépend pas de l'application à services qui l'utilise lors de l'exécution. Cependant, l'exécution de l'application à services est contrainte par le cycle de vie des instances de services dont elle est composée.

Une instance de service correspond à une configuration des propriétés caractérisant l'implantation de service correspondante. Elle permet ainsi de fournir concrètement un service. L'instance de service peut être configurée de façon statique ou dynamique, lors de l'ingénierie applicative ou de l'exécution. De fait, les instances de services sont les entités logicielles et exécutables qui sont capable de constituer l'application à services finale lors de l'exécution.

5. Architecture

5.1. Définition

Un point majeur de notre approche est que nous définissons les architectures de référence de l'ingénierie domaine et les architectures applicatives de l'ingénierie applicative suivant le même langage. Ainsi, nous parlerons simplement *d'architecture* dans cette section. Une architecture décrit la structuration d'une famille d'applications en un ensemble de spécifications de service reliées par des connecteurs. Une architecture comprend des parties stables et des points de variation explicites permettant de capturer les différences architecturales entre applications. Comme détaillé ci-après, les points de variation prennent différentes formes.

Plus précisément, une architecture comprend les éléments suivants :

- **Des spécifications de service** décrites suivant le formalisme présenté précédemment. Une spécification de service peut être associée à une ou plusieurs implantations de service. Ceci est une première forme de variabilité. Si aucune implantation n'est précisée, alors toute implantation de service répondant aux contraintes de la spécification est valable, quelle que soit la technologie finale d'implantation. Sinon, la spécification de service sera implantée suivant l'une des implantations mentionnées. S'il n'y en a qu'une, alors il n'y a plus de variabilité concernant la réalisation de l'implantation.
- **Des connecteurs (ou *bindings*)** décrivant les connections entre spécifications de service. Les connecteurs fournissent les informations nécessaires à l'exécution pour gérer correctement les liens entre instances d'implantations de service (réalisant les spécifications de service).
- **Des points de variation (ou *VariantPoints*)** décrivant les variations possibles entre applications. Ces points de variation se situent au niveau des connecteurs entre spécifications de services. De plus, chaque point de variation doit être lié un type de variabilité selon la logique, telles que « optionnel », « multiple », « alternatif » et « indispensable ».
- **Des contraintes** exprimées, par exemple, en OCL au niveau des connecteurs entre spécifications de services.

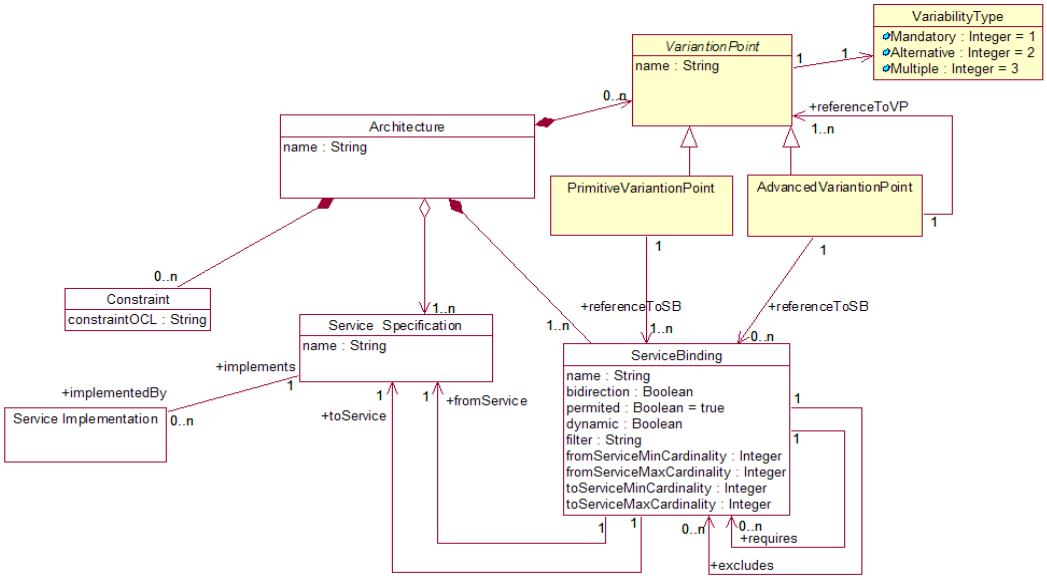


Figure 5.6. Méta-modèle pour l'architecture de référence

Un connecteur de services, appelé « ServiceBinding » dans le métamodèle précédent, est utilisé pour définir la relation de dépendance et les règles d'interaction entre deux spécifications de services. Il est caractérisé par les propriétés suivantes :

- la propriété ***bidirection*** définit la direction d'invocation des services liés. Il s'agit ici de déterminer la direction du transfert de messages ou la direction de l'invocation entre deux services se liant via ce connecteur de services ;
- la propriété ***filter*** spécifie une contrainte suivant la spécification ***LDAP*** [45] permettant de sélectionner des instances de services à l'exécution ;
- les propriétés de cardinalité représentent le nombre des instances de services autorisées à se lier à chacun des services impliqués dans la connexion. Pour le service appelant, ce nombre d'instances doit se situer entre ***fromServiceMinCardinality*** et ***fromServiceMaxCardinality.*** Pour le service appelé, ce nombre d'instances doit se situer entre ***toServiceMinCardinality*** et ***toServiceMaxCardinality***.
- la propriété ***dynamic*** indique si la sélection d'un service implantant la spécification de service défini dans l'architecture peut être pris au moment de l'exécution. Il s'agit concrètement d'un booléen. Cette propriété est utilisée en accord avec les propriétés de cardinalité présentées ci-dessus. Si elle est assignée à « vrai », alors une instance du service appelant peut se lier à une ou plusieurs instances du service appelé de façon dynamique. Le nombre des instances du service appelé doit être une valeur valide dans l'intervalle de cardinalité défini. Si elle est assignée à « faux », alors une instance du service appelant ne peut plus se lier à un nombre non modifiable d'instances du service appelé lors de l'exécution. Le nombre des instances du service appelé doit être une valeur valide dans l'intervalle de cardinalité défini ;
- la propriété ***permitted*** définit l'autorisation d'invocation entre les services liés. Par défaut, la valeur de cette propriété est assignée à « vrai ». Lorsqu'elle est affectée à « faux », la liaison entre services liés est interdite. Cette propriété est utilisée dans des cas exceptionnels. Prenons l'exemple du connecteur de services appelé ***AtoB*** liant les spécifications de service ***A*** et ***B***. Tous les services héritant du ***A*** peuvent interagir avec le service ***B*** et ses sous types. Cependant, dans un cas exceptionnel, un service ***A1*** héritant du ***A*** n'est pas en mesure d'interagir avec ***B*** en fonction d'une contrainte sémantique. Dans ce cas, nous affectons la propriété ***permitted*** avec la valeur en « ***faux*** ». Cela permet d'éviter d'énumérer tous les connecteurs de services pour lier les autres services héritant de ***A*** (différent ***A1***) vers le service ***B*** ;
- la propriété ***requires* et *excludes*** définit la dépendance entre points de variations. Par exemple, la sélection d'un service à un point de variation peut requérir ou exclure la sélection d'un autre service sur un autre point de variation.

Nous définissons deux types de points de variation :

- le **point de variation primitif**, appelé *PrimitiveVariationPoint* dans le métamodèle, permet de définir les variations possibles parmi une ou plusieurs connexions entre une spécification de service et ses dépendances relatives. Ces dépendances relatives sont groupées en tant que ensemble des variantes associées à ce point de variation ;
- le **point de variation avancé**, appelé *AdvancedVariationPoint* dans le métamodèle, permet de définir les variations possibles parmi soit un ou plusieurs points de variation, soit parmi une ou plusieurs connexions et un ou plusieurs points de variation. Cela conduit à une structure de dépendance entre points de variation et connexions de spécifications de service. Par exemple, la fonctionnalité d'une spécification de service plus complex peut être fournie par deux spécifications de service atomiques. Cette fonctionnalité est requis par le même service dépendant. Dans la structure architecturale, les deux services atomiques peuvent être groupés en

[45] *LCDP :* Lightweight Directory Access Protocol, http://www.faqs.org/rfcs/rfc1960.html

tant que variantes associéee à un point de variation primitif selon la logique « indipensable ». Un point de variation avancé est également défini afin décrire un relation selon la logique « alternatif » parmi ce point de variation primitive et la connexion de la spécification de service plus complexe et son dépendant.

- Dans notre approche, un point de variation est lié aux connexions à partir d'une spécification de service avec plusieurs spécifications de service. Il indique qu'une connexion peut être « indispensable », « alternative », « multiple » ou « optionnelle ». Ce dernier type de variabilité est défini à travers la définition de propriétés de cardinalité. Le nombre minimal d'instances pouvant être connectées est fixé à zéro ([***0..1***] ou [***0..n***]), ce qui signifie qu'un service n'est pas nécessaire à l'exécution.

5.2. Variabilité au sein d'une architecture

La flexibilité des architectures est apportée par les notions de spécifications de service et de points de variation qui permettent de faire des adaptations aux tous moments du cycle de vie. Chaque mécanisme fournit en fait différents types de variabilité. Nous précisons, ci-après, les différents types de variabilité au sein de l'architecture :

- La **spécification de service** donne lieu à des choix d'implantations de service lors de l'ingénierie applicative ou lors de l'exécution. Ceci peut apporter des comportements différents, adaptés au contexte courant (en termes de besoins exprimés par l'utilisateur et par l'état de l'environnement), tout en respectant les fonctionnalités attendues ;
- L'**implantation de service** donne lieu à des choix d'instances de service lors de l'exécution en fonction des propriétés configurées au préalable. Ceci permet de s'adapter dynamiquement au contexte courant en termes de disponibilité de service.
- La **relation d'héritage entre spécificatio**ns de service apporte un autre type de variation. Cette relation soulage les architectures applicatives d'une partie des restrictions fortes liées en conformité avec l'architecture de référence, puisque le service héritant peut reprendre implicitement toutes les dépendances du service hérité. Cela donne lieu à des choix d'un service et ses services héritant, lors de la spécialisation d'une architecture applicative ou lors de l'exécution ;
- Les **points de variation** permettent d'exprimer explicitement des variantes architecturales sur le nombre de spécification de service en relation. Ces variantes sont exploitées lors de la sélection des implantations de services.
- Les **cardinalités au niveau des connexions** permettent d'introduire de façon explicite des contraintes sur le nombre d'instances pouvant être connectées, de façon dynamique, lors de l'exécution.
- La **propriété « *filter* » au niveau des connexions** entre spécifications de service permet de définir diverses formes de restrictions utilisées lors de la création ou sélection d'instances à l'exécution. Par exemple, dans des applications de télécommunication, on peut imposer des contraintes sur les fournisseurs de certains services, tels que Orange, SFR ou Bouygues. Un filtre peut également prohiber l'établissement des connexions entre certaines instances.

5.3. Raffinement des architectures de référence

L'utilisation d'une unique architecture de référence s'applique à des lignes de produits logiciels de petite taille où la qualité fonctionnelle et les besoins de produits individuels sont très similaires [105]. Il est en effet possible dans de telles situations de développer des applications logicielles à partir d'une seule architecture commune. Il faut néanmoins comprendre que, suivant cette approche, on ne peut pas définir des attributs de qualité ou des fonctionnalités pour un sous ensemble de produits. Un aspect

commun ne peut être défini que pour tous les logiciels de la ligne de produits. Il est également clair qu'une architecture de référence fournit un canevas assez rigide : lors de l'ingénierie applicative, les adaptations possibles sont limitées et très encadrées (au niveau des points de variations, essentiellement).

Pour traiter ces limitations, il est possible de raffiner une architecture de référence et de créer des architectures de référence plus spécifiques. Ces architectures traitent un sous ensemble de produits logiciels de la famille initiale en adaptant certains aspects architecturaux, tels que les contraintes, les dépendances, les attributs de qualité. Elles offrent un meilleur niveau d'information pour guider la conception et le développement de ces applications plus spécifiques.

Comme indiqué par la figure ci-dessous, nous proposons d'organiser les architectures de référence en une hiérarchie. L'architecture de référence située au sommet de la hiérarchie est l'architecture la plus générique. Elle peut même, dans certains cas, être abstraite et ne jamais être instanciée. Elle est valable pour toutes les applications de la famille logicielle considérée. Les architectures de référence situées plus bas dans la hiérarchie traitent des sous ensembles de la famille logicielle considérée. Elles sont de plus en plus raffines au fur et à mesure que l'on descend dans la hiérarchie.

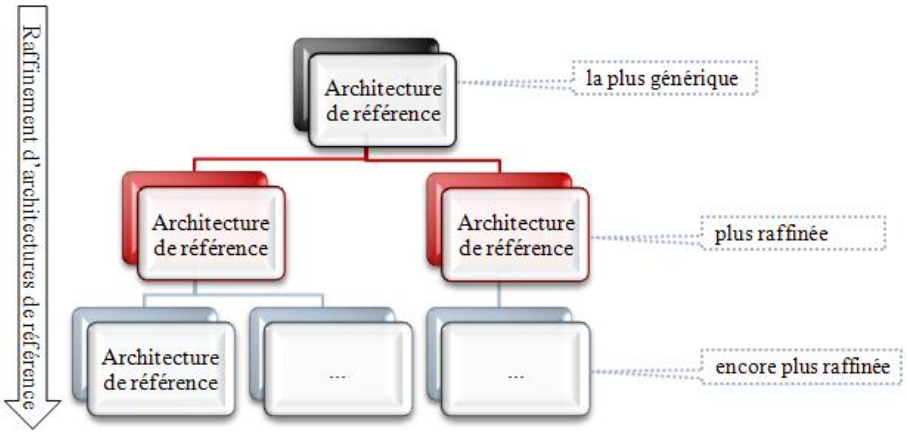


Figure 5.7. Raffinement d'architectures de référence dans un domaine

5.3.1. Règle du raffinement d'architectures de référence

Nous avons défini un ensemble de règles permettant de raffiner une architecture de référence. Tout d'abord, il est possible de spécialiser une spécification de service. Une spécialisation peut se faire en ajoutant des propriétés de qualité, en attribuant des valeurs fixes à des attributs, en ajoutant des opérations. Cela peut amener à créer de nouvelles dépendances potentielles puisque les dépendances entre spécifications de services s'héritent.

Ajouter de nouvelles dépendances permet de créer les nouvelles connexions entre les dépendances d'une spécification de service et ses sous-types, lors du remplacement de cette spécification de service par ses sous-types du service. Comme l'exemple illustré dans la figure 5.8, ci-après :

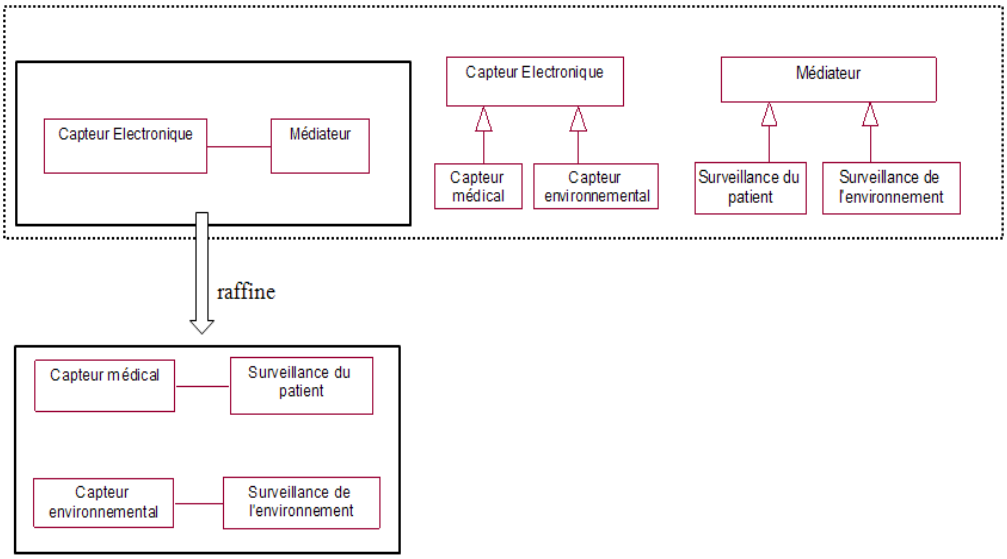


Figure 5.8. Création de nouvelles dépendances entre spécifications de service.

La structure en haut à gauche de la figure 5.8 est composée de « Capteur Electronique » et « Médiateur ». Les deux services peuvent s'interagir. En haut à droite de la figure, les services « Capteur médical » et « Capteur environnemental » héritent de « Capteur Electronique » ; « Surveiller le patient » et « Surveiller l'environnement » héritent de « Médiateur ». Etant donné que des spécifications de service spécialisées peuvent reprendre les dépendances du père, la connexion entre « Capteur médical » et « Surveiller le patient » et la connexion entre « Capteur environnemental » et « Surveiller l'environnement » peuvent être créées dans une architecture raffinée.

Il est également possible de définir de nouvelles connexions de spécifications de service. Nous pensons ici au cas particulier où une nouvelle connexion de spécifications de service est définie pour prohiber explicitement une dépendance à partir d'un service spécialisé avec les dépendances de services de son père, qui sont reprises par le service spécialisé de façon implicite.

Il est également possible de raffiner une architecture de référence au niveau de ses points de variation en éliminant une ou plusieurs variantes, en ajoutant de nouvelles variantes sur les points de variation ouverts ou en ajoutant ou en éliminant des dépendances entre certains points de variation. Ces règles mentionnées sont faciles à mettre à exécution pour le raffinement. On peut aussi être amené à éliminer, ajouter et remplacer certains points de variation lors de l'ajout de nouvelles spécifications de service et de ses dépendances.

Nous avons défini les règles suivantes de façon à raffiner une architecture de référence au niveau de ses points de variation :

- **La règle de l'élimination d'un point de variation** permet de supprimer un point de variation au sein d'une architecture de référence. Il s'agit de faire un choix entre les variantes associées à ce point de variation. Cependant, la règle ne permet pas d'éliminer un point de variation selon le type de variabilité « indispensable ». Par exemple, dans la figure 5.9, la structure plus générique (en haut à gauche de la figure) est composée de la connexion entre « Service Métier » et « Actionneur » ainsi qu'un point de variation « pvp_o4 » suivant la logique « optionnel ». La relation d'héritage entre « Service Métier » et son fils « Service Médical » est aussi représentée en haut à droite de la figure. Nous pouvons éliminer le point de variation « pvp_o4 » afin de déduire la structure plus raffinée (illustrée en bas de la figure). Ce raffinement signifie que le « Service Métier» doit interagir avec, au moins, un « Actionneur » dans la structure plus raffine

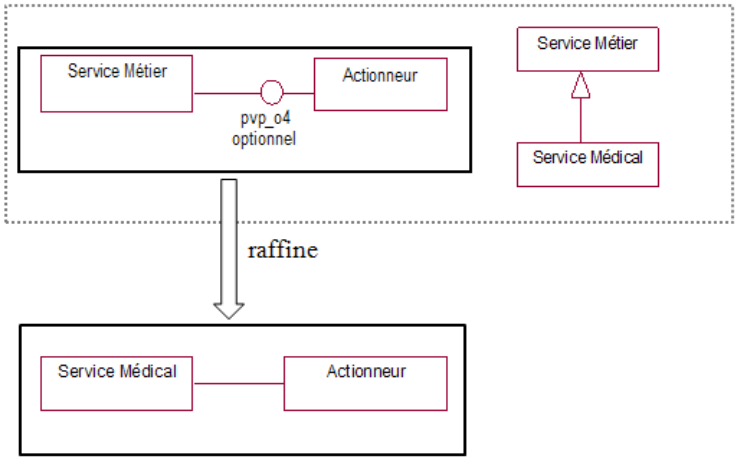


Figure 5.9. L'exemple de l'élimination d'un point de variation

- **La règle de l'ajout d'un point de variation** permet d'ajouter un point de variation au sein d'une connexion entre spécifications de service. Elle permet également d'ajouter un ou plusieurs points de variation afin de réaliser la décomposition d'un point de variation. Cette règle définit que le point de variation selon la logique « optionnel » ne peut pas être ajouté au sein d'une connexion des spécifications de service. Nous reprenons l'exemple, ci-dessus, afin de raffiner l'architecture de référence par l'ajout d'un point de variation selon la logique « multiple », au niveau de la connexion entre « Service Médical » et « Actionneur ». Par exemple, la structure plus générique (en haut à gauche de la figure 5.10) est composée de la connexion entre « Service Médical » et « Actionneur ». La partie en haut à droite de la figure illustre également le service « Actionneur » spécialisé par les services « Dispositif d'alerte », « Contrôleur médical » et « Service d'alerte ». Cette la relation d'héritage peut amener à créer de nouvelles dépendances ; il s'agit des connexions entre « Service Médical » vers « Dispositif d'alerte », « Contrôleur médical » ou « Service d'alerte ». Le point de variation « pvp1 » est ajouté et défini selon le type de variabilité « multiple ». Il permet de décrire la variabilité parmi les trois connexions précédentes selon la logique « multiple ». Ce raffinement signifie que le « Service Médical » peut interagir avec un ou plusieurs des trois sous-types de « Actionneur ».

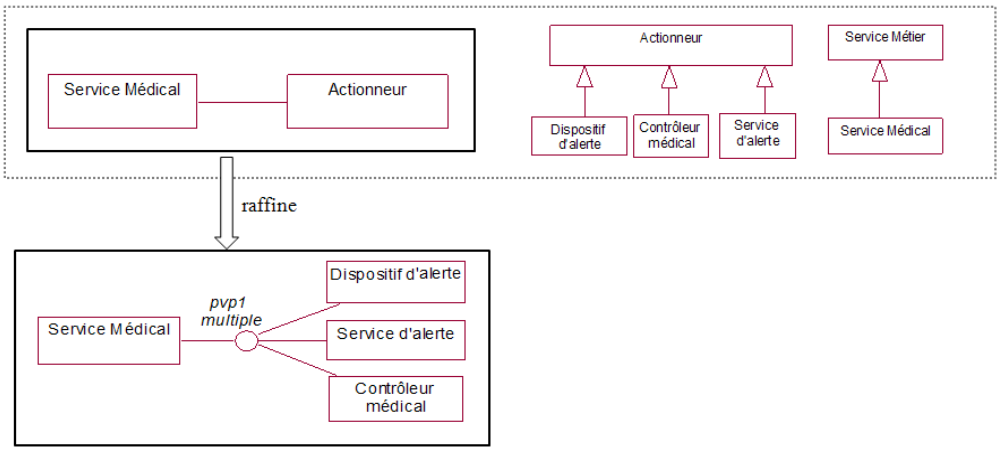


Figure 5.10. L'exemple de l'ajout d'un point de variation

- **La règle de la décomposition** d'un point de variation permet de décomposer un point de variation en plusieurs points de variation. Dans certains cas, les variantes d'un point de variation doivent être regroupées, en fonction de la logique métier, dans plusieurs ensembles des variantes. Un ou plusieurs point(s) de variation peuvent être décomposé(s) afin de lier aux ensembles des variantes regroupées comme illustrée par la figure 5.11. Ainsi, les points de variation décomposés peuvent décrire la variation entre les connexions de spécification de service et d'autres points de variation. Le nombre des points de variation à ajouter dépend du nombre des groupes classifiant les variantes. De plus, chaque point de variation ajouté ne peut pas modifier la définition du type de variabilité du point de variation décomposé. Nous réutilisons l'exemple précédent afin de montrer la décomposition du point de variation « pvp1 ». Ses variantes « Dispositif d'alerte », « Service d'alerte » et « Contrôleur médical » sont regroupées dans deux ensembles selon la logique métier. Le « pvp1 » est décomposé en deux points de variation « pvp1 » et « avp1 ». Ils se lient respectivement aux deux ensembles des variantes. Les variantes liées au « pvp1 » sont « Dispositif d'alerte », « Service d'alerte ». La variante liée au « pvp2 » est « Contrôleur médical ». Finalement, le type de variabilité selon la logique « multiple » n'est pas modifié pour les deux points de variation.

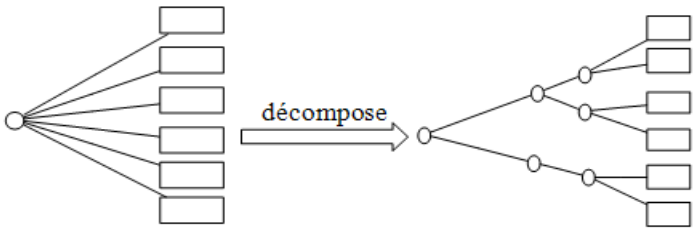


Figure 5.11. La forme générale de la décomposition d'un point de variation

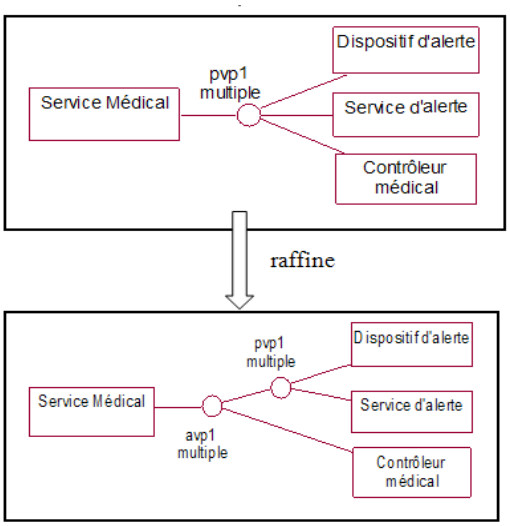


Figure 5.12. Décomposition d'un point de variation afin de regrouper les variantes relatives

- **La règle de la modification d'un point de variation** permet de redéfinir la logique du type de variabilité d'un point de variation. Comme nous avons représenté précédemment, le type de variabilité peut être défini en quatre logiques différentes, à savoir « optionnel », « multiple », « alternatif » et « indispensable ».

Nous avons utilisé la théorie des ensembles afin de formaliser la règle de redéfinition de la logique du type de variabilité. Les nomenclatures ci-dessous permettent de décrire la variabilité d'un point de variation au sein d'une architecture de référence indépendamment des termes du domaine :

- *vp* représente le point de variation à modifier,
- n représente le nombre des variantes associées au point de variation *vp*,
- v_i représente une variante associée au point de variation *vp*,
- vp_{var} représente l'ensemble des variantes associées au *vp*, $vp_{var} = \{i \in [1..n] \mid v_1, v_2, \ldots v_i, \ldots v_n\}$
- $\mathcal{P}(vp_{var}, logique)$ représente l'ensemble de toutes les combinaisons des variantes possibles en fonction de la *logique* du type de variabilité pour le point de variation à modifier. Ils sont repectivement définis ci-dessous :

1. $\mathcal{P}(vp_{var}, \text{optionnel})$ est **l'ensemble des parties de vp_{var}**,

   si $k \in [0..n]$, $\mathcal{P}_k(vp_{var})$ présente l'ensemble des k-combinaisons de vp_{var}

   alors, $\mathcal{P}(vp_{var}, \text{optionnel}) = \{\mathcal{P}_0(vp_{var}) \cup \mathcal{P}_1(vp_{var}) \cup \mathcal{P}_k(vp_{var}) \ldots \cup \mathcal{P}_n(vp_{var})\}$

   Par exemple:

   si $vp_{var} = \{v_1, v_2\}$, $\mathcal{P}_0(vp_{var}) = \{\emptyset\}$, $\mathcal{P}_1(vp_{var}) = \{v_1\}, \{v_2\}\}$, $\mathcal{P}_2(vp_{var}) = \{\{v_1, v_2\}\}$
   alors, $\mathcal{P}(vp_{var}, \text{optionnel}) = \{\{\emptyset\}, \{v_1\}, \{v_2\}, \{v_1, v_2\}\}$

2. si $k \in [1..n]$, $\mathcal{P}_k(vp_{var})$ présente **l'ensemble des k-combinaisons de vp_{var}**

   alors, $\mathcal{P}(vp_{var}, \text{multiple}) = \{\mathcal{P}_1(vp_{var}) \cup \mathcal{P}_k(vp_{var}) \ldots \cup \mathcal{P}_n(vp_{var})\}$

   Par exemple:

   si $vp_{var} = \{v_1, v_2\}$, alors $\mathcal{P}(vp_{var}, \text{multiple}) = \{\{v_1\}, \{v_2\}, \{v_1, v_2\}\}$

3. si $\forall x \in [1..n]$, $P_x(vp_{var})$ présente **une partition de vp_{var}**,

   $$P_x(vp_{var}) \neq \emptyset, \quad \bigcup P_x(vp_{var}) = vp_{var},$$

   $$\forall x, y \in [1..n], \; x \neq y, \; P_x(vp_{var}) \cap P_y(vp_{var}) = \{\emptyset\},$$

   alors, $\mathcal{P}(vp_{var}, \text{alternatif}) = P_x(vp_{var})$

   Par exemple:

   si $vp_{var} = \{v_1, v_2\}$,
   alors soit $P(vp_{var}, \text{alternatif}) = \{\{v_1\}\}$, soit $P(vp_{var}, \text{alternatif}) = \{\{v_2\}\}$

4. $\mathcal{P}(vp_{var}, \text{indispensable})$ correspond à l'ensemble $\{vp_{var}\}$

   Par exemple:

   si $vp_{var} = \{v_1, v_2\}$, alors $P(vp_{var}, \text{indispensable}) = \{\{v_1, v_2\}\}$

Il en découle les relations suivantes :

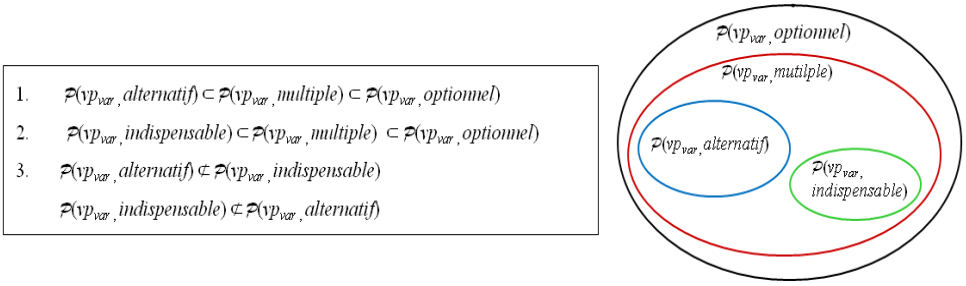


Figure 5.13. La relation des ensembles des combinaisons des variantes possibles selon la logique du type de variabilité

Les règles de la modification de la *logique* du type de variabilité pour *vp* sont définies ci-après :

> Soit *logiqueA*, *logiqueB* deux différentes *logique* du type de variabilité. La règle de modification autorise que la *logiqueA* soit modifié par la *logiqueB* si et seulement si
>
> $$\mathcal{P}(vp_{var}, logiqueB) \subset \mathcal{P}(vp_{var}, logiqueA)$$

Dans le contexte du raffinement de l'architecture, cette formule signifie que $\mathcal{P}(vp_{var}, logiqueB)$ permet de donner moins de combinaisons possibles entre les variantes associées au *vp* selon la *logiqueB* par rapport au $\mathcal{P}(vp_{var}, logiqueA)$. En d'autres termes, elle présente que le *vp* selon $\mathcal{P}(vp_{var}, logiqueB)$ définit plus de contraintes pour l'architecture de référence. La figure 5.13 illustre la relation des ensembles des combinaisons des variantes possibles selon la *logique* du type de variabilité *vp*.

Cette règle découle de l'idée intuitive que la *logique* du type de variabilité *vp* peut être modifiée par une *logique* imposant plus de contraintes. Par conséquent, le nombre des combinaisons de variation possibles du *vp* sont définitivement réduites, car la *logique* du type de variabilité *vp* est redéfinie.

> An contraire, cette modification ne doit pas autoriser la redéfinition de la *logiqueA* par la *logiqueB* si
>
> $$\mathcal{P}(vp_{var}, logiqueB) \not\subset \mathcal{P}(vp_{var}, logiqueA).$$

Nous reprenons l'exemple illustré précédemment dans la figure 5.12 afin de raffiner continuellement l'architecture de référence par la modification de la *logique* du type de variabilité du point de variation « avp1 ».

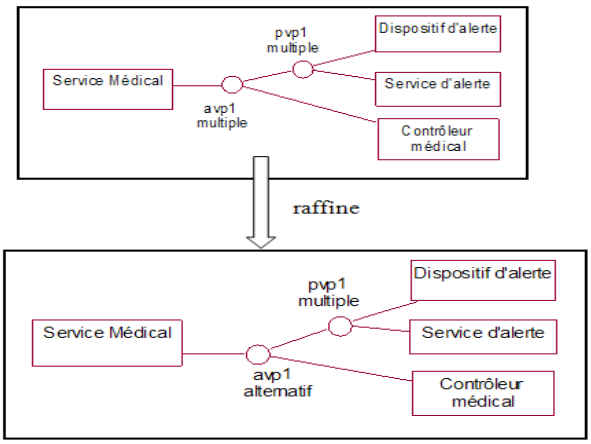


Figure 5.14. L'exemple de la modification du type de variabilité d'un point de variation

Etant donne que l'ensemble $\mathcal{P}(avp1_{var}, alternatif)$ est inclut dans $\mathcal{P}(avp1_{var}, multiple)$, la règle de la modification permet de modifier la logique « multiple » par la logique « alternatif » au point de variation « avp1». Cependant, si la logique du « avp1 » est définie initialement par « alternatif », alors il ne peut être pas modifié par d'autres *logiques* du type de variabilité. Le résultat est finalement illustré dans la figure 5.14.

5.4. Exemple

Nous présentons dans cette section trois architectures de référence de façon à illustrer les règles du raffinement d'une architecture de référence. La première architecture est raffinée par les deux autres, de manière à atteindre différents objectifs. Ces architectures concernent le domaine de l'informatique ambiante dans le bâtiment, et plus précisément celui de l'hospitalisation à domicile. Le but de ces applications est de récolter des données issues de divers capteurs médicaux et environnementaux de façon à surveiller continuellement l'état d'un patient à son domicile. Des modules logiciels spécifiques sont utilisés pour appliquer des opérations de médiation de données (filtrage, corrélation, associations, …). Ces modules adressent ensuite les données formatées à des services dits « métiers ».

L'architecture de référence, très générique, de ce domaine est présentée sur la figure 5.15. Elle est, conformément à sa définition, constituée de spécifications de service, de connecteurs de service et de points de variation. Les spécifications de service sont les suivantes :

- « Capteur Electronique » représente tous les équipements électroniques potentiellement disponibles permettant de récolter des informations sur l'état du patient surveillé ou de l'environnement et de les rendre disponibles ;
- « Médiateur » représente les divers modules logiciels chargés de récolter et de traiter les données issues des capteurs médicaux et environnementaux ;
- « Service Métier » utilise les données préparées par les médiateurs afin d'effectuer certaines opérations propres au domaine et, ainsi, de fournir un service aux malades ou aux personnes surveillantes ;
- « Service Technique » fournit des propriétés techniques transverses telles que la persistance de données ou de planification d'action ;
- « Actionneur » permet de pouvoir influer sur les valeurs physiques de l'environnement ou sur l'état du patient.

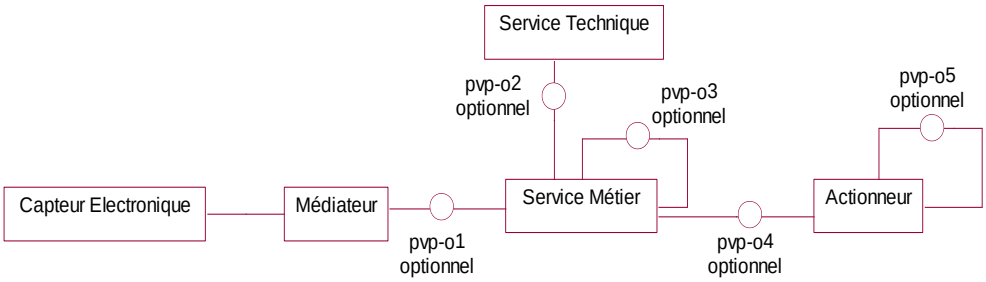


Figure 5.15. L'architecture de référence la plus générique dédié au domaine de l'hospitalisation à domicile

Cette architecture de référence définit également quatre points de variation :

- « pvp-o1 », selon la logique « optionnel », sur la connexion entre « Médiateur » et « Service Métier » indique que « Médiateur » peut interagir avec « Service Métier » en cas de besoin ;
- « pvp-o2 », selon la logique « optionnel », sur la connexion entre « Service technique » et « Service Métier » indique que « Service technique » peut interagir avec « Service Métier » en cas de besoin ;
- « pvp-o3 », selon la logique « optionnel », sur la connexion réflexive du « Service Métier », indique que une « Service Métier» peut interagir avec les autres sous-types de ce service ou lui-même en cas de besoin ;
- « pvp-o4 », selon la logique « optionnel », sur la connexion entre « Service Métier » et « Actionneur » indique que « Service Métier » peut interagir avec « Actionneur» en cas de besoin ;
- « pvp-o5 », selon la logique « optionnel », sur la connexion réflexive du « Actionneur » indique qu'un « Actionneur» peut interagir avec les autres sous-types de ce service ou lui-même en cas de besoin.

La **première architecture de référence raffinée** de ce domaine est présentée sur la figure 5.16. Elle est partagée par le sous-ensemble des applications métier dont le but est de surveiller ou d'influer l'état du patient.

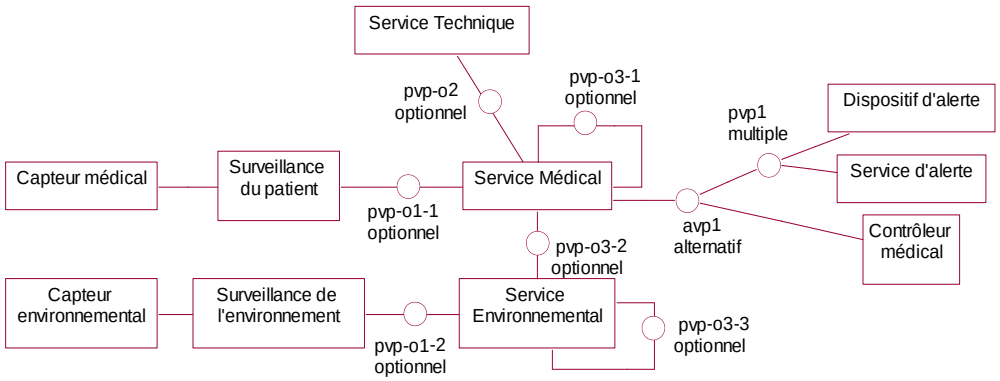


Figure 5.16. Architecture de référence raffinée dédiée aux applications médicales

Les spécifications de service constituant cette architecture sont les suivantes :

- « Capteur médical » représente tous les équipements médicaux potentiellement disponibles permettant de récolter des informations sur l'état du patient surveillé et de les rendre disponibles pour l'application ;
- «Capteur environnemental» représente tous les équipements électroniques communicants permettant de caractériser le contexte physique du domicile ;
- « Surveillance du patient » héritant de « Médiateur » permet de traiter les données récoltées à partir des capteurs médicaux afin de surveiller l'état du patient ;
- « Surveillance de l'environnement» héritant de « Médiateur » permet de traiter les données issues des capteurs environnementaux ;
- « Service Médical » héritant de « Service Métier » utilise les données préparées par ces médiateurs précédents afin de fournir les informations d'alerte aux utilisateurs, ou d'effectuer certaines opérations pour influer sur l'état du patient surveillé ;

- « Service Environnemental » héritant de « Service Métier » utilise les données préparées par ces médiateurs afin de fournir les informations concernant l'état de l'environnement, ou d'ajuster l'état de l'environnement pour s'adapter une description initiale ;
- « Service Technique » fournit des propriétés techniques transverses telles que la persistance de données ou de planification d'action ;
- « Dispositif d'alerte » héritant de « Actionneur » permet de pouvoir émettre l'information d'alerte aux utilisateurs via un dispositif électronique ;
- « Contrôleur médical » héritant de « Actionneur » permet de pouvoir influer sur l'état du patient.
- « Service d'alerte » héritant de « Actionneur » permet de pouvoir émettre l'email ou le message afin d'informer les utilisateurs.

Cette architecture de référence dédiée aux applications médicales est raffinée par l'adoption des règles présentées précédemment :

- **l'ajout des nouvelles dépendances** : les connexions réflexives du « Service Médical » et du « Service Environnemental » sont ajoutées dans l'architecture de référence, puisque les deux spécifications de service héritent de « Service Métier ».
- **la décomposition d'un point de variation** : le point « pvp-o1 » sur la connexion entre « Médiateur » et « Service Métier » est décomposé en « pvp-o1-1 » et « pvp-o1-2 », toujours selon la logique « optionnel ». Le point « pvp_o3 » est décomposé en « pvp-o3-1 », « pvp-o3-2 » et « pvp-o3-3 » selon toujours la logique « optionnel ».
- **l'élimination d'un point de variation** : le point de variation « pvp-o2 » sur la connexion entre « Service technique » et « Service Métier » est éliminé. Cela signifie que « Service technique » est un élément nécessaire pour la construction des applications médicales.

Les points de variation topologique « pvp1 » et « avp1 » de cette architecture de référence sont représentés comme les exemples de la mise en œuvre des règles de raffinement. Ils sont illustrés dans les figures 5.10, 5.12 et 5.14.

La **deuxième architecture de référence raffinée** est présentée sur la figure 5.17. Elle est partagée par le sous-ensemble des applications de ce domaine dont le but est de surveiller ou ajuster l'état de l'environnement à domicile.

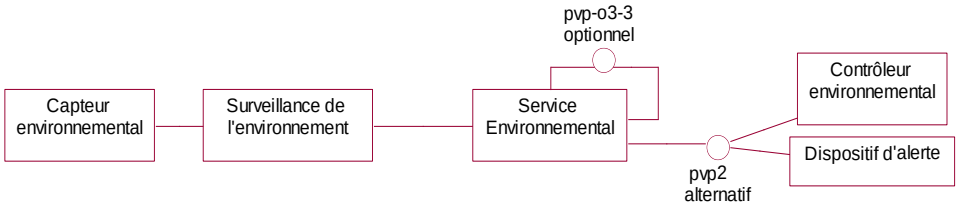


Figure 5.17. Architecture de référence raffinée dédiée aux applications environnementales

Cette architecture de référence raffinée, est constituée des spécifications de service « Capteur environnemental», « Surveiller l'environnement», « Service Environnemental », « Dispositif d'alerte » et « Contrôleur Environnemental ». Cette dernière spécification de service héritant de « Actionneur » permet de pouvoir influer les valeurs sur l'état de l'environnement à domicile afin de l'ajuster.

Cette architecture de référence dédiée aux applications environnementales est raffinée par l'adoption des règles présentées précédemment :

- En adoptant la règle de l'ajout des nouvelles dépendances, la connexion réflexive de « Service Environnemental » et la connexion entre «Service Environnemental » et « Actionneur » sont ajoutées dans cette architecture de référence.
- En adoptant la règle de l'élimination, le point de variation « pvp-o4 » de type « optionnel » sur la connexion entre « Service Environnemental » et « Actionneur » est éliminé.
- En adoptant la règle de l'ajout des nouvelle dépendances, les connexions entre le « Service Environnemental » et « Dispositif d'alerte » et « Contrôleur Environnemental » sont potentiellement autorisées dans l'architecture de référence raffinée, puisque les deux derniers services héritent de « Actionneur ».
- En adoptant de la règle de l'ajout d'un point de variation, le « pvp2 » selon la logique « alternatif » est également ajouté au sein des deux connexions potentielles ci-dessous.

6. Conclusion

Ce chapitre est dédié à la représentation du langage permettant de définir les architectures de référence, lors de la phase de l'ingénierie du domaine, avec les spécifications et implantations de service. Les trois concepts principaux sont utilisés pour décrire un domaine métier. Ils sont définis par trois méta-modèles distincts mais reliés.

Les spécifications et implantations de service se caractérisent par les interfaces et les propriétés de service. Elles sont respectivement définies avec le méta-modèle de la spécification et de l'implantation de service. Elles sont réutilisables lors de la construction des architectures de référence ou des architectures d'application individuelle du domaine métier.

La séparation des principaux concepts dans différents méta-modèles a pour but de réduire l'impact de leur évolution propre sur les autres. De fait, la mise à jour d'une version de l'implantation de service n'impose pas l'évolution de la description d'implantation de service correspondante. L'ajout d'une nouvelle description d'implantation de service s'adaptant une technologie à service spécifique n'impose pas non plus l'évolution de sa spécification de service référencée.

Le méta-modèle d'architecture utilise les deux concepts précédents afin de définir les architectures de référence dans un domaine. Ces architectures peuvent être communes et partagées par plusieurs applications ou par toutes les applications dans le domaine métier. Pour exprimer la diversité entre les différentes applications individuelles, nous avons intégré un mécanisme, concernant la gestion de variabilité des produits, aux seins des architectures de référence dans le méta-modèle d'architecture.

Etant donné les limitations liées à une architecture de référence unique, dirigeant et encadrant toutes les applications d'un domaine, une hiérarchie des architectures du domaine peut être définie par le méta-modèle d'architecture. Nous avons également défini un ensemble des règles permettant de mettre en place le raffinement d'une architecture de référence.

Chapitre 6 Le prototype-*ChiSpace*

Les chapitres précédents ont présenté notre approche, fondée sur un rapprochement entre les principes des lignes de produits et des approches à services, pour faciliter le développement d'applications pervasives. Nous avons également vu que nous utilisons l'ingénierie dirigée par les modèles pour définir la notion de domaine autour des concepts de spécifications de service, d'implantations de service et d'architectures de référence.

Notre approche repose donc sur la notion de domaine et de services, et s'articule au travers de trois phases de développement, à savoir :

- *la phase d'ingénierie du domaine* durant laquelle l'architecte définit les architectures de référence du domaine constituées de spécifications de service, de liaisons entre spécifications de service et de points de variation au niveau de ces liaisons, et de contraintes diverses. L'architecture de référence est partagée par toutes les applications ou un sous ensemble des applications du domaine ;
- *la phase d'ingénierie applicative* durant laquelle l'ingénieur du domaine choisit et spécialise une architecture de référence afin de créer une architecture applicative constituée, également, de spécifications de service, d'implantations de service, de points de variation et de contraintes. Cette architecture applicative sert à dériver, à l'exécution, une application à services capable de s'adapter dynamiquement au contexte ;
- *la phase d'exécution* durant laquelle une machine d'exécution compose et exécute une application à services à partir de l'architecture applicative précédente et des services disponibles sur la machine et sur le réseau. La machine d'exécution se charge également d'adapter l'application aux changements du contexte en fonction des variations possibles définies au sein de l'architecture applicative.

Cette approche a donné lieu à un outil, actuellement à l'état de prototype, qui soutient les trois phases mentionnées ci-dessus. Cet outil, qui cherche à fournir un environnement de développement adapté aux domaines abordés, s'appelle *ChiSpace*. Ce nom s'inspire du *ChiGong*[46], qui est une pratique des mouvements se reliant aux énergies célestes et telluriques pour harmoniser la respiration à celle de l'univers. Ce prototype de l'environnement de développement est composé des trois outils correspondant respectivement aux trois phases précédentes : l'outil d'ingénierie du domaine, l'outil d'ingénierie des applications et la machine d'exécution.

Ce chapitre est structuré de la façon suivante. Nous commençons par une présentation des outils de base utilisés pour la mise en œuvre du prototype. Son architecture est ensuite présentée dans la section suivante ; elle s'articule autour des trois outils précédemment mentionnés. Les trois sections suivantes sont naturellement dédiées à ces trois outils. Elles illustrent également l'utilisation de ces outils sur des applications pervasive liée à la santé et au maintien à domicile. La sixième section conclut ce chapitre par une synthèse des points importants du prototype *ChiSpace*.

[46] **Chi Gong, Qi Gong**, ou **Chi Kung** « travail du souffle » http://fr.wikipedia.org/wiki/Qigong

1. Introduction

Le but de l'environnement *ChiSpace* est de soutenir **de façon intégrée** les trois phases de notre approche. Il est, lui-même, composé de trois outils destinés aux différents acteurs intervenant lors de la mise en place d'une application pervasive. Les deux premières phases de l'approche proposée, assimilables aux étapes d'ingénierie domaine et applicative des lignes de produits, sont soutenues par deux outils complémentaires. La dernière phase, celle de l'exécution, est mise en œuvre par une machine d'exécution spécifique.

Nous avons décidé d'exprimer et de manipuler la plupart des artefacts comme des modèles. C'est notamment le cas des spécifications de service, des implantations de service et des architectures, de référence ou applicatives. Nous avons choisi de mettre en œuvre une approche dirigée par les modèles de façon à formaliser certains aspects de notre démarche et, également, pour bénéficier d'outils, souvent disponibles en logiciel libre, permettant d'accélérer notablement les développements. La mise en œuvre de cette approche dirigée par les modèles nous a amené à réaliser les activités suivantes :

- **Définition de méta-modèles pour chacun des concepts manipulés.** Les méta-modèles fournissent un langage bien défini pour la création des différents modèles. Ils permettent aussi la création des outils de développement spécifiques conduisant à la définition de modèles conformes par construction.

- **Transformation de modèles en méta-modèles.** De façon à lier les deux premiers outils, nous avons mis en place une transformation particulière pour passer d'un modèle exprimant une architecture de référence vers un méta-modèle fournissant le langage de construction des architectures applicatives (équivalent à un passage instance vers type).

Concernant les outils de support, nous avons choisi d'utiliser le framework de modélisation Eclipse EMF (*Eclipse Modeling Framework*). Ce framework, qui reprend les principes de base de l'ingénierie des modèles, permet la manipulation de méta-modèles et de modèles**,** définis à l'aide du format Ecore. Il permet la définition de méta-modèles conformes au standard XMI[50] et de générer automatiquement les interfaces (en Java) dédiées à ces méta-modèles afin de pouvoir manipuler les modèles qui les instancient. De plus, un outil de génération de code produit un ensemble de classes permettant de visualiser et d'éditer les éléments du méta-modèle, ainsi qu'un ensemble d'éditeurs extensibles dédiés à la création des modèles et offrant une vue arborescente de ces modèles. Un méta-modèle peut être spécifié en utilisant des annotations Java, des documents XML, ou des outils de modélisation comme Rational Rose[51], et ensuite importé dans EMF. Ce framework de modélisation est basé sur la plate-forme ouverte, extensible et universel Eclipse**.** La plate-forme Eclipse est facile aujourd'hui largement utilisée et est extensible étendre par l'utilisation des mécanismes de *plug-ins*. Le framework permet également d'automatiser des tâches lourdes liées à la réalisation des approches dirigées par les modèles, par exemple, des transformations de modèles entre deux niveaux d'abstraction consécutifs.

Concernant la plate-forme d'exécution, nous nous sommes basés sur iPOJO. iPOJO fournit une plate-forme d'exécution permettant de simplifier le développement d'applications basées sur l'approche à service dynamique. Cette plate-forme est basée sur la spécification OSGi et comporte, par conséquent, les mêmes caractéristiques : centrée sur Java, centralisée et supportant un haut degré de dynamisme. De plus, cette plate-forme est extensible, ce qui permet aux utilisateurs de spécialiser iPOJO en fonction de leur domaine d'utilisation. Etant donné que le système d'exécution d'Eclipse repose sur Equinox (une

[50] **XMI** : XML Metadata Interchange - http://www.omg.org/technology/documents/formal/xmi.htm
[51] **Rational Rose** : http://www-01.ibm.com/software/awdtools/developer/rose/

implantation de OSGi), nous l'avons choisi comme base de la plate-forme d'exécution afin de faciliter l'utilisation des modèles d'architecture définis en **Ecore** lors de l'exécution. Notre plate-forme d'exécution correspond donc à une extension de iPOJO. Elle est aussi compatible avec d'autres implantations de OSGi, par exemple, Felix du consortium Apache.

2. Architecture globale de *ChiSpace*

L'architecture globale de *ChiSpace* est illustrée par la figure ci-après. Elle fait bien apparaître les trois outils correspondant au trois phases de l'approche, à savoir l'outil d'ingénierie domaine, l'outil d'ingénierie applicative et la machine d'exécution. Comme indiqué précédemment, ces outils sont intégrés au sein de l'environnement Eclipse et des transitions automatiques des uns vers les autres ont été réalisées.

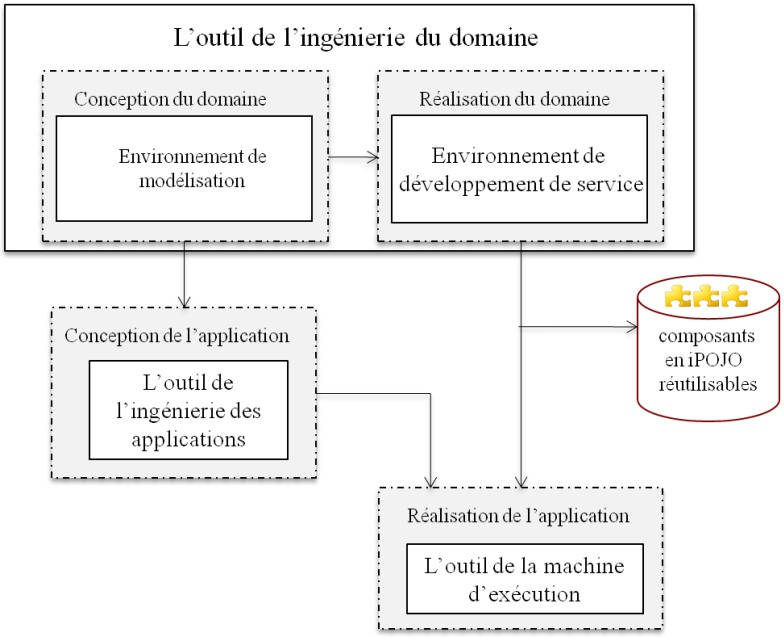


Figure 6.1. Architecture globale du prototype *ChiSpace*

L'outil d'ingénierie du domaine est composé des deux environnements de développement visant à réaliser les deux sous activités de l'ingénierie du domaine. Il s'agit, plus précisément, de la conception du domaine et de la réalisation du domaine sous forme de services (voir le chapitre 3 concernant l'état de l'art de l'ingénierie de lignes de produits logiciels) :

- **L'environnement de modélisation** du domaine est destiné aux experts et aux architectes du domaine. Il est constitué d'un ensemble d'éditeurs permettant la définition correcte des principaux concepts utilisés par notre approche. Les experts du domaine peuvent utiliser l'éditeur de services afin de décrire les exigences du domaine et représenter les connaissances du domaine. Les architectes peuvent spécifier les architectures satisfaisant à un sous-ensemble ou à toutes les applications du domaine dans l'éditeur d'architectures du domaine.

- **L'environnement de développement de services** est destiné aux développeurs et ingénieurs domaine. Cet environnement permet de développer et de décrire des implantations de services qui vont s'exécuter sur la machine d'exécution *ChiSpace*. La technologie de développement retenu est iPOJO. Une implantation de service peut correspondre au codage complet d'un service ou au codage d'un proxy qui a pour rôle d'interagir avec un service distant. La description d'une

implantation de service est plus riche que les spécifications de services puisqu'elle comprend des informations techniques additionnelles. Cette description peut être utilisée par un générateur de code, fourni dans l'environnement, pour générer automatiquement une partie du code. Les développeurs peuvent ensuite compléter le code généré dans l'éditeur Java fourni par Eclipse. Finalement, toutes les implantations de service sont enregistrées dans une base de données distante et dédiée au domaine métier étudié.

L'outil d'ingénierie applicative permet de faciliter la conception d'applications spécifiques. Cet outil spécialisé est généré automatiquement à partir des modèles définis lors de l'étape d'ingénierie du domaine. Une telle automatisation a soulevé de réels problèmes techniques mais était nécessaire. On ne pouvait demander aux experts du domaine de générer eux-mêmes les environnements applicatifs à partir de l'expression des éléments du domaine. Tout le but de l'approche est en effet de fournir des environnements masquant la complexité technique de la construction d'applications pervasives. La génération de l'outil est en fait réalisée en deux temps :

- Dans un premier temps, nous opérons une transformation des modèles issus de l'ingénierie domaine. Le but est de transformer les architectures de référence, qui sont des modèles à ce stade, en méta-modèles utilisables par les outils d'ingénierie des modèles. Cette opération est réalisée par un moteur de transformation que nous avons développé à l'aide du générateur de code *EMF* et le framework *JET*.

- Dans un second temps, l'environnement d'ingénierie applicative est généré à partir du nouveau méta-modèle. Comme pour l'environnement d'ingénierie domaine, l'outil *EMF* est utilisé pour assurer cette génération.

Au final, les ingénieurs du domaine peuvent réutiliser les artéfacts abstraits définis (*i.e.* spécifications et implantations de service) afin de créer l'architecture applicative en spécialisant l'architecture de référence. Cette opération est effectuée dans l'éditeur dédié au domaine.

La machine d'exécution, enfin, a pour but de mettre en œuvre, de façon dynamique, l'application définie lors de l'ingénierie applicative. Plus précisément, elle construit et de maintien de façon autonomique l'application à services à partir de son architecture applicative et des services disponibles sur le réseau lors de la phase d'exécution. Elle est composée d'un moteur d'interprétation et d'un modèle d'exécution:

- *Le moteur d'interprétation* se charge de l'assemblage des services, suivant une certaine stratégie, à partir de l'architecture applicative afin de produire automatiquement l'application à services. Il se charge de gérer le dynamisme et l'hétérogénéité de l'environnement et maintient un annuaire de services hétérogènes.
- *Le gestionnaire du modèle d'exécution* prend en charge la découverte des services disponibles et hétérogènes sur le réseau lors de la phase d'exécution. Il est également chargé de surveiller la disponibilité de services et de notifier les changements du contexte au moteur d'interprétation. De plus, il enregistre tous ces changements, incluant l'état de la disponibilité des services et l'historique de l'état des applications à service lors de l'exécution.

Les outils d'ingénierie domaine et applicative sont intégrés dans la plate-forme Eclipse, alors que la machine d'exécution est réalisée comme une extension de la plate-forme à services iPOJO. Etant donné que l'architecture d'Eclipse est développée autour de la notion de *plugins* en conformité avec la norme OSGi, la machine d'exécution a été naturellement intégrée dans la plate-forme Eclipse. Nous verrons, au cours de cette présentation des outils, que les outils d'ingénierie domaine et applicative sont d'un bon niveau de maturité et utilisable de façon sûre et robuste aujourd'hui. La machine d'exécution est, quant à elle, sous forme de prototype. Des travaux complémentaires seront nécessaires pour implanter de façon complète des propriétés autonomiques.

3. Outil de l'ingénierie du domaine

3.1. Vision globale

L'outil d'ingénierie du domaine de *ChiSpace* est composé de deux parties :

- **Un environnement de modélisation** matérialisé par un ensemble d'éditeurs de modèles permettant de spécifier les données, les services (abstraits et concrets), et les architectures ;
- **Un environnement de développement** de services iPOJO conformes aux spécifications.

L'**environnement de modélisation** permet aux experts du domaine de spécifier sous forme de modèles les différents éléments de solution du domaine. Par solution, nous entendons les services et les architectures permettant de répondre aux exigences du domaine (qui ne sont pas abordées dans cette suite d'outils). Le schéma ci-dessous montre le processus de construction automatisée de ces différents éditeurs de modèles.

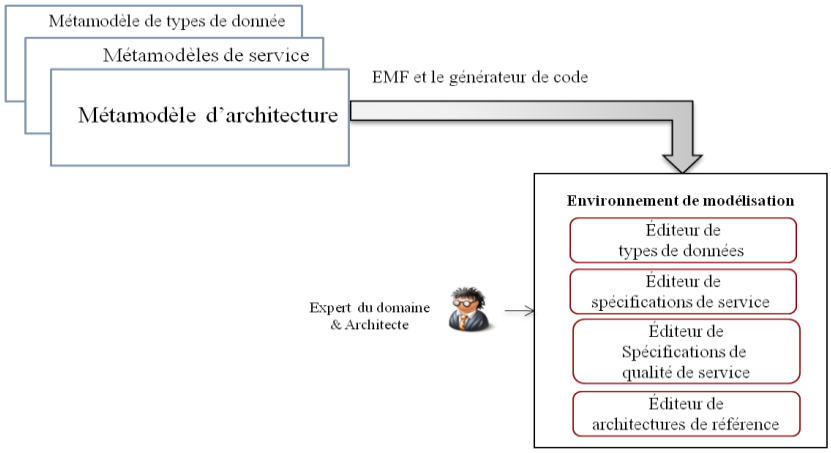


Figure 6.2. Génération d'éditeurs de modèles à partir de méta-modèles

Les éditeurs sont générés à partir des méta-modèles d'architecture, de services et de données définis lors du chapitre précédent. Les éléments définis au sein des éditeurs sont ainsi conformes par construction. De façon plus précise, nous avons utilisé l'outil de modélisation graphique *Rational Rose* pour définir les méta-modèles et le framework *EMF* pour générer les éditeurs. *EMF* prend les méta-modèles en entrée, les transforme au format *Ecore* et produit automatiquement les éditeurs. Les éditeurs comprennent un ensemble d'interfaces Java dédiés à chaque élément des méta-modèles pour accéder et manipuler leurs instances. Chaque élément d'un métamodèle peut être créé et configuré à l'aide d'une palette dédiée. Les éditeurs de modèles implantés sont les suivants :

- **l'éditeur de types de données** permet aux experts du domaine de définir les types de données spécifiques au domaine. Ceci regroupe tous les types de données nécessaires à la spécification et l'implantation des services ;
- **l'éditeur de spécifications de services** permet aux experts du domaine de définir des services métier qui se caractérisent par des interfaces fonctionnelles et des propriétés spécifiques ;
- **l'éditeur d'architecture de référence** permet aux architectes de définir les architectures de référence du domaine valables pour toutes les applications du domaine ou, dans certains cas, pour certains sous-ensembles de ces applications ;

- **l'éditeur de spécifications de qualité de** service permet de décrire les aspects non fonctionnels spécifiques au métier.

L'environnement de développement de services est, quant à lui, destiné aux développeurs et aux ingénieurs du domaine. Tous les services prennent la forme d'un composant iPOJO : certains services implantent eux-mêmes les fonctionnalités attendues, d'autres sont des « proxies » (ou des ponts) vers des services situés sur le réseau et implantés avec d'autres technologies.

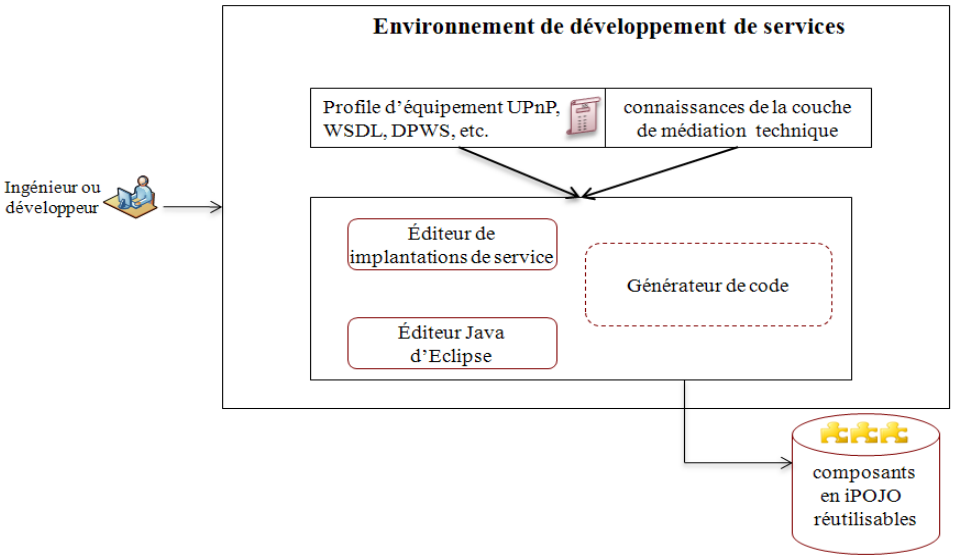


Figure 6.3. Environnement de développement de services

L'environnement de développement de services, illustré par la figure 6.3, est constitué des éléments suivants :

- **Un éditeur de description des implantations de service.** Cet éditeur permet de décrire les implantations de service en fournissant les informations concrètes attendues (de façon conforme au méta-modèle d'implantation de service présenté dans le chapitre précédent). Les descriptions d'implantations de service sont stockées au sein d'un registre. Elles y sont structurées en fonction de leur catégorie (« type » de service) ou de la technologie à services cible (UPnP par exemple).
- **Un générateur de code** qui crée les squelettes de code des composants iPOJO. En effet, à partir de la description de l'implantation de service, il est possible de générer les interfaces fonctionnelles et les interfaces de gestion du cycle de vie du composant iPOJO.
- **Un éditeur de développement Java de Eclipse**. Cet éditeur permet aux développeurs de compléter l'implantation des services iPOJO, tout en incorporant le code généré précédemment. Il peut être utile, à ce niveau, que les développeurs soient familiarisés avec la technologie OSGi.
- **Une base de données distribuée** où les différents composants iPOJO (implantations de services) sont stockées et accessibles pour être utilisés lors de la construction d'une application spécifique (lors de l'ingénierie applicative).

Une extension de cet environnement est en cours de réalisation. Il s'agit d'ajouter un générateur de code permettant de produire, de façon partielle dans la plupart des cas, des « proxies » iPOJO faisant le pont vers des services sur le réseau. Pour prendre en compte les caractéristiques d'une technologie spécifique, des informations de spécification supplémentaires sont nécessaires. Il faut non seulement disposer d'une description des services attendus suivant le format de la technologie cible mais également avoir des informations sur les relations entre les « fonctions iPOJO » et les fonctions de la technologie cible. Par exemple, comment doit être traduit un point de contrôle UPnP en interface Java. L'approche

actuelle consiste en la mise en place d'une couche de médiation technique prenant en charge l'interprétation de la spécification du service pour les transformer en interfaces Java lors de l'exécution.

3.2. Définition d'un domaine dans ChiSpace

La première fenêtre fournie par *ChiSpace* permet de définir, puis d'étendre ou de modifier, un domaine. Cette fenêtre initiale est présentée sur la figure 6.4 ci-dessous.

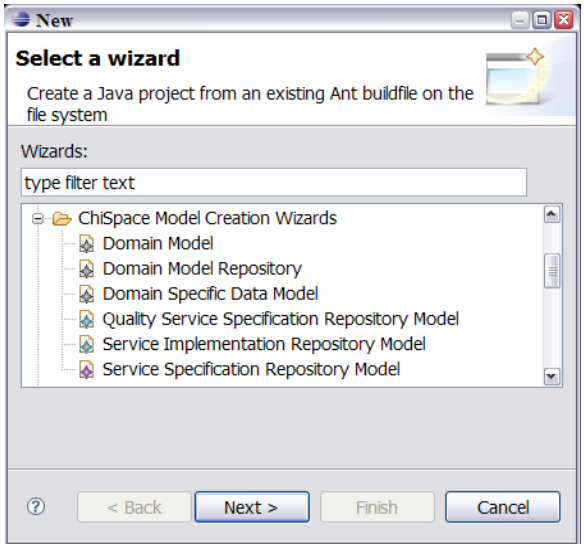


Figure 6.4. Création d'une de spécifications de service.

La création dans *ChiSpace* d'un domaine comprend ainsi la définition des éléments suivants :

- **Domain model.** Il est constitué des architectures de référence d'un domaine spécifique. Il est référencé du modèle de « Domain model repository » et réfère également aux autres modèles ci-dessous.
- **Domain Model repository.** Il organise les modèles de domaine spécifiques définis dans un registre.
- **Domain Specific Data Model.** Il est constitué de tous les types de données dédiées au domaine spécifique défini. Ces types de données sont utilisés pour définir les concepts de spécification ou d'implantation de service.
- **Quality Service Specification Repository Model.** Il est constitué de spécifications de qualité de service permettant de définir l'aspect non-fonctionnel du domaine spécifique. Ces spécifications de qualité de service sont organisées dans le registre du modèle.
- **Service Specification Repository Model.** Il est constitué de spécifications de service permettant de définir l'aspect fonctionnel du domaine spécifique. Ces spécifications de service sont organisées dans le registre du modèle.
- **Service Implementation Repository Model.** Il est constitué d'implantations de service permettant de représenter les informations techniques concernant la réalisation concrète de services. Chaque implantation de service correspondant à une spécification de service est une réalisation utilisant une technologie particulière à services. Ces implantations de service sont organisées dans le registre du modèle.

Nous présentons ci-après la définition des modèles de spécification de service, d'architecture.

3.3. Définition d'une spécification de service

Les experts du domaine commencent la description du domaine par la définition des spécifications de service. Ces définitions de services sont réalisées en utilisant l'éditeur « *servicespecificationrepository* » accessible depuis la fenêtre d'accueil proposée par ChiSpace, présentée dans la section précédente. Cette sélection est illustrée par la figure 6.5 ci-dessous.

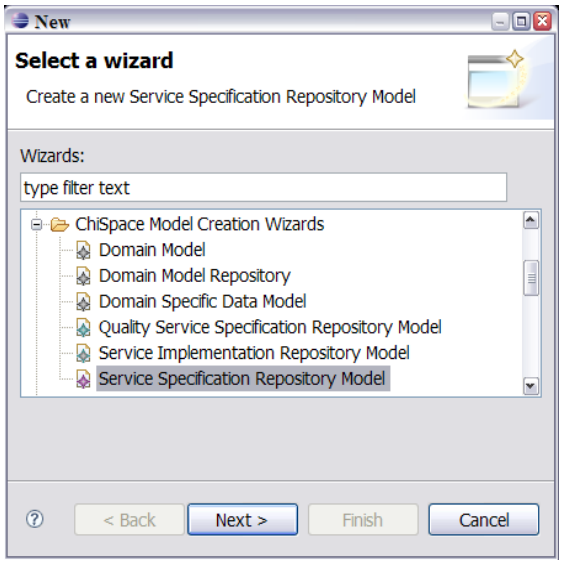


Figure 6.5. Création d'une de spécifications de service.

Les spécifications de service sont conservées au sein du registre de spécifications de service et présentées sous forme arborescente dans l'éditeur accessible par « *servicespecificationrepository* ». L'expert du domaine peut, à ce niveau, créer une spécification de service ou une spécification de qualité de service via une palette dédiée comme illustré par la figure 6.6.

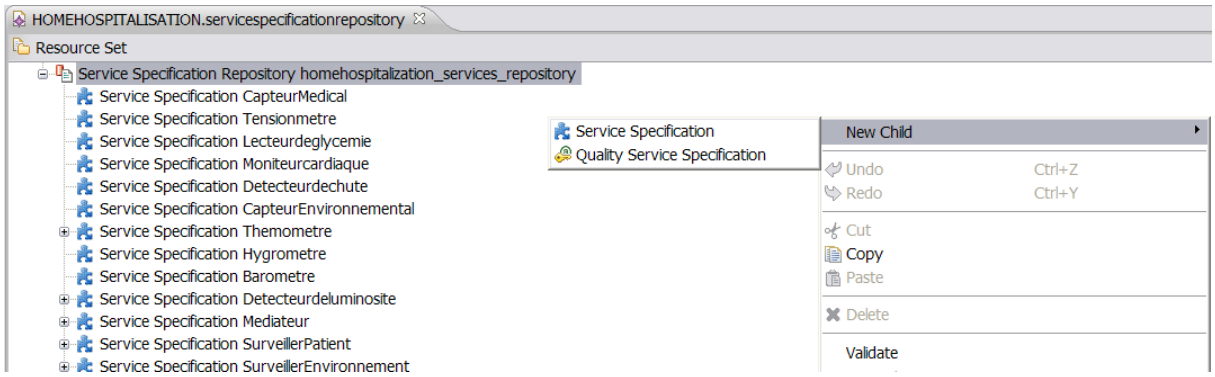


Figure 6.6. Création d'une spécification de service.

Cette même palette permet également de définir les différents éléments de spécification d'un service comme les interfaces fonctionnelles et les propriétés. Les experts du domaine définissent et configurent l'élément sélectionné dans l'éditeur à l'aide du « *Properties viewer* » dédié à cet élément du modèle. La figure 6.7, ci-après, présente la spécification d'un service « *Tensiomètre* »:

- la propriété « Name » affecte le nom de la spécification de service avec la valeur « Tensiometre »,

- la propriété « Implemented By » illustre que cette spécification de service est réalisée par « TensiometreSimulatorImpl »,
- la propriété « Extends » signifie que cette spécification de service hérite de la spécification de service « CapteurMedical ».

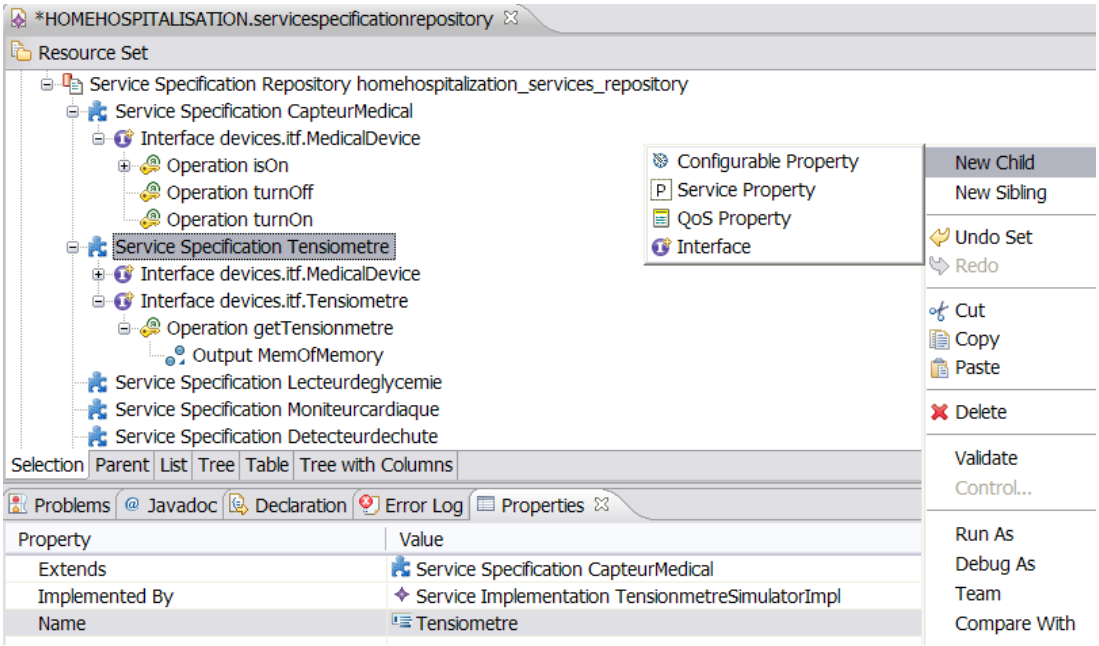


Figure 6.7. Définition d'une spécification de service.

Cette spécification de service est caractérisée par deux interfaces fonctionnelles :

- « *devices.itf.MedicalDevice* » est une interface héritée de « *CapteurMedical* »,
- « *devices.itf.Tensiometre* » est une interface spécifique contenant une opération appelée « getTensiometre », qui récupère la collection des valeurs du tensiomètre du patient. Les données récupérées correspondent au type «*MemofMemory* » spécifique au domaine de l'hospitalisation à domicile. Ces types de donnée sont définis dans le modèle de types de donnée.

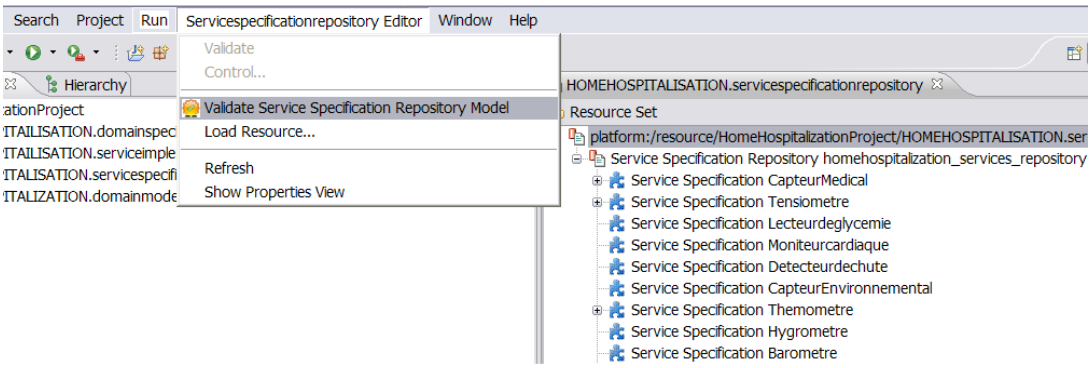


Figure 6.8. L'outil de vérification du modèle des architectures de référence

De plus, l'éditeur « *servicespecificationrepository* » fournit un outil (sous forme d'un menu spécifique illustré par la figure 6.8) permettant d'assister l'expert du domaine pour vérifier la correction du registre de spécifications de service. Plus précisément, il se charge, d'abord, de vérifier s'il ne manque pas une propriété obligatoire dans la spécification d'un service. C'est par exemple le cas du nom de la spécification de service, du nom d'une interface d'une spécification de service, ou du nom d'une

opération. Cet outil vérifie également certaines incohérences au sein des spécifications. Il vérifie, par exemple, que deux spécifications de service ne portent pas le même nom ou que deux spécifications de service ne sont pas réalisées par la même implantation de service.

3.4. Définition d'architectures de référence

Comme pour tous les autres éléments du domaine, la définition d'une architecture de référence commence par le choix de l'outil approprié dans la liste fournie par la fenêtre initiale de *ChiSpace*. Dans ce cas, ainsi qu'illustrée par la figure 6.9, il s'agit de sélectionner « *Domain Model* ».

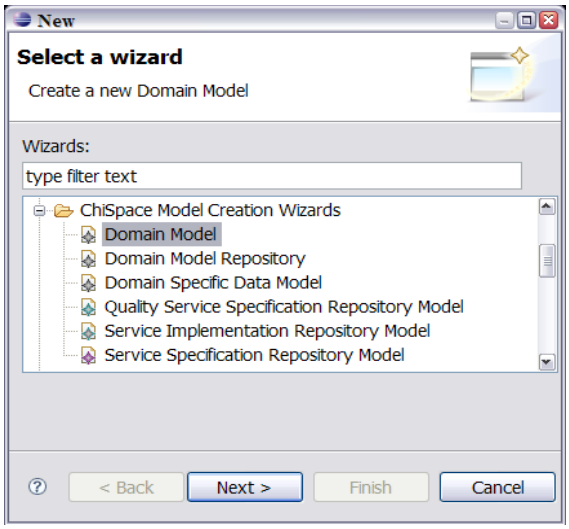


Figure 6.9. Création d'un modèle du domaine.

La notion de « *Domain Model* » est plus large que celle d'architecture de référence. Un modèle du domaine peur en effet comprendre une hiérarchie d'architectures de référence, comme introduit dans le chapitre précédent. Au sommet de la hiérarchie se trouve l'architecture la plus générique, c'est-à-dire avec le plus haut degré de variabilité. Ensuite, au sein de la hiérarchie se trouvent des architectures raffinées. Dans l'exemple ci-dessous (figure 6.10), nous avons défini une architecture de référence nommée « *HomeHopspitalizationAbstractRA* » et une spécialisation nommée « *HomeHospitalizationRefinedRA* ».

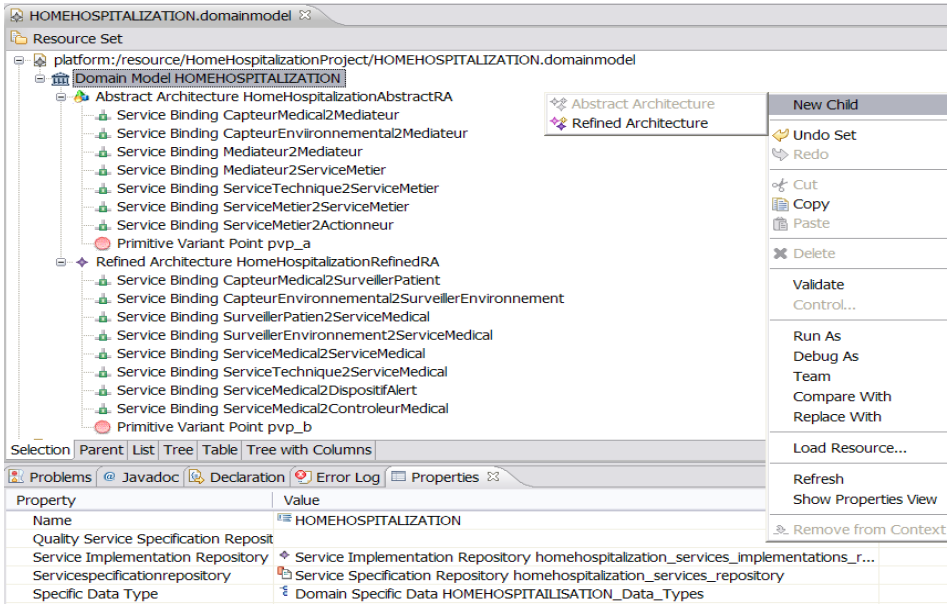


Figure 6.10. Création d'architectures de référence.

La relation de raffinement entre les deux architectures de référence est exprimée par l'utilisation de la propriété « *Refines* » lors de la définition de l'architecture « HomeHospitalizationRefinedRA ». Ceci est illustré par la figure 6.11 ci-dessous.

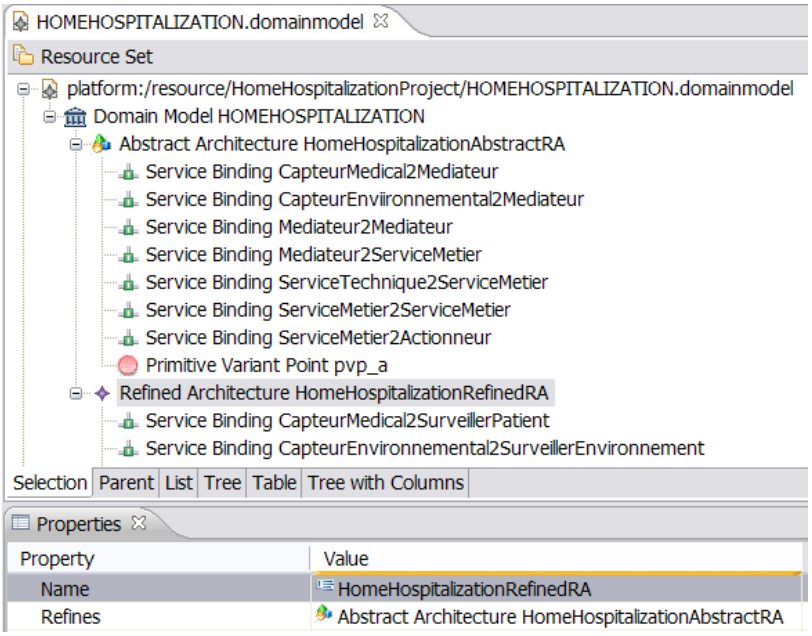


Figure 6.11. Création d'une architecture de référence.

Une architecture de référence est définie par des spécifications de service, des connecteurs entre ces spécifications, des points de variation (primitifs ou avancés) et des contraintes. Ces différents éléments peuvent être créés par l'utilisation de la palette dédiée à l'architecture de référence (illustrée par la figure6.12).

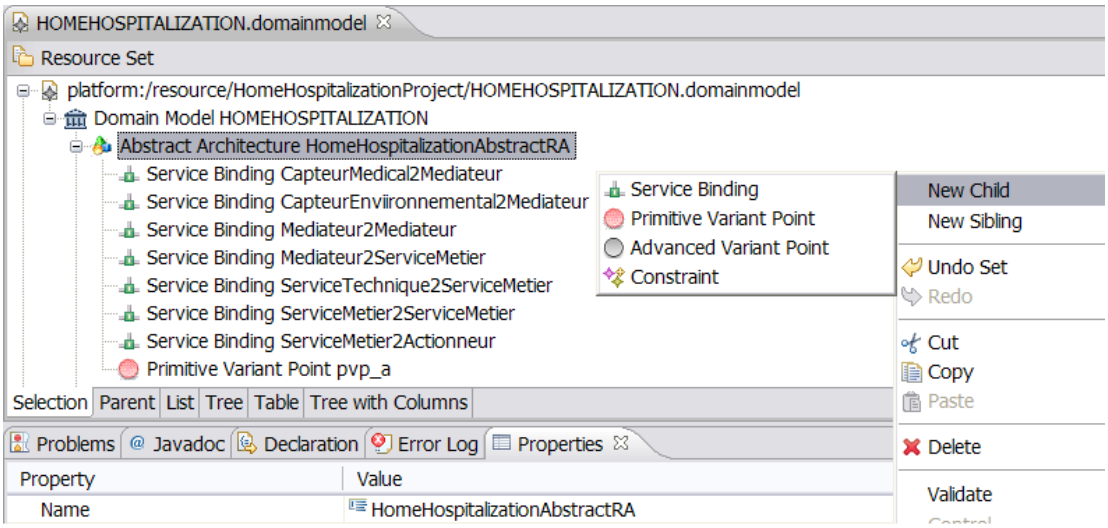


Figure 6.12. Définition d'une architecture de référence.

Dans un premier temps, l'architecte doit choisir les spécifications de services formant l'architecture de référence. Pour ce faire, il choisit des spécifications mises à disposition dans le modèle du registre de spécifications de service. Ce modèle est chargé dans le modèle du domaine à l'aide de la propriété « *servicespecificationregistry* » à partir du concept « *Domain Model* ».

Dans un second temps, l'architecte doit créer les connecteurs entre spécifications de services. Dans l'outil, les connecteurs sont des « *Service Binding* ». Ils se caractérisent par les propriétés listées dans le « *Properties viewer* » de la figure 6.13 ci-dessous.

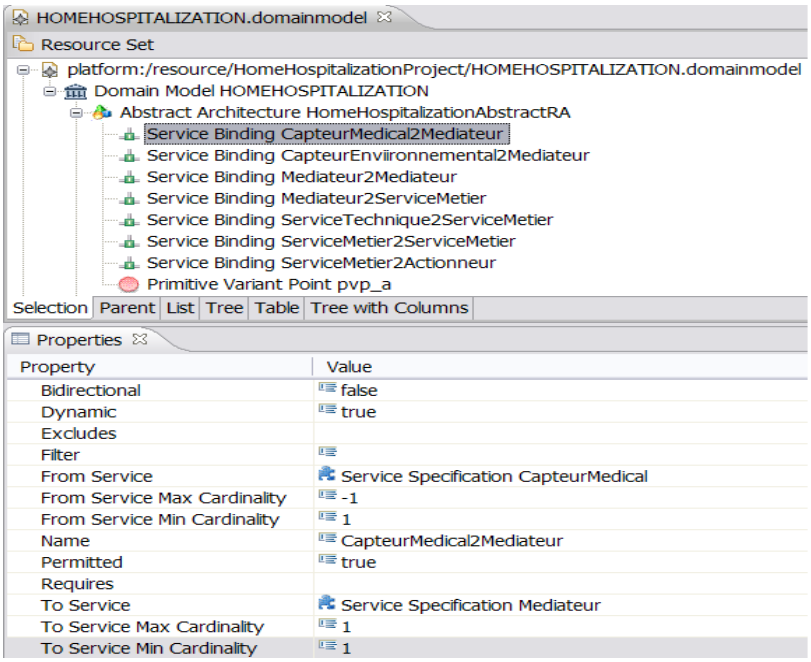


Figure 6.13. Définition d'un connecteur de spécifications de service

Dans cet exemple, le connecteur « *CapteurMedical2Mediateur* » définit la liaison entre « *CapteurMedical* » et « *Médiateur* ». La propriété « Bidirectional » est assignée à la valeur « *false* », ce qui signifie que le sens de l'invocation entre les deux services est unidirectionnel de « *CapteurMedical* » vers « *Médiateur* ». Les cardinalités exprimées indiquent qu'une ou plusieurs instances de

« *CapteurMedical* » peuvent invoquer une instance de « *Médiateur* ». Cette liaison des spécifications de service est autorisée dans l'architecture de référence « *HomeHospitalizationAbstractRA* » par l'assignation de la valeur « *true* » à la propriété « *Permitted* ».

La figure 6.14 illustre la définition d'un point de variation primitif au niveau d'une connexion. On voit apparaître, dans la partie basse, les différentes propriétés devant être renseignées.

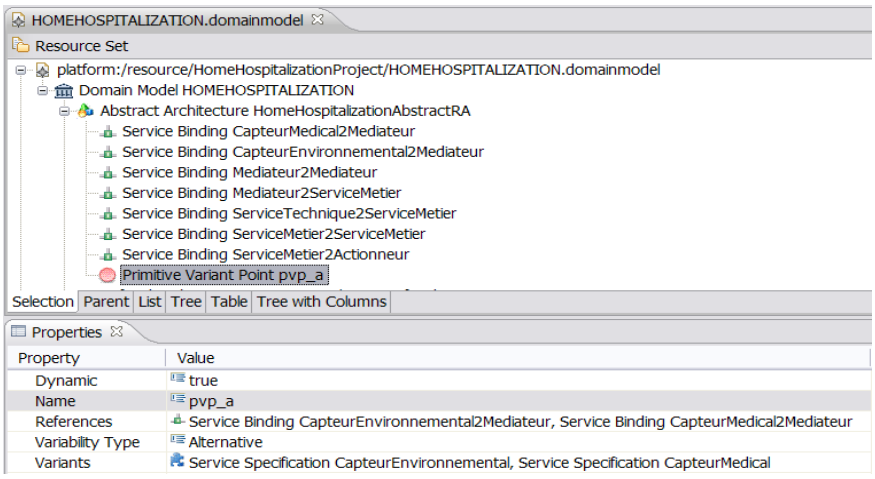


Figure 6.14. La définition du point de variation

Plus concrètement, dans notre exemple, le point de variation primitif « *pvp_a* » définit les variantes au niveau de la connexion de « *CapteurEnvironnemental2Mediateur* » et « *CapteurMedical2Mediateur* » grâce à la propriété « *References* ». Le type de variabilité de ce point de variation est « *Alternative* » suivant le logique « *xor* » (soit l'un, soit l'autre) présenté dans le chapitre précédent. De plus, la propriété « *dynamic* » avec la valeur « *true* » signifie que les variantes associées à ce point, à savoir « *CapteurEnvironnemental* » et « *CapteurEnvironnemental* » peuvent être dynamiquement sélectionnées lors de l'exécution afin de s'adapter aux changements du contexte.

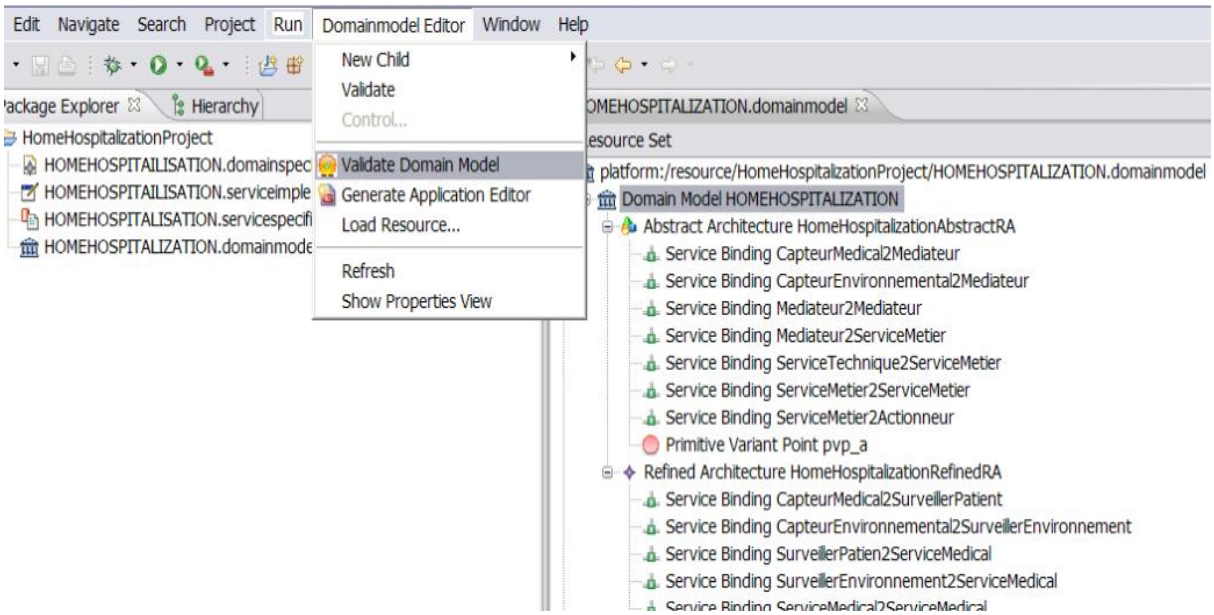


Figure 6.15. L'outil de vérification du modèle des architectures de référence

Enfin, tout comme l'éditeur supportant la définition de spécification de services, l'éditeur des architectures de référence du domaine fournit des outils permettant d'assister l'architecte lors de la vérification de la correction du modèle des architectures de référence. Ces outils sont accessibles depuis des sous-menus, comme illustré par la figure 6.15.

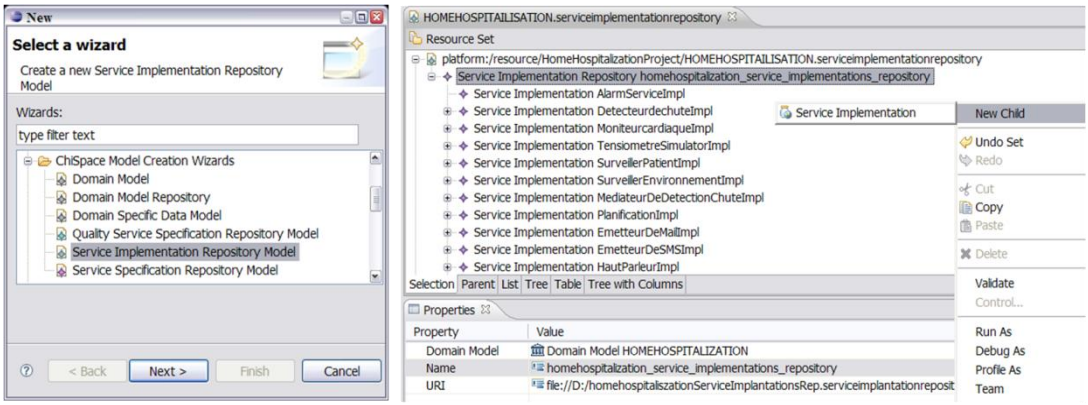
Plus précisément, l'outil se charge, d'abord, de vérifier si toutes les propriétés nécessaires sont bien définies pour chaque élément du modèle. Il est, par exemple, nécessaire de bien spécifier le nom identifiant les différentes architectures de référence ou les différents points de variation. Pareillement, au niveau des *ServiceBinding*, il est nécessaire de spécifier les propriétés *fromService* ou *toService*.

L'outil se charge également de vérifier la cohérence d'ensemble des architectures de référence. Par exemple, deux architectures ne peuvent avoir le même nom. Pareillement, cet outil de vérification se charge de vérifier la définition contradictoire concernant les identificateurs des ***ServiceBindings*** et des points de variation. De plus, il permet d'analyser les dépendances contradictoires entre les ***ServiceBindings*** ou les points de variation de l'architecture de référence à l'aide des propriétés *« Requires » et « Exludes »* du ***ServiceBindings.***

Plus important, l'outil assure que des architectures raffinées restent bien en conformité avec les architectures qu'elles affinent (suivant la définition donnée dans le chapitre précédent).

3.5. Définition d'implantations de service

L'expert du domaine doit également spécifier les implantations de service et faire le lien avec les spécifications de service (une spécification pouvant donner lieu à plusieurs implantations). Les implantations de service sont sauvegardées au sein du registre d'implantations de service et présentées dans l'éditeur approprié sous une forme arborescente. Pour spécifier une implantation de service, l'expert du domaine doit choisir l'éditeur lié au registre d'implantations de service, accessible depuis la fenêtre d'accueil proposée par *ChiSpace*. Après la création du modèle du registre d'implantations de service, l'expert du domaine peut utiliser la palette dédiée au registre pour créer l'implantation de service.



a) Création du modèle du registre d'implantations de service

b) Création d'implantation de service

Figure 6.16. Spécification d'une implantation de service

L'exemple « *AlarmServiceImpl* » (présenté dans le chapitre précédent) est défini en utilisant l'éditeur d'implantations de service comme illustré sur la figure 6.17 ci-dessous. Cette implantation de service est identifiée par le nom « *AlarmServiceImpl* » et la version « *0.1* ». Elle implante la spécification de service « *Alarme* ». Elle comprend un fichier décrivant les interfaces fonctionnelles réalisées par cette implantation et un fichier des propriétés, incluant des propriétés de configuration des instances de service. Enfin, le mode de communication réalisé par cette implantation de service est « *Event* ». Tous les fichiers sont générés automatiquement à partir de la spécification de service « *Alarme* » implantée.

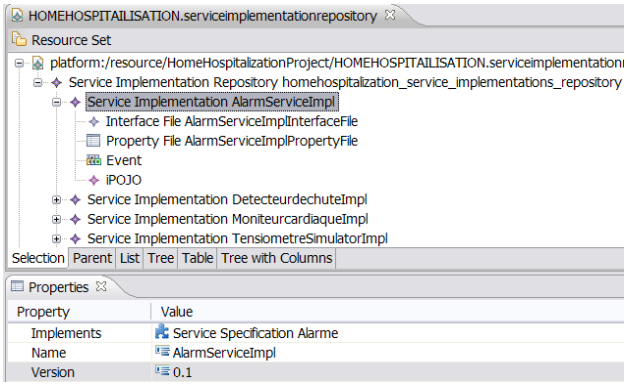


Figure 6.17. Définition d'une implantation de service.

Cet éditeur fournit aussi un outil (sous forme de sous-menu) pour vérifier la correction du modèle du registre d'implantations de service. Le fonctionnement de cet outil de vérification est assimilable à celui de l'outil de vérification de l'éditeur « *servicespecificationrepository* » introduit précédemment.

4. Outil d'ingénierie applicative

4.1. Génération de l'outil

La séparation entre les processus d'ingénierie domaine et applicative est à la base des lignes de produits et de notre approche pour la conception d'applications pervasives. Ce processus à deux phases apporte de nombreux avantages mais induit également de réelles difficultés au niveau de la synchronisation des processus et de la gestion de la cohérence entre les modèles du domaine et les modèles des applications.

Pour remédier à ces problèmes fondamentaux, nous avons choisi de générer les outils d'ingénierie applicative à partir des modèles issus de l'ingénierie domaine. Cette approche, qui repose sur des techniques de transformation de modèles et sur l'utilisation du framework *EMF*, apporte les avantages suivants :

- Elle automatise le passage de l'ingénierie domaine à l'ingénierie applicative ;
- Elle garantit la conformité, par construction, des modèles au niveau de l'ingénierie applicative ;
- Elle réduit fortement les coûts de mise en place d'environnements de travail liés à l'ingénierie applicative.

Pour construire automatiquement un éditeur de développement d'architecture applicative avec *EMF*, il convient, dans une approche dirigée par les modèles, de disposer d'un méta-modèle des architectures applicatives. Il s'agit donc, pour nous, de générer un tel méta-modèle à partir d'une architecture de référence (qui est un modèle conforme par instanciation au méta-modèle des architectures de référence).

Notre approche pour cela est de mettre en place une transformation de modèle. Une transformation de modèles est effectuée grâce à un moteur de transformation prenant un modèle original en entrée et produisant un modèle cible en appliquant certaines règles de transformation. Dans notre cas, le but de la transformation est de passer d'un modèle à un méta-modèle : ceci est bien une transformation de modèle puisqu'un méta-modèle est lui-même un modèle. Ceci est illustré, de façon globale, par la figure 6.18.

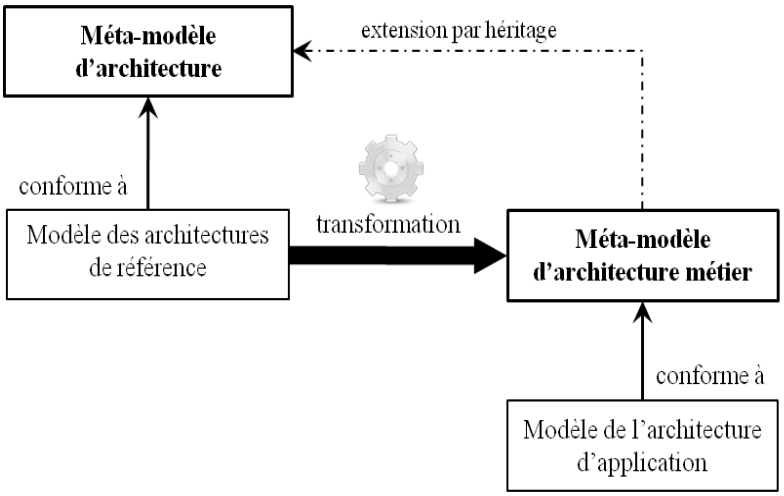


Figure 6.18. Schéma de la transformation de modèles

Nous avons choisi de construire le méta-modèle des architectures métier comme une extension du méta-modèle des architectures de référence. Des concepts généraux, comme le concept de services par exemple, sont étendus par héritage avec des concepts métier, comme *Capteur* ou *Température*. De cette façon, les principes architecturaux restent les mêmes pour toutes les architectures manipulées dans notre approche, qu'elles soient de référence ou applicatives.

Pour résumer :

- **Le modèle original** est une architecture de référence. C'est une instance du méta-modèle d'architecture. Ce dernier est constitué de spécifications de services, de connexions entre ces spécifications et de concepts permettant d'exprimer la notion de variabilité, tels que *PrimitiveVariationPoint* et *AdvancedVariationPoint*.

- **le modèle cible** est un méta-modèle. Il s'agit du méta-modèle des architectures métier fournissant le langage permettant de concevoir et développer les architectures applicatives. C'est une extension par héritage du méta-modèle d'architecture avec des spécifications de service métier.

Un effet important de notre démarche est que la plupart des concepts techniques disparaissent au niveau de l'ingénierie applicative. Lors de construction d'une architecture applicative, on ne parle plus de spécifications de service mais de spécifications métier : on associe une *Alarme* et un *ServiceMétier* par exemple. Cela permet aux développeurs de se concentrer sur l'aspect métier et de créer des architectures applicatives à services en ne manipulant que des « concepts » du domaine métier. La communication entre les différents intervenants du développement est ainsi rendue plus facile et plus naturelle.

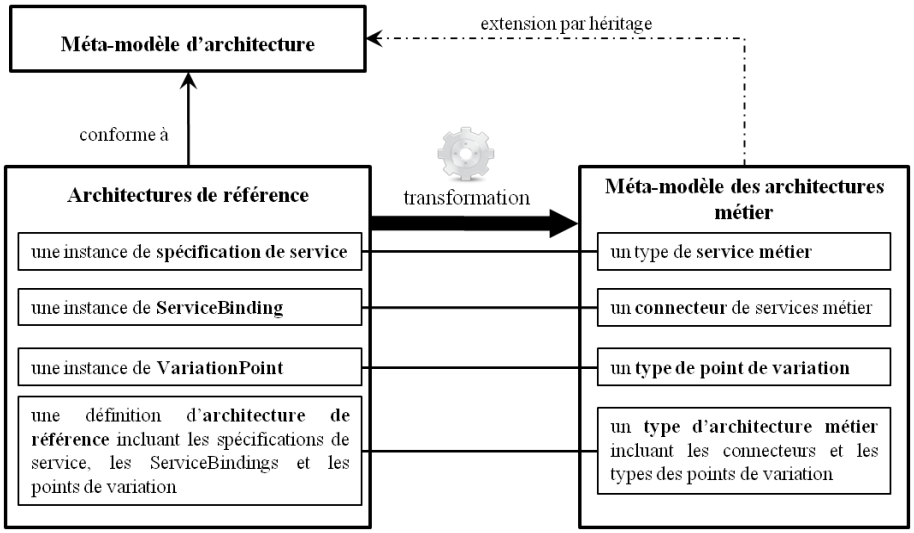


Figure 6.19. Correspondance des concepts lors de la transformation de modèle en méta-modèle.

La figure 6.19 détaille les règles de correspondance entre concepts du modèle original et concepts du modèle cible. Les relations majeures sont les suivantes :

- chaque instance de spécification de service définie au sein d'une architecture de référence est transformée en une spécification de service métier dans le méta-modèle d'architectures métier. Cette spécification de service métier hérite du concept de spécification de service emprunté au méta-modèle d'architecture de référence. Les propriétés de la spécification de service spécifique sont des copies des propriétés définies avec leur(s) valeur(s) donnée(s) dans la définition de l'instance de la spécification de service correspondant ;

- chaque lien entre des instances de spécification de service défini au sein d'une architecture de référence est transformé en un connecteur spécifique dans le méta-modèle d'architectures métier. Ce connecteur spécifique hérite du concept ServiceBinding pour définir les liaisons entre spécifications de service spécifiques. Pour chaque connecteur, ses propriétés sont la copie des propriétés avec leur valeur donnée de ServiceBinding correspondant. Ces valeurs ne peuvent plus être modifiées, à l'exception des nombres de services liés qui peuvent être configurés avec une valeur de leurs intervalles ;

- chaque point de variation d'une architecture de référence est transformé en un point de variation dans le méta-modèle des architectures métier. Il décrit les variations au niveau d'une connexion entre plusieurs spécifications de service métier. Il hérite des concepts « PrimitiveVariationPoint » ou « AdvancedVariationPoint » définis dans le méta-modèle original d'architecture. Il faut noter que la valeur du type de variabilité du point de variation défini dans le modèle des architectures de référence ne peut pas être modifiée par le point de variation héritant dans le méta-modèle d'architectures métier ;

Pour réaliser notre transformation de modèles, nous avons utilisé l'outil *JET*[52] intégré sous la plate-forme *Eclipse*. *JET* fournit un framework de génération de code et les outils qui sont utilisées par le framework *EMF*.

Nous avons défini en ensemble de *Templates JET* pour transformer les architectures de référence en méta-modèles d'architecture métier, pour ensuite créer les fichiers .Ecore correspondants. Ensuite, nous réutilisons le générateur de code d'*EMF* de façon programmatique afin de générer automatiquement un éditeur spécifique de base ainsi que ses outils pour le développement d'architectures d'applications du domaine.

```
DomainModelEcore.jet
<%@jet imports="domainmodel.*"%>
<%@taglib prefix="ws" id="org.eclipse.jet.workspaceTags" %>
<?xml version="1.0" encoding="utf-8"?>
<ecore:EPackage xmi:version="2.0"xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
    nsURI="http:///<c:get select = "$nameDM"/>.ecore" nsPrefix="<c:get select = "$nameDM"/>">
<eClassifiers xsi:type="ecore:EClass" name="Application" eSuperTypes="platform:/plugin/SpecificDomain/model/specificdom
<c:iterate select = "$domainModel/abstractArchitecture" var="aArchitecture">
<eSubpackages name="<c:get select="$aArchitecture/@name"/>" nsURI="http:///<c:get select = "$nameDM"/>/<c:get select="$
      nsPrefix="<c:get select = "$nameDM"/>.<c:get select="$aArchitecture/@name"/>">
  <eClassifiers xsi:type="ecore:EClass" name="<c:get select="$aArchitecture/@name"/>" eSuperTypes="platform:/plugin/Spe
    <c:iterate select = "$aArchitecture/serviceBindings" var="serviceBinding">
 <eClassifiers xsi:type="ecore:EClass" name="<c:get select="$serviceBinding/@name"/>" eSuperTypes="platform:/plugin/Spe
    <%
    ServiceBinding sb = (ServiceBinding)context.getVariable("serviceBinding");
    String nameFromService = sb.getFromService().getName();
    String nameToService = sb.getToService().getName();
    %>
    <eStructuralFeatures xsi:type="ecore:EReference" name="from" lowerBound="1"
        eType="ecore:EClass <c:get select = "$nameDM"/>AbstractRepository.ecore#//<%=nameFromService%>"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="to" lowerBound="1"
        eType="ecore:EClass <c:get select = "$nameDM"/>AbstractRepository.ecore#//<%=nameToService%>"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="bidirectional" lowerBound="1"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBooleanObject"
          changeable="false" defaultValueLiteral="<%=sb.getBidirectional()%>"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dynamic" lowerBound="1"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBooleanObject"
          changeable="true" defaultValueLiteral="<%=sb.getDynamic()%>"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="permitted" lowerBound="1"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBooleanObject"
          changeable="false" defaultValueLiteral="<%=sb.getPermitted()%>"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="filter" lowerBound="1"
          eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"
          changeable="false" defaultValueLiteral="<%=sb.getFilter()%>"/>
</eClassifiers>
    </c:iterate>
    <c:iterate select="$aArchitecture/variantPoints" var="vp">
        <c:choose>
<%-- ------------------------------------------------------------------------------------------------------------
            <c:when test="$vp/self::PrimitiveVariantPoint">
    <%
    PrimitiveVariantPoint vpp = (PrimitiveVariantPoint)context.getVariable("vp");
    String nameVariabilityType = vpp.getVariabilityType().getName();
```

Figure 6.20. Un morceau de *Template* JET transformant une architecture de référence en méta-modèles d'architecture métier

[52] **JET** : Java Emmitter Templates - http://www.eclipse.org/modeling/m2t/?project=jet#jet

Nous avons également personnalisé les outils qui permettent de visualiser et d'éditer les éléments du méta-modèle d'architectures métier. Cela simplifie et facilite la définition d'architectures applicatives grâce à la fourniture d'un plus grand nombre d'informations ou de contraintes issues des architectures de référence. Enfin, tous les outils générés sont intégrés de façon automatique au-dessus du framework *EMF* et *JET* afin de constituer l'outil d'ingénierie des applications.

4.2. Définition d'architecture applicative

La création de l'architecture d'une application est réalisée par le technicien du domaine ou le développeur des applications. Il utilise l'éditeur de l'architecture applicative dédié au domaine afin de réaliser la conception pour une application donnée à partir de l'architecture métier sélectionnée.

Pour réaliser cette création, le technicien du domaine choisit d'abord l'outil dédié au domaine spécifique qui est généré par la transformation de modèles. Dans notre exemple, l'outil dédié au domaine des applications d'hospitalisation à domicile est l'éditeur « *homehospitalization* » permettant de définir les modèles des architectures applicatives - « HOMEHOSPITALIZATION Model » - illustré dans la figure 6.21, ci-après :

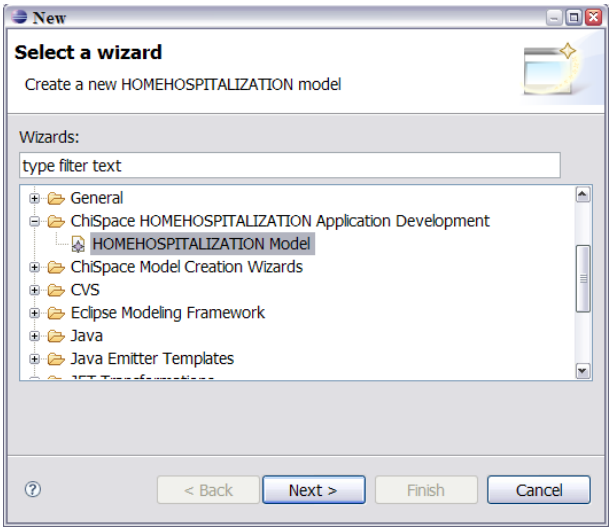


Figure 6.21. La création du modèle de l'architecture d'une application dédiée au domaine de l'hospitalisation à domicile

Dans notre exemple, nous allons définir l'architecture de l'application « Rappel des rendez-vous avec les spécialistes ou les médecins ». Nous précisons le fonctionnement de cette application dans le chapitre suivant. Cette section présente uniquement comment créer l'architecture de cette application avec l'outil de l'ingénierie des applications. Cette application est identifiée par le nom « RappelRDVAvecMedecins ». Le concept « Application » est considéré comme l'élément radical du modèle de l'architecture applicative. A partir de ce concept, le technicien du domaine doit choisir une architecture métier parmi les architectures métier, qui sont transformées à partir des architectures de référence définies lors de la phase précédente. La figure 6.22 illustre l'éditeur du modèle de l'architecture de cette application et la palette dédiée à la sélection d'une architecture métier.

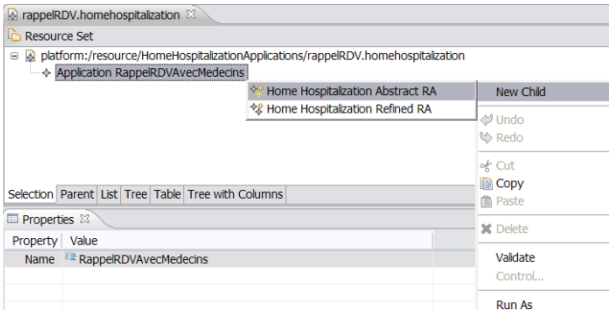


Figure 6.22. La sélection d'une architecture métier

Puis, le technicien spécialise l'architecture métier sélectionnée afin de définir l'architecture applicative avec ses types de composant abstrait, ses connecteurs et ses points de variation. Dès que l'architecture métier est sélectionnée, une fenêtre associée à cette architecture métier est automatiquement lancée pour assister la définition de l'architecture de l'application « RappelRDVAvecMedecins ». Cette fenêtre illustre uniquement les types de composant abstrait utilisés par l'architecture métier « Home Hospitalization Abstract RA » et les sous-types de ces types de composant abstrait sous une forme arborescente. La figure 6.23 montre une capture d'écran de cette fenêtre:

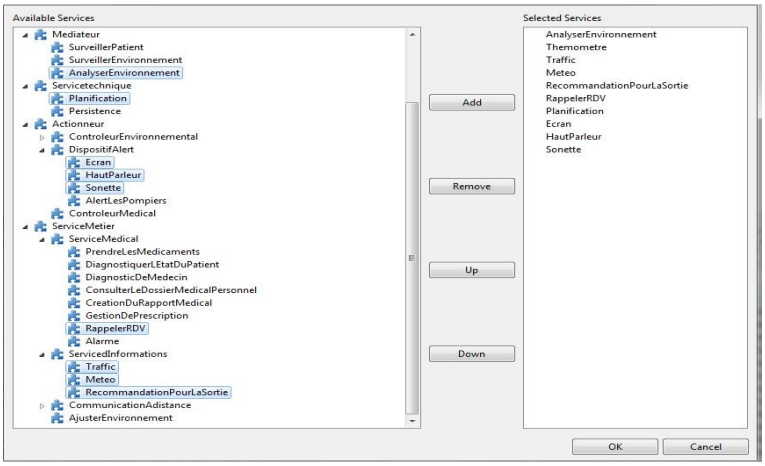


Figure 6.23. La fenêtre de sélection des types de composant abstrait constituant l'architecture applicative

Le technicien peut ensuite sélectionner les services métier pour définir l'architecture applicative. Après la validation des choix des services métier, les connecteurs décrivant les liaisons entre les services métier sélectionnés peuvent être automatiquement créés. De plus, certains points de variation peuvent être créés automatiquement en fonction de la définition de l'architecture métier « Home Hospitalization Abstract RA ». Le technicien peut évidemment continuer la spécialisation de cette architecture métier jusqu'à son objectif spécifique : l'application « RappelRDVAvecMedecins ». Nous illustrons le modèle de l'architecture applicative correspondant à cette application dans l'éditeur « *homehospitalization* » ci-après :

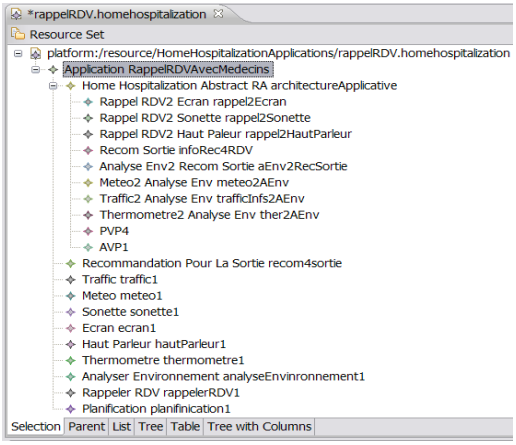


Figure 6.24. L'architecture de l'application « *RappelRDVAvecMedecins* » définie par l'utilisation l'éditeur « *homehospitalization* »

Finalement, l'architecture applicative construite lors de cette phase est considérée comme le cadre des contraintes structurales permettant de dériver l'application à services exécutable ainsi que les contraintes sur son évolution lors de l'exécution.

5. L'outil de la phase d'exécution - *Machine d'Exécution*

5.1. Vision globale

L'objectif de la machine d'exécution est de créer et de maintenir de façon autonomique une application faite de services hétérogènes à partir d'une architecture applicative contenant encore des points de variation. De façon plus précise, une telle application est constituée d'un ensemble d'instances de composants iPOJO ; ces instances peuvent dans certains cas correspondre à des « proxies » vers des services distribués. Cette architecture concrète, que nous appellerons également l'application en cours d'exécution, est formée en tenant compte des contraintes structurelles apportées par l'architecture applicative et du contexte d'exécution, c'est-à-dire des services réellement disponibles lors de l'exécution.

La machine d'exécution est constituée de deux composants logiciels essentiels : une plate-forme d'intégration de service et d'exécution, et un moteur d'interprétation. L'architecture de la machine d'exécution est schématisée sur la figure 6.25.

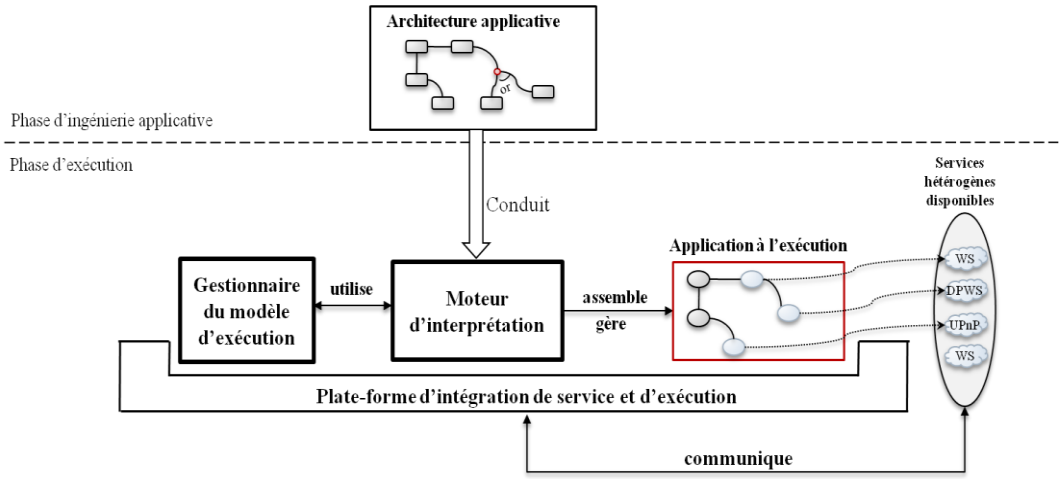


Figure 6.25. L'architecture globale de la machine d'exécution

Le rôle de la plate-forme d'intégration de service et d'exécution est d'exécuter les applications à base de composants iPOJO et de gérer la découverte et l'interaction des services se situant sur le réseau. Cette plate-forme doit également tenir à jour une vue synthétique des services disponibles dans l'environnement. Son rôle est donc de surveiller leur disponibilité et de renseigner, de façon dynamique, un ensemble de propriétés les concernant. Elle est également chargée d'inspecter continuellement l'état de la plate-forme d'exécution. Toutes ces informations sont reportées au sein d'un modèle d'exécution. Le modèle d'exécution peut, en fait, être vu comme un *model@runtime* tel que récemment défini par la communauté de l'ingénierie des modèles [109].

Le moteur d'interprétation s'occupe de la création puis de la gestion des applications. Pour cela, le moteur d'interprétation reçoit l'architecture applicative développée lors de l'ingénierie applicative et utilise le modèle de l'environnement. Ce moteur d'interprétation gère de façon autonomique les applications dont il a la charge en se basant sur la politique suivante : maximiser la disponibilité globale de l'application et économiser les ressources de la plate-forme d'exécution. Il sélectionne, crée et assemble ainsi les instances de services correspondants à l'architecture applicative tout en se conformant à cette politique. Les services utilisés sont ceux présents au sein du modèle d'exécution (lequel est géré par la plate-forme d'intégration de service sous-jacente).

5.2. Modèle d'exécution

L'objectif du gestionnaire du modèle d'exécution est de répondre aux défis liés à l'hétérogénéité des services au sein d'un environnement pervasif et à leur dynamisme. Plus précisément, il a pour but de présenter deux informations majeures au moteur d'interprétation : l'ensemble des services disponibles avec un ensemble de qualités sous une forme homogène et compréhensible par la plate-forme d'exécution, et l'architecture concrète de l'application en cours d'exécution.

Dans notre cas, la plate-forme d'exécution cible repose sur le modèle à composant iPOJO, au dessus de l'intergiciel OSGi. Ainsi, toutes les implantations de service sont présentées sous la forme de composants iPOJO, qu'il s'agisse de services « locaux » ou distribués sur le réseau et basés sur une autre technologie. Ainsi, l'intégration de technologies hétérogènes comme UPnP, les Services Web ou DPWS est réalisé à travers des ponts permettant la communication vers ces différentes technologies.

Le gestionnaire du modèle d'exécution fournit les fonctionnalités suivantes :

- découvrir les services iPOJO disponibles sur la plate-forme d'exécution cible ;
- importer dynamiquement des services hétérogènes (réalisés dans d'autres technologies qu'iPOJO) disponibles sur le réseau local ou sur l'Internet. Il s'agit d'abord de découvrir les services et ensuite d'importer ou de générer les « proxies » spécifiques appropriés permettant aux autres services iPOJO d'invoquer ces services ;
- enregistrer toutes les implantations de service déployées sur la plate-forme. Ces implantations de service au niveau du code permettent de créer les instances de service lors de la phase d'exécution à partir de la configuration des instances dans l'architecture applicative ;
- enregistrer toutes les instances de services disponibles dans le modèle d'exécution ;
- mettre à jour le modèle d'exécution et notifier les changements des instances de services disponibles pour le moteur d'interprétation lors de l'exécution.

Il maintient également les informations suivantes :

- la liste des instances de services disponibles ;
- la liste des implantations de service déployées ;
- l'historique des états de l'architecture concrète de l'application.

La liste des instances de services disponibles a pour but de mémoriser l'ensemble des instances de services disponibles sur la plate-forme d'exécution. Ces instances de services sont prises en compte comme les éléments de base afin de construire l'application à l'exécution.

La liste des implantations de service déployées mémorise les implantations de service disponibles sur la plate-forme d'exécution. Une implantation de service peut être considérée comme une « factory », utilisable par le moteur d'interprétation afin de créer des instances du service considéré. Chaque implantation de service peut correspondre à une ou plusieurs des instances de service de l'architecture concrète de l'application.

L'historique des états a pour but de mémoriser toutes les architectures concrètes depuis la création de l'application. Chaque application possède ainsi une journalisation de ses architectures concrètes, toujours conformes à son architecture applicative. A tout moment, l'état de l'application à l'exécution peut être représenté par un « snapshot » architectural de l'application. Les différents « snapshots » sont pris lorsqu'un événement vient modifier l'architecture concrète de l'application. Lorsqu'un tel événement apparait, le nouvel état de l'application est enregistré avec l'événement ayant causé la modification et la date du changement. Cette connaissance est utilisée lors de la sélection des instances de services pour la construction et la gestion de l'application à l'exécution.

5.3. Plate-forme d'intégration de service

La plate-forme d'intégration de service repose sur l'outil open source ROSE[53], développé au sein de l'équipe Adele du LIG (www-adele.imag.fr). Comme nous l'avons introduit précédemment, le principe d'intégration repose sur l'utilisation du patron de conception « proxy » [9].

Le rôle de ROSE est de découvrir les services disponibles sur le réseau, suivant différentes technologies, de faire le lien avec des « proxies » implantés en iPOJO, et de charger ces « proxies » dans l'annuaire à service de la plate-forme d'exécution. De cette façon, les services distants sont mis à disposition des applications en cours d'exécution sur la plate-forme d'exécution.

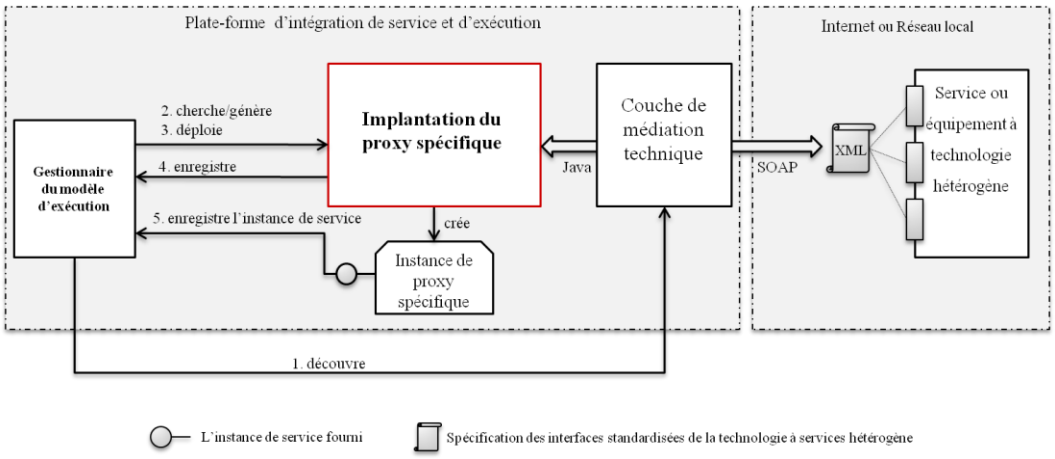


Figure 6.26. L'architecture de l'intégration de multiples technologies à services

De façon plus détaillée, et comme schématisé par la figure 6.26, l'intégration de services distribués et hétérogènes au sein d'une application se déroule comme suit :

1. ROSE découvre le départ ou l'arrivée de services sur le réseau ;

2. ROSE cherche/génère le « bon » proxy spécifique permettant d'invoquer le service externe découvert. Les « proxies » spécifiques sont généralement préalablement implantés afin d'invoquer certaines opérations du service découvert selon un besoin personnalisé. Les implantations de « proxy » spécifique sont stockées dans une base de données (pouvant être dédiée à une technologie particulière) locale ou distante. Dans le cas où aucun proxy correspondant ne serait disponible, un « proxy » spécifique peut être automatiquement généré à partir de la spécification du service externe. Par exemple, la spécification WSDL de service Web permet la génération de « proxy », certes sommaire en terme de qualité de service mais utilisable pour invoquer des opérations ;

3. ROSE déploie l'implantation du « bon » proxy spécifique correspondant au service externe découvert sur la plate-forme d'exécution ;

4. Le moteur d'interprétation instancie l'implantation du proxy spécifique afin de créer une instance du proxy configuré pour fonctionner avec le service externe correspondant ;

[53] Chameleon.ow2.org

5. Le moteur d'interprétation enregistre l'implantation du proxy de service spécifique dans le modèle d'exécution. Les implantations du proxy de service spécifique permettent de créer les instances de service en cas de besoin lors de l'exécution ;

6. Le moteur d'interprétation enregistre l'instance de service du proxy spécifique dans le modèle d'exécution. L'instance de service du proxy spécifique permet d'invoquer son service externe correspondant via les interfaces en Java transformées par la couche de médiation technique.

L'intégration de services implantés suivant la technologie UPnP est illustrée dans la figure 6.27. Le modèle d'exécution utilise le « BaseDriverUPnP » [23] afin de découvrir et notifier la disponibilité des équipements en UPnP. Dès qu'un nouvel équipement UPnP apparait sur le réseau local, l'annuaire cherche le proxy spécifique correspondant à cet équipement dans un registre distant. Ensuite, il déploie l'implantation du « bon » proxy spécifique sur la plate-forme d'exécution s'il en trouve. Puis, l'annuaire enregistre cette implantation du proxy spécifique dans le modèle d'exécution. En même temps, le moteur d'interprétation peut utiliser cette implantation du proxy spécifique afin de créer une instance fournissant le service qui est enregistré aussi dans le modèle d'exécution.

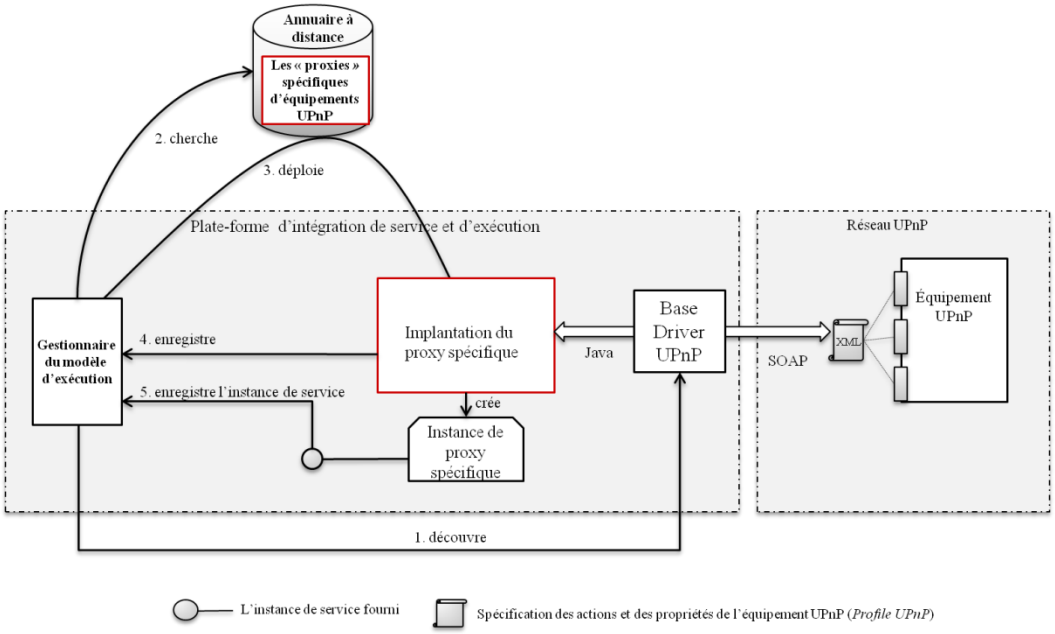


Figure 6.27. La réalisation de l'intégration de la technologie **UPnP**

L'intégration de services implantés suivant la technologie Web Service est réalisée suivant une architecture similaire. Dans ce cas particulier, le modèle d'exécution utilise **CXF** [112] afin de découvrir et notifier la disponibilité des services Web. Lorsqu'un service implanté en iPOJO souhaite invoquer un service Web disponible sur l'internet, une implantation du proxy spécifique correspondant au service Web est automatiquement générée par **CXF** à partir de la description **WSDL** de ce service Web. Ensuite, il déploie cette implantation du proxy spécifique sur la plate-forme d'exécution. Puis, l'annuaire enregistre cette implantation du proxy spécifique et l'instance de service fournie par le proxy spécifique dans le modèle d'exécution. Enfin, le service iPOJO est en mesure d'établir la liaison avec l'instance de service du proxy spécifique WS afin d'invoquer réellement le service Web via la couche de médiation technique - **CXF.**

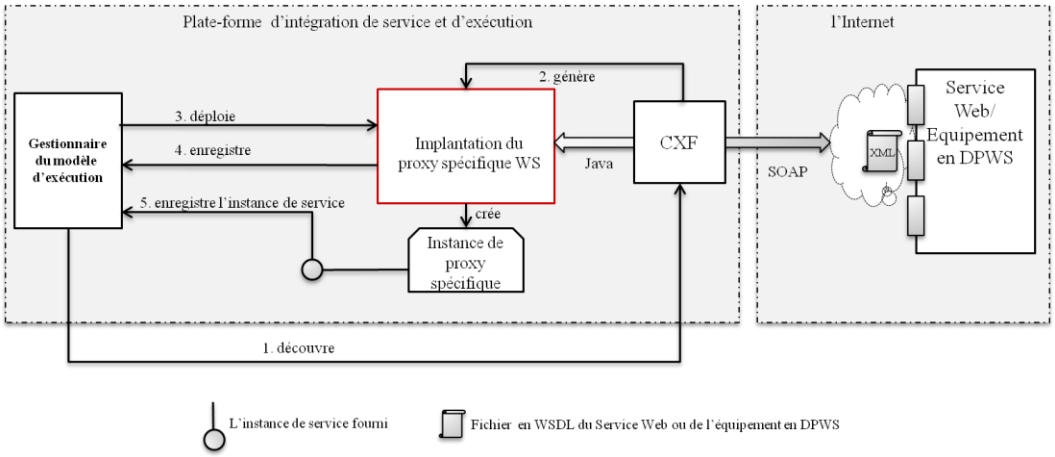


Figure 6.28. La réalisation de l'intégration de la technologie en Services Web ou DPWS

5.4. Moteur d'interprétation

Le moteur d'interprétation joue un rôle clé dans la machine d'exécution. Ses objectifs sont, d'une part, de créer l'application à l'exécution en fonction de l'architecture applicative, tout en considérant l'état courant de la plate-forme cible à l'aide du modèle d'exécution. D'autre part, il doit gérer dynamiquement l'évolution de l'application à l'exécution en conformité avec l'architecture applicative afin de s'adapter aux changements du contexte. Ces deux tâches sont basées sur la définition des variations au sein de l'architecture applicative permettant de retarder la prise des décisions de conception jusqu'au moment de l'exécution de l'application à services.

Dès que le moteur d'interprétation a fixé toutes les variations au sein de l'architecture applicative, l'application à l'exécution peut être créée par la mise en application d'une stratégie de sélection de services. Etant donné que la disponibilité des ressources et des services utilisées par l'application à l'exécution est susceptible de changer au cours du temps, le moteur d'interprétation doit reprendre les décisions de conception sur les variations architecturales en réappliquant la stratégie de sélection des services.

La stratégie de sélection des services respecte les deux objectifs suivants :

- **maximiser** la disponibilité des services constituant l'application à l'exécution,
- **économiser** les ressources de la plate-forme d'exécution.

Pour illustrer le fonctionnement du moteur d'interprétation, nous allons utiliser l'exemple de l'application du « rappel des rendez-vous avec les spécialistes ou les médecins » qui sera reprise, plus en détail, dans le chapitre suivant. Le schéma ci-dessous montre l'architecture applicative de notre exemple :

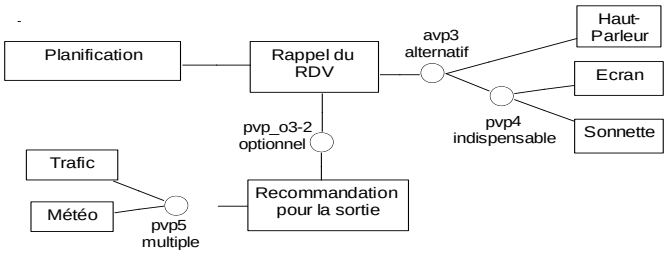


Figure 6.29. L'architecture de l'application du « rappel des rendez-vous avec les médecins »

5.4.1. Créer l'application à l'exécution

Le moteur d'interprétation crée l'application à l'exécution suivant l'architecture applicative (constituant le cadre des contraintes architecturales), tout en prenant en compte l'état courant de la plate-forme d'exécution. Plus précisément, le moteur d'interprétation met en application une stratégie de sélection de services en se basant sur l'architecture applicative afin de choisir les instances de services appropriées à l'aide du modèle d'exécution. Il génère également le « *glue code* » nécessaire afin de configurer les instances de service concrètes. Par exemple, le moteur d'interprétation génère le code nécessaire à la traçabilité des opérations d'une spécification de service. Finalement, le moteur d'interprétation assemble les instances de services sélectionnées pour construire l'application à l'exécution.

En prenant en compte les différents types de variabilité au sein de l'architecture applicative, nous allons préciser la stratégie de sélection de services utilisée par notre moteur d'interprétation :

- les variations causées par ***l'implantation de service.*** Ces variations correspondent aux différentes configurations possibles pour l'instanciation d'une même implémentation. Ainsi, dans le cas où plusieurs instances d'une même implantation de service sont disponibles en même temps, le moteur d'interprétation doit effectuer une sélection entre les instances de service disponibles. La stratégie mise en place pour cette sélection prend en compte les différentes caractéristiques de ces instances telles que la stabilité (le taux de déconnexion), la fiabilité (le taux de défaillance), la fréquence et l'utilisabilité d'une instance de service (les retours des utilisateurs). Plus concrètement, la stratégie mise en place dans le contexte de notre moteur d'exécution est de favoriser le choix des services « *pur* » iPOJO par rapport à des services externes afin d'augmenter la fiabilité de l'application obtenue ;
- les variations causées par ***la spécification de service***. Ces variations correspondent aux différentes implantations de service permettant de réaliser une même spécification de service. Ces implantations se distinguent au niveau de la version implantée, de la technologie à service utilisée, des propriétés non fonctionnelles et du mode de communication employé. La stratégie de sélection favorise la sélection des implantations de service possédant déjà une ou plusieurs instances disponibles, ainsi que les implantations les plus récentes. Cette stratégie est mise en place de la façon suivante :
  - tout d'abord, nous observons les implantations de service disponibles sur la plate-forme d'exécution à l'aide du modèle d'exécution ;
  - puis, le moteur d'interprétation sélectionne une version de l'implantation de service compatible avec toutes ses dépendances dans l'architecture applicative. Si plusieurs implémentations satisfont les critères liés aux dépendances, le moteur d'interprétation favorise la sélection de l'implémentation la plus récente ;
  - ensuite, dans le cas où l'implantation de service sélectionnée ne possède aucune instance sur la plate-forme d'exécution, le moteur d'interprétation instancie cette implémentation. Dans le cas où l'implantation de service sélectionnée possède déjà plusieurs instances disponibles, le moteur d'interprétation met en application la stratégie définie dans le point précédent.

  Finalement, dans le cas où aucune implémentation de service ne convient aux critères mentionnés précédemment, le moteur d'interprétation échoue et envoie un avertissement au technicien chargé de l'installation de l'architecture applicative.
- les variations causées par ***le point de variation topologique*** au sein de l'architecture applicative. Ce type de variabilité est représenté par les deux concepts *PrimitiveVariationPoint et AdvancedVariationPoint* du méta-modèle d'architecture. Les deux concepts définissent les variations au niveau des connexions de spécifications de service et des points de variations eux-mêmes. Le moteur d'interprétation doit choisir les candidats au point de variation entre les spécifications de service liées, en fonction d'une des logiques de variation définie comme *indispensable, alternatif, multiple* et *optionnel*. La stratégie de sélection favorise la sélection des spécifications de service possédant déjà une ou plusieurs implantations de service disponibles. Cette stratégie est mise en place de la façon suivante :
  - tout d'abord, nous observons les implantations de service correspondant aux spécifications de service liées au point de variation à l'aide du modèle d'exécution ;
  - puis, le moteur d'interprétation met en application la stratégie définie dans le point précédent afin de sélectionner les instances de service correspondant aux implantations de service choisies. Ici, nous précisons les deux cas particuliers :
    a) si aucune implémentation de service ne convient aux spécifications de service, le moteur d'interprétation échoue et envoie un avertissement au technicien chargé de l'installation de l'architecture applicative.
    b) si la stratégie ne peut pas différencier les instances de service disponibles selon les critères défini précédemment, le moteur d'interprétation effectue un choix arbitraire.

Dans l'exemple, au point de variation « avp3 » suivant la logique « alternatif », les instances de service « Ecran », « Sonnette » et « Haut-parleur » sont simultanément disponibles sur le réseau. Dans ce cas là, le moteur d'interprétation peut choisir l'instance de service « Haut-parleur » au point « avp3 ». Il peut aussi choisir les instances de service « Ecran » et « Sonnette » pour fournir la même fonctionnalité à la place de « Haut-parleur ». Ce choix est ainsi effectué par le moteur d'interprétation de façon arbitraire. En effet, l'expression de critères de sélection plus complexe est possible mais dépasse le cadre de notre proposition.

5.4.2. Gérer l'évolution de l'application à l'exécution

La gestion de l'évolution de l'application à l'exécution a pour l'objectif de permettre à cette application de s'exécuter constamment, tout en prenant compte les impacts des changements causés par la disponibilité des services et des ressources. Le moteur d'interprétation doit ainsi pouvoir maintenir dynamiquement l'application à l'exécution malgré les changements de l'environnement d'exécution. D'une part, il fait évoluer l'application à l'exécution dans le cadre de l'architecture applicative grâce à sa définition des variations. D'autre part, le moteur d'interprétation réalise également un mécanisme de la gestion du cycle de vie des instances de services afin de garantir l'intégrité de l'évolution de l'application à l'exécution en conformité avec l'architecture applicative.

Comme mentionnés précédemment, plusieurs événements peuvent influer sur la disponibilité de services, tels que l'intermittence de la connectivité de service, des pannes de la connexion de service et des nouvelles installations de services. Il faut noter que les ressources sont prises en compte comme les services dans le cadre de ce document. Ces changements de la plate-forme d'exécution sont surveillés et enregistrés par le modèle à exécution qui, de plus, est chargé de les notifier au moteur d'interprétation. Le moteur d'interprétation traite différemment les deux cas concernant la disponibilité de services, ci-après, à partir de l'état courant de la plate-forme et de l'historique de l'application à l'exécution :

- la ***disparition de l'instance d'un service*** constituant l'application à l'exécution. Par exemple, le service « Thermomètre » se déconnecte du réseau. Le moteur d'interprétation recherche alors l'instance d'un service disponible pouvant remplacer le service disparu. Il peut s'agir dans notre exemple du service web « Météo ». Cette nouvelle instance de service doit être capable de répondre entièrement à la fonctionnalité fournie par l'instance du service disparu « Thermomètre ». Puis, le moteur d'interprétation applique le mécanisme de gestion du cycle de vie des instances de services afin d'activer toutes les dépendances de l'instance du service ajoutée, tout en désactivant les dépendances de l'instance de service disparu.
  Dans notre exemple, le moteur d'interprétation doit charger d'abord l'implantation du proxy du service Web « Météo » et l'instancier. Ce proxy requiert l'implantation du service « Recommandation pour la sortie » avec la version 2.0. Mais, l'application en cours d'exécution utilise la version 1.0 de ce service. L'implantation du service « Thermomètre » utilisé dans l'application était en effet compatible avec les deux versions de l'implantation « Recommandation pour la sortie ». Dans ce cas, le moteur d'interprétation doit chercher l'implantation de « Recommandation pour la sortie » en version 2.0, et puis instancier ce service. En même temps, il doit désactiver l'instance de l'implantation en version 1.0.
- l'***apparition d'une l'instance du service*** répondant à une partie des fonctionnalités de l'application en cours d'exécution. Le moteur d'interprétation peut faire évoluer cette application afin d'en améliorer la qualité.
  Dans notre exemple, le service « Recommandation pour la sortie » est un service optionnel pour l'application « rappel des rendez-vous avec les médecins ». Initialement, ce service est disponible et peut fournir les informations supplémentaires concernant la météo au moment de la sortie et l'état actuel du trafic de la route pour aller à l'adresse de destination. A partir de ces informations, le service donne des conseils tels que la route la plus rapide, le meilleur moyen de

transport, etc. Si ce service n'est pas disponible sur le réseau, l'application donne seulement à l'utilisateur l'adresse de destination et l'heure pour son rendez-vous médical. Dès que le service « Recommandation pour la sortie » se reconnecte au réseau, le moteur d'interprétation peut faire évoluer l'application à l'exécution afin d'utiliser ce service. Cette évolution est réalisée par la reprise de la configuration antérieure à partir de l'historique des états de cette application à l'exécution dans le modèle d'exécution.

Dans ce cas, le moteur d'interprétation peut sauter l'étape de la sélection de services et faire directement l'évolution de l'application à services par la reprise d'une architecture antérieure de l'application à services. Cette évolution permet de s'adapter aux changements qui font la situation du contexte d'exécution revenir à un état antérieure enregistré dans l'historique de l'application à services.

6. Synthèse

Ce chapitre est dédié à la présentation du prototype *ChiSpace*. Celui-ci permet d'assister les différents intervenants de développement des applications à services pour réaliser l'approche proposée dans ce document. Nous avons utilisé une approche dirigée par les modèles pour mettre en œuvre *ChiSpace*. Nous avons choisi Le framework *EMF* (intégré dans la plate-forme *Eclipse*) comme la base de développement du prototype. Ce framework améliore la productivité du développement de notre prototype et d'applications à services.

Le prototype *ChiSpace* est constitué des trois outils correspondant aux trois phases de notre approche. Ces outils fournissent les fonctionnalités attendues par les différents intervenants:

- **l'outil d'ingénierie du domaine** correspond à la phase de développement du domaine. Cet outil, destiné aux experts du domaine et aux architectes, est divisé en deux parties :
  a) *l'environnement de modélisation* consacré à la réalisation de la conception du domaine. Cet environnement permet à l'expert du domaine et à l'architecte de créer les modèles à base de services afin de définir le domaine métier ;
  b) *l'environnement de développement de services* consacré à la création d'implantations de service. Il accélère dans une certaine mesure le développement de services grâce à ses générateurs de code dédiés aux différentes technologies à services.

- **l'outil d'ingénierie des applications** correspond à la conception d'applications particulières. Cet outil fournit un éditeur du modèle dédié au domaine étudié précédemment. Cet éditeur permet aux techniciens du domaine de décrire facilement l'architecture d'une application par la spécialisation d'une architecture du domaine déjà définie. De fait, cet éditeur est construit automatiquement par *EMF* suite à une transformation de modèles.
- **l'outil de la machine d'exécution** correspond à la phase de réalisation de l'application à services. Cet outil permet d'automatiser la construction de l'application à services à partir de l'architecture applicative (spécialisée lors de la deuxième phase de l'approche) et les services disponibles. Il permet également de gérer l'application à l'exécution sans intervention humaine afin de l'adapter au dynamisme du contexte. Cet outil est mise en œuvre en se basant sur la plate-forme iPOJO.

Ces trois outils ont été testés sur des cas d'utilisation liés à la santé et au maintien de personnes à domicile. Ils se sont révélés pertinents et efficaces, comme cela est montré dans le chapitre suivant. Les deux premiers outils ont aujourd'hui atteint un bon niveau de qualité et sont utilisables dans divers contextes. La machine d'exécution est aujourd'hui à l'état de prototype. Nous verrons dans la partie « conclusion » de cette thèse que des extensions sont en cours d'étude.

Chapitre 7 Validation

Les chapitres précédents sont dédiés à présenter l'approche proposée et l'outillage associé afin de développer les applications dynamiques à services, de la conception jusqu'à l'exécution. Les outils servant à la modélisation de telles applications lors des deux premières phases de l'approche sont détaillés et validés dans le chapitre 6. Dans ce chapitre, nous présentons le contexte dans lequel les outils produits durant mes travaux de thèse ont été validé. Ainsi, plusieurs applications ont été implantées et étudiées pour démontrer la validité de notre approche.

Le travail de cette thèse a été réalisé dans le domaine de l'informatique résidentielle ou du bâtiment intelligent. Les travaux ont été validés au sein d'un projet impliquant des collaborations industriels -- ANSO. Ce domaine se caractérise par une grande diversité d'applications, par exemple, les applications pour le divertissement, les applications surveillant la sécurité du domicile, les applications d'assistance ou de surveillance à distance d'un patient à domicile, les applications permettant de maîtriser la consommation d'énergie du domicile, etc.

Une partie de notre chapitre détaille les objectifs de la validation de notre approche et la méthodologie utilisée pour remplir ces objectifs. L'objectif est à la fois de valider la partie de conception des applications et la partie création et gestion de l'évolution de l'application au cours de l'exécution. Pour cela nous utilisons une méthode expérimentale.

Nous présentons ainsi deux applications pour assister et superviser un patient à domicile. Elles ont été implémentées en respectant scrupuleusement l'approche présentée dans les chapitres précédents et en utilisant le prototype fourni.

Nous détaillons, dans un premier lieu, la façon dont nous avons conçu les architectures applicatives en spécialisant l'architecture de référence. Le processus de spécialisation a été réalisé en suivant les règles de raffinement d'architecture définies dans le chapitre 5.

Puis, nous présentons comment ces applications à services sont automatiquement créées par la machine d'exécution à partir de leur architecture. Cette machine d'exécution permet ensuite de gérer les applications à services, de façon dynamique, pour s'adapter aux changements du contexte. Nous présentons également les scénarios que nous avons développés afin de vérifier que les adaptations attendus étaient effectivement réalisé.

Finalement, nous avons évalué les performances de l'exécution de la machine d'exécution sur les deux aspects suivants: a) la consommation de mémoire et de processeur de la machine d'exécution afin de créer et d'adapter l'application en cours d'exécution ; b) le temps moyen de réaction de la machine d'exécution pour effectuer une reconfiguration de l'application en cours d'exécution.

1. Contexte

Le travail de cette thèse s'intéresse au développement les applications dans le domaine de l'informatique résidentielle ou du bâtiment intelligent. Ce domaine de l'informatique se consacre aux aspects suivants: le confort, la sécurité, la sécurité des personnes et la maîtrise de l'énergie. La technologie « Smart Home » répond à ces objectifs de la façon suivante [107] :

- Le confort est accrue en automatisant les tâches fastidieuses et l'interface utilisateur est intuitive et bien conçue ;
- La sécurité est adressée par l'identification des mécanismes et des dispositifs de surveillance, par exemple, les caméras à l'extérieure. De plus, les mécanismes de notification contribuent également à la sécurité en permettant des réactions immédiates, par exemple, dans le cas d'une tentative de cambriolage ;
- La sécurité des personnes (*life safety en anglais*) correspond aux deux aspects : la sécurité de la vie quotidienne et le « healthcare ». Les capteurs peuvent, par exemple, détecter un danger causés par le feu ou l'électricité. La patrie « healthcare » correspond à la surveillance de l'état d'un patient à domicile. Il peut également fournir les services permettant de faire face à certaines situations anormales sur l'état de santé du patient. Par exemple, lorsque le capteur détecte une chute du patient, le système est en mesure d'envoyer un message à une infirmière par une alarme;
- La maîtrise de l'énergie à pour objectif de réduire les coûts de fonctionnement des dispositifs électroniques par une gestion intelligente de l'énergie. Ce type de gestion permet par exemple d'éviter de chauffer ou d'éclairer un bâtiment vide réduisant ainsi le gaspillage d'énergie. Ce domaine permet également d'ajuster dynamiquement le confort de la maison en fonction de paramètres d'économie d'énergie qui peuvent être définie par les habitants.

Cette thèse a fait partie des développements du projet européen ITEA ANSO[55]. Ce projet a pour objectif de développer une plate-forme open source, intelligente et fiable pour les différents environnements du réseau domotique. Cette plate-forme permet d'accélérer grandement le développement des applications à services dans le contexte domotique. Elle permet également leurs compositions dans des applications innovantes pour accroître l'exploitation des services pour la maison numérique en Europe. Pour cela, ANSO projette de résoudre les contraintes technologiques en créant des standards pour les dispositifs et un intergiciel distribué universel permettant un accès homogène à des services multimédia hétérogènes, et supportant l'ensemble des dispositifs disponibles dans un tel environnement. Le projet ANSO s'intéresse également à la création d'un intergiciel distribué autonomique, qui fournit un environnement de développement ouvert pour les applications domotiques. Une partie de cet intergiciel, la plate-forme H-Omega, a été réalisé au sein de l'équipe ADELE lors de la thèse de Johann Bourcier [108].

La contribution principale de cette thèse est d'étudier et de développer une infrastructure logicielle facilitant le développement et la gestion des applications à services dans un environnement dynamique, distribué et hétérogène. Ce travail a été réalisé en se basant sur la plate-forme H-OMEGA développé dans le contexte du projet ANSO. Nous avons ainsi développé plusieurs applications domotiques, fonctionnant sur la plate-forme H-OMEGA, grâce au prototype *ChiSpace* représenté dans le chapitre précédent. Ce prototype est constitué des trois outils : deux assistant le design des applications domotiques et le dernier pour automatiser la création et la gestion de l'application lors de l'exécution.

[55] ANSO : ***A**utonomic **N**etwork for **S**mall **O**ffice Home **O**ffice*

2. Objectifs et méthodologies

L'objectif de notre validation est double. Nous souhaitons à la fois valider l'intérêt de la partie conception d'applications, avec la phase d'ingénierie du domaine et la phase d'ingénierie applicative, et la partie exécution et gestion de ces applications à services. Nous avons décidé d'utiliser une méthode expérimentale pour valider l'ensemble de notre approche. En effet, nos outils ont été utilisés dans le contexte du projet ITEA ANSO présenté ci-dessus et dans des expérimentations internes à l'équipe ADELE.

Etant donné la diversité des applications du domaine domotique, il est extrêmement complexe de produire une seule architecture commune du domaine. Nous avons ainsi tout d'abord élaboré une architecture de référence générique du domaine de l'informatique résidentielle. Nous avons par la suite raffinée cette architecture pour le domaine des applications se rapportant à la sécurité des personnes à leur domicile. Cette architecture raffinée est présentée dans le chapitre 5. La définition des architectures de référence du domaine est réalisée lors de l'ingénierie du domaine dans l'outil dédié. Nous avons par la suite généré l'outil dédié au domaine spécifique permettant de développer facilement nos applications. Nous utilisons ainsi cet outil pour concevoir les deux applications qui sont présentées dans la suite de ce chapitre. L'utilisation de nos outils pour définir ces architectures applicatives nous permet de valider expérimentalement la partie conception d'applications de notre approche.

Nous avons par la suite exécutés ces applications sur la machine d'exécution. L'objectif de cette évaluation est à la fois de montrer que la machine d'exécution implémentée fonctionne correctement et que ce prototype n'introduit pas de surcoût important lors de l'exécution. Nous voulons ainsi montré la validité globale de notre approche. En effet, si la machine d'exécution engendre un surcoût excessif lors de l'exécution, l'approche ne sera pas validée car l'impact sur l'exécution des applications doit rester minime. Nous introduisons ainsi des scénarios d'exécution pour les deux applications afin de tester notre machine d'exécution dans différents contextes. Nous instrumentons la machine d'exécution afin d'observer son comportement, la surcharge de consommation de ressources et le temps de réaction en présence d'événements nécessitant des reconfigurations de complexités variées des applications.

3. Applications

L'objective de cette section est de présenter les deux applications que nous avons développé pour valider l'approche proposée dans ce document. Ces deux applications sont développées en utilisant le prototype – *ChiSpace*. Nous avons commencé par décrire les architectures applicatives de ces 2 applications à l'aide de notre outil. Ces architectures sont définies en respectant scrupuleusement les règles de raffinement utilisées pour spécialiser l'architecture de référence définie dans le chapitre 5. Nous utilisons ensuite notre prototype de la machine d'exécution pour automatiser la construction de ces applications. L'adaptation dynamique de l'application à services en fonction de l'architecture est également gérée par la machine d'exécution.

Dans les deux chapitres précédents, nous avons présentés plusieurs architectures de référence du domaine de l'informatique résidentielle en raison de la diversité des applications domotiques. Une architecture de référence très générique correspond à l'ensemble du domaine domotique et deux architectures de référence plus raffines visent deux aspects particuliers du domaine résidentiel. Nous avons ainsi pu développer plusieurs applications domotiques en utilisant ces architectures, telles que des applications de sécurité des personnes et des applications permettant la maîtrise de la consommation énergétique d'une maison.

Dans ce chapitre nous ne présentons que deux applications assistant et supervisant un patient à domicile. Ces deux applications correspondent à l'aspect de la sécurité des personnes introduit précédemment. Elles nous servent à illustrer le fonctionnement de notre approche et à évaluer l'impact de notre moteur d'exécution sur les performances des applications.

3.1. Rappel de l'architecture de référence dédiée aux applications médicales

La figure 7.1 illustre l'architecture de référence dédiée aux applications médicales. Elle permet de créer des applications pour surveiller l'état d'un patient ou pour l'assister dans sa vie quotidienne. Cette architecture est composée de capteurs, services et actionneurs médicaux, des dépendances entre ces différents services ainsi que de points de variabilité. Il est important de noté que cette architecture reste abstraite : elle ne permet pas d'instancier directement une application sans effectuer un ensemble de choix. Les capteurs et les services environnementaux sont des éléments optionnels pour constituer la structure d'une application permettant la surveillance d'un patient à son domicile. Les points de variation topologique sont utilisés pour planifier les variations structurales au sein de l'architecture.

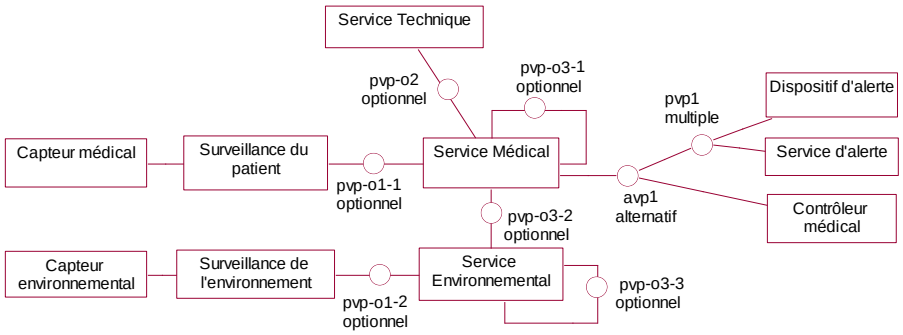


Figure 7.1. L'architecture de référence raffine dédiée aux applications médicales

3.2. Application de rappel de prise des médicaments

La première application développée a pour objectif de rappeler au patient de prendre ses médicaments selon la prescription donnée par son médecin. L'application utilise les dispositifs d'alerte, tels que « Haut-parleur », « Écran » et « Sonnette », afin de rappeler au patient l'heure, le dosage des médicaments à prendre. Cette application permet d'assister le patient dans sa vie quotidienne.

3.2.1. L'architecture applicative

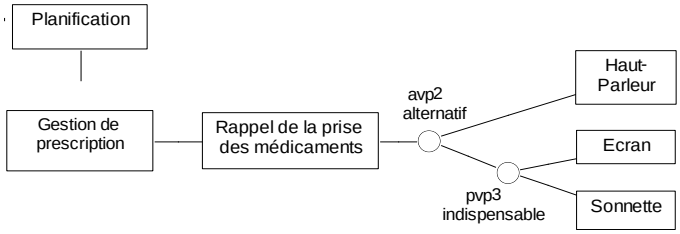


Figure 7.2. Application de rappel de prise des médicaments

L'architecture applicative est définie lors de la deuxième phase de notre approche en spécialisant l'architecture de référence dédiée aux applications médicales. Ce processus est réalisé grâce à notre prototype *ChiSpace*. L'architecture de cette application (illustrée dans la figure 7.2) est constituée d'un service technique, de services médicaux et d'actuateurs:

- **les services médicaux.** Le service « Gestion de prescription », hérite de « Service médical » et permet d'accéder à la base de données et de récupérer la prescription à partir du dossier du patient. Il rappel précisément l'heure et le dosage des médicaments, au service « Rappel de la prise des médicaments ». Ce service hérite également de « Service médical ». Le service « Gestion de prescription » interprète les prescriptions, afin de configurer le service technique « Planification » avec les horaires de prise de médicaments. Ce service technique peut ensuite planifier les différents rappels pour la prise des médicaments via le service de « Gestion de prescription » ;
- **le service technique**. Le service de « Planification » hérite de « Service technique » et est en mesure de notifier le service de « Gestion de prescription » au "bon" moment pour informer le patient ;
- **les dispositifs d'alerte.** Finalement le service de « Rappel de la prise des médicaments » utilise un « Haut-parleur » ou un « Écran » et une « Sonnette », afin d'informer le patient. Ces différents services héritent tous de « Dispositif d'alerte ».

Spécialisation de l'architecture de référence

L'architecture de référence illustrée dans la figure 7.1 a été spécialisée en suivant les règles de raffinement précisées dans le chapitre 5. Cette spécialisation s'est faite de la façon suivante :

- en éliminant un point de variation. Les points de variation « pvp-o1-1 » et « pvp-o1-2 » au sein de l'architecture de référence sont éliminés, puisque la logique du type de variabilité des deux points de variation correspond à « optionnel ». Il signifie que les capteurs et les médiateurs médicaux et environnementaux ne servent pas à constituer cette application ;
- en éliminant un point de variation. Au point de variation avancé « avp1 », le service « Contrôleur médical » n'est pas choisi pour constituer l'application. Puisque la logique du type de variabilité au « avp1 » correspond à « alternatif », « Dispositif d'alerte » et « Service d'alerte » sont

sélectionnés afin d'être candidats à « pvp1 ». Ce point de variation étant de logique « multiple », nous avons sélectionnés « Dispositif d'alerte » en tant qu'actuateur pour communiquer avec l'utilisateur.

- en ajoutant les nouvelles dépendances. La connexion entre « Rappel de la prise des médicaments » héritant du « Service médical » et « Dispositif d'alerte » est reprise par ses sous-types, tels que « Haut-parleur », « Écran » et « Sonnette ».
- en ajoutant un point de variation. Le point de variation primitif « pvp3 » selon le type de variabilité « multiple » est ajouté au sein des connexions entre « Rappel de la prise des médicaments » et « Haut-parleur », « Écran » et « Sonnette » ;
- en décomposant un point de variation. Le point de variation « pvp3 » est décomposé en « avp2 » et « pvp3 » par le regroupement des trois actuateurs « Haut-parleur », « Écran » et « Sonnette » en deux ensembles.
- en modifiant un point de variation. Le type de variabilité du « pvp3 » définissant la variation au niveau des connexions entre « Rappel de la prise des médicaments » et « Écran » ainsi que « Sonnette » est modifié en logique « indispensable ». Cette modification, en fonction de la règle de la modification présentée dans le chapitre 5, signifie que les deux dispositifs d'alerte doivent fonctionner ensemble. En utilisant la même règle, le type de variabilité au point de variation avancé « avp2 » est modifié en « alternatif ». Cela signifie que « Rappel de la prise des médicaments » utilise soit « Haut-parleur », soit « Écran » et « Sonnette », pour afficher la prescription de la prise des médicaments au patient ;
- en ajoutant la dépendance. La connexion entre « Service Technique » et « Service Médical » est reprise par « Planification » et « Gestion de prescription », puisque le « Planification » hérite du « Service Technique » et « Gestion de prescription » hérite du « Service Médical ». Ensuite, le « pvp-o2 » est éliminé au niveau de la connexion entre les deux services précédents.

3.2.2. Création et gestion de l'évolution de l'application

Dans cette partie, nous décrivons un scénario d'exécution pour cette application. Un scénario décrit un contexte initial et un ensemble d'évènements qui vont arriver lors de l'exécution. Nous décrivons ainsi comment la machine d'exécution va réagir à ces différents changements de contexte et reconfigurer dynamiquement l'application en se basant sur l'architecture applicative. Dans ce scénario, nous étudions différents événements ayant attrait à la disponibilité des services, à savoir les événements d' « apparition » et de « disparition » de service. Dans ce scénario, les deux formes de variabilité principales sont utilisées au sein de l'architecture applicative présentée : les variations au niveau de point de variation topologique et de spécification de service ayant plusieurs implantations.

Le contexte initial est le suivant : les implantations et les instances de service suivantes sont disponibles sur la plate-forme d'exécution:

- les implantations de service : « Planification », « Rappel de la prise des médicaments » ;
- les instances de service : « Haut-parleur », « Gestion de prescription », « Sonnette » ;
- le service « Écran » n'est pas disponible.

Création de l'application à partir du modèle de l'architecture :

Premièrement, le moteur d'interprétation prend l'architecture applicative en entrée. Puis il recherche les services disponibles incluant les instances et les implantations de service, dans le modèle d'exécution. Ces services doivent satisfaire les spécifications de service utilisées dans l'architecture applicative. Deuxièmement, ce moteur d'interprétation, selon la disponibilité des services et l'architecture applicative, utilise les implantations de « Rappel de la prise des médicaments » et « Planification » afin de créer les instances respectives de service. Finalement, il assemble les instances de service disponibles,

telles que « Planification », « Rappel de la prise des médicaments », « Haut-parleur » et « Gestion de prescription » pour construire l'application à l'exécution. Malgré le fait que l'instance de service « Sonnette » soit disponible et satisfasse à une partie de l'architecture applicative, elle n'est pas utilisée dans la configuration initiale de l'application. Ce choix est réalisé à cause de la définition du point de variation « pvp3 » de type de variabilité « indispensable » : l'utilisation de « Sonnette » requiert obligatoirement une instance de service « Écran ».

La gestion de l'évolution de l'application à l'exécution en fonction des variabilités définies au sein de l'architecture applicative :

Événement 1: l'instance de service « Haut-parleur » disparaît ; le service « Écran » fourni par un équipement UPnP apparait sur le réseau.

Étant donné la déconnexion du service « Haut-parleur », l'application ne peut plus utiliser ce service afin d'alerter l'utilisateur final. D'autre part, le service « Écran » fourni par un équipement UPnP est détecté de façon dynamique, et devient disponible à l'aide du gestionnaire du modèle d'exécution. D'abord, le gestionnaire du modèle d'exécution recherche et sélectionne un proxy, en tant qu'implantation de service correspondant à cet équipement dans le registre de services à distance en UPnP. Puis il déploie ce proxy choisi et mémorise cette information. A la fin, le moteur d'interprétation est notifié de ce changement par ce gestionnaire du modèle d'exécution. Par conséquent, le moteur d'interprétation emploie cette implantation de service comme « factory » afin de créer une instance de service et d'enregistrer cette instance dans le modèle d'exécution. Étant donné que « Écran » requiert « Sonnette » afin de communiquer avec l'utilisateur final, le moteur d'interprétation intègre donc les deux instances de service dans l'application en cours d'exécution. Cela permet à l'application de s'adapter dynamiquement au changement de contexte provoqué par cet événement. De plus, il permet d'assurer l'intégrité de l'application à l'exécution par rapport à l'architecture applicative définie par le développeur.

Événement 2: l'équipement UPnP fournissant le service « Écran » disparaît du réseau.

Le point de variation « avp2 » de type de variabilité « alternatif » définit que l'utilisation du service « Rappel de la prise des médicaments » requiert soit un « Haut-parleur », soit un « Écran » et une « Sonnette ». Ainsi, l'application à l'exécution doit s'arrêter quand l'équipement « Écran » disparaît du réseau, alors que « Haut-parleur » n'est pas encore connecté au réseau. Dans ce cas, la machine d'exécution envoie un message d'alerte afin d'avertir l'administrateur de cette situation anormale de l'application à l'exécution.

Événement 3: le service « Haut-parleur » réapparaît sur la plate-forme d'exécution.

L'instance de service « Haut-parleur » réapparaît sur la plate-forme d'exécution ; la plate-forme d'intégration de service et d'exécution la détecte et notifie ce changement au moteur d'interprétation. Celui-ci doit réparer l'application à l'exécution, tout en prenant en compte le fait que ce changement permet de reconfigurer l'état de l'application pour la rendre identique à une configuration précédente de l'application. En conséquence, le moteur d'interprétation intègre de nouveau l'instance de service « Haut-parleur » dans l'application à l'exécution. Ainsi, l'application à l'exécution redémarre et fonctionne correctement. Il est important de noté que la sonnette ne fait plus partie de l'application à l'exécution.

Événement 4: Le service « Écran » précédent réapparait sur le réseau et un autre service « Écran » fourni par un dispositif mobile conforme à la technologie iPOJO se connecte au réseau.

Un téléphone portable Nokia fournissant un service « Écran » se connecte au réseau. La plate-forme d'intégration de service et d'exécution détecte ce nouveau service et notifie ce changement au moteur d'interprétation. De ce fait, le moteur d'interprétation trouve deux implantations différentes du

service « Écran » disponibles sur la plate-forme d'exécution. Une des implantations de service « Écran » est un « proxy » de l'équipement UPnP. L'autre est une implantation de service en iPOJO. « Haut-parleur » et « Sonnette » sont disponibles en même temps dans la plate-forme d'exécution. Dans ces conditions, plusieurs choix sont possibles au niveau du point de variation topologique et au niveau de l'implantation de service. Premièrement, en fonction de la stratégie de sélection au niveau des points de variation topologiques introduite dans le chapitre précédent, le moteur d'interprétation sélectionne l'alternative comprenant les services « Écran » et « Sonnette », de façon arbitraire, car « avp2 » correspond au type de variabilité « alternatif ». Puis, le moteur d'interprétation sélectionne le service « Écran » fourni par le téléphone portable car il est réalisé en iPOJO. Cette sélection de variations respecte la stratégie de sélection de services au niveau de l'implantation de service, qui favorise les implémentations homogènes (iPOJO plutôt qu'UPnP). Finalement, le moteur d'interprétation reconfigure l'application à l'exécution, tout en considérant son intégrité en fonction de son architecture.

3.2.3. Résultat

La machine d'exécution, composée du moteur d'interprétation et du gestionnaire du modèle d'exécution, est réalisée au-dessus de la plate-forme d'exécution H-Omega. Pour évaluer les résultats obtenus par la machine d'exécution, nous avons développé le scénario défini ci-dessus pour cette application particulière. Les résultats sont obtenus sous forme de courbes de la consommation du processeur et de la mémoire au cours de l'exécution. L'objectif est d'illustrer les performances de la machine d'exécution. Les courbes montrent également le temps moyen d'adaptation aux changements de contexte. Finalement, les résultats montrent que notre machine d'exécution n'a que peu d'impact sur la performance de la plate-forme H-Omega et que les temps de réaction aux différents événements sont acceptables par rapport aux applications considérées.

La machine d'exécution fonctionne sur un ordinateur portable pouvant accéder à l'Internet via le réseau local. La vitesse du processeur et la capacité de mémoire du portable correspondent respectivement à 1,73GHz et1, 5Go.

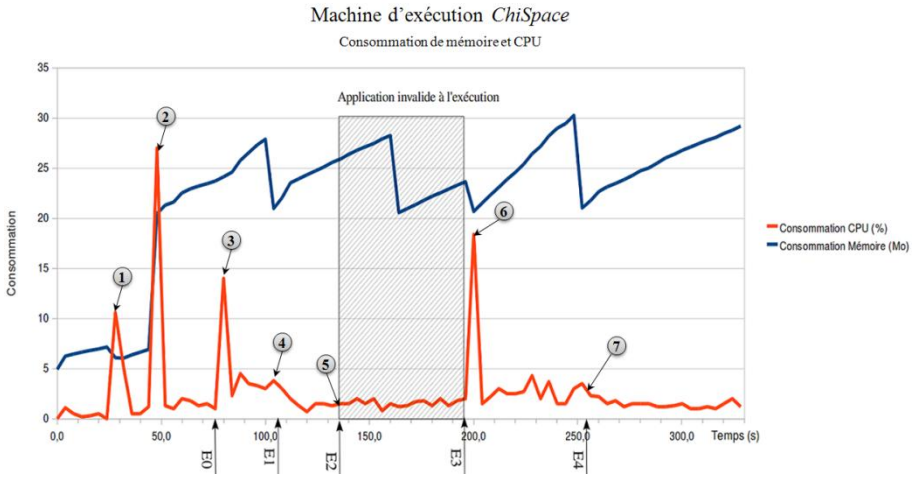


Figure 7.3. la consommation de mémoire et CPU

La figure 7.3 illustre un diagramme représentant la consommation en ressources de la machine d'exécution :

- l'axe des abscisses représente le déroulement du temps d'exécution en seconde,
- l'axe des ordonnées représente la consommation en ressources de la machine d'exécution,
- la courbe bleue représente la valeur mesurée de la consommation de mémoire en Mégaoctet,

- la courbe orange représente la valeur mesurée du pourcentage d'utilisation du processeur.

Les événements causant les changements sont pointés par des flèches numérotées, par exemple « E0 », « E1 » etc., sur l'axe des abscisses du déroulement de temps de l'exécution. Les points numérotés sur la courbe en orange représentent l'utilisation du processeur en réaction aux changements de contexte selon le scénario :

- Le premier point représente le pic de consommation associé au chargement de la plate-forme d'exécution H-Omega et de la machine d'exécution.
- Le deuxième point représente le pic de consommation associé à l'établissement de l'état initial du scénario.
- Le troisième point représente le pic de consommation lié au chargement du modèle de l'architecture applicative et à la création de l'application. Le moteur d'interprétation est chargé d'interpréter l'architecture applicative en entrée, tout en analysant les instances et les implantations de service utiles et disponible afin de créer l'application à l'exécution.
- Le quatrième point représente le pic de consommation induit par le premier événement – « Haut-parleur » disparu et « Écran » apparu. « Écran » étant fourni par un équipement UPnP, le moteur d'interprétation charge l'implantation de « proxy » correspondante : un composant iPOJO. Ensuite, il crée une instance de ce « proxy » à partir de son implantation. La courbe illustre ainsi que cette opération n'engendre qu'un faible surcoût en termes de charge CPU sur notre plate-forme.
- Le cinquième point représente le pic de consommation au moment de l'apparition du deuxième événement : « Ecran » disparu. Du fait que ni « Ecran » ni « Haut-parleur » ne sont disponible, l'application à l'exécution ne fonctionne plus en conformité avec l'architecture applicative. La zone hachurée représente la durée ou l'application ne fonctionne pas. Nous pouvons constater que la consommation de ressource est très faible dans l'intervalle ou l'application n'est pas disponible. Ceci est dû au fait que le mécanisme pour détecter l'apparition d'un nouveau service est un mécanisme passif : le moteur d'exécution attend simplement la notification qu'un nouveau service est disponible. Ce mécanisme ne produit donc pas de surcoût, alors qu'un mécanisme de recherche actif produirait un grand surcoût.
- Le sixième point représente le pic de consommation au moment de l'apparition du troisième événement du scénario : « Haut-parleur » apparaît de nouveau sur le réseau. Le moteur d'interprétation ne recalcule pas la configuration de l'application, puisque le modèle d'exécution a enregistré dans l'historique un état identique de la plate-forme permettant ainsi de recréer l'application à l'identique. Cependant, l'implantation du service « Haut-parleur » est assez « lourde » : elle utilise une librairie de transformation de texte en flux sonore. Le moteur d'interprétation devant créer une instance de ce service, la courbe montre donc un pic de consommation du processeur causé par la création de cette instance.
- Le septième point représente le pic de consommation au moment de l'apparition du quatrième événement : un autre « Ecran » fourni par un dispositif mobile est apparu. Le moteur d'interprétation doit prendre une décision de conception entre les deux implantations de service. Ce type d'adaptation au changement de contexte consomme peu de ressource du processeur.

Le reste de la courbe de la consommation du processeur représente l'état stabilisé de la machine d'exécution et de l'application à l'exécution qui fonctionne continuellement et correctement.

La courbe bleue montre que la valeur moyenne de la consommation de mémoire est stable au cours de l'exécution de l'application (environ 25 Mo). Les décroissances soudaines représentent la libération de mémoire causée par l'exécution du ramasse miette de la « JVM ». La stabilité de cette courbe montre que notre plate-forme d'exécution à une consommation mémoire constante au cours du temps. La comparaison entre la consommation mémoire avant et après le point 1, chargement de la plate-forme H-OMEGA et chargement de notre machine d'exécution, montre le surcoût mémoire de notre machine d'exécution. Le surcoût est très faible et n'est pas visible sur ce graphique. Le pic d'utilisation

mémoire au point 2 correspond au déploiement du scénario initial qui met en jeu le service « haut-parleur », qui lui-même fait intervenir une librairie permettant de transformer le texte en voix. Cette dernière librairie est très gourmande en ressources. On se rend donc compte que l'essentiel de la charge mémoire dans cet exemple est induit par l'application et non par notre machine d'exécution.

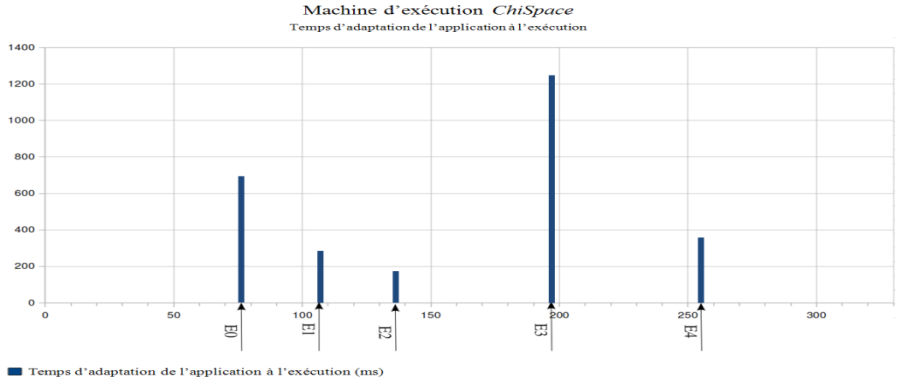


Figure 7.4. Le moyen du temps d'adaptation de l'application à l'exécution

Le diagramme (dans la figure 7.4) illustre le temps d'adaptation moyen de l'application à l'exécution correspondant aux différents événements. Ce test est réalisé automatiquement grâce à l'instrumentation de la machine d'exécution. L'axe des abscisses, dans le diagramme, correspond au déroulement du temps d'exécution en seconde. L'axe des ordonnées correspond au temps d'adaptation de l'application à l'exécution en microseconde. Les différent événements de création de l'application et d'adaptation au changement de contexte sont explicitement indiqués par les flèches numérotées « E0 », « E1 », etc. Les valeurs du temps d'adaptation représenté sur ce schéma correspondent à une moyenne sur 50 exécutions du scénario. Nous constatons que les temps de réponse sont très variables en fonction des événements considérés et des réactions engendrées. Cependant, si nous ne prenons pas en compte l'événement E3 qui engendre un long temps de réaction dû à la longue instanciation de la librairie permettant de convertir du texte en parole, le temps de réponse à un événement de changement de contexte est inférieur à 400ms ce qui nous paraît très acceptable compte tenu du domaine d'application considéré.

3.3. Application de rappel des rendez-vous avec les médecins

L'application de rappel des rendez-vous avec les médecins a pour objectif d'avertir le patient d'un rendez-vous avec les médecins. Cette application utilise également « Haut-parleur », « Écran » et « Sonnette » afin de communiquer avec l'utilisateur final comme l'application précédente. Elle emploie également le même service de la planification pour organiser l'horaire des rendez-vous. De plus, cette application peut fournir des informations supplémentaires, de façon optionnelle, concernant les moyens de transport et la météo. Cette application à pour but de montrer l'exécution d'une application avec une architecture plus complexe et incluant de nouveau type de variation telle que le point de variation optionnel. Elle montre également l'intégration des technologies hétérogènes à services, telles que service Web et UPnP, dans la plate-forme d'exécution.

3.3.1. L'architecture applicative

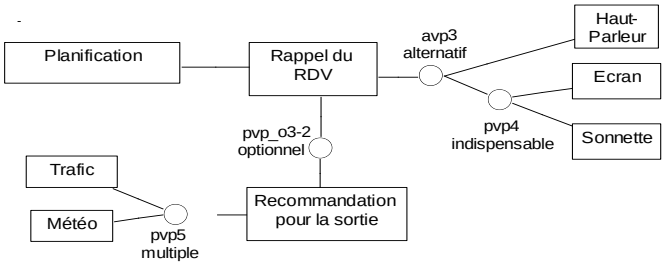


Figure 7.5. L'architecture de l'application de rappel des rendez-vous avec les médecins

L'architecture applicative est définie, lors de l'ingénierie applicative, par la spécialisation de l'architecture de référence (figure 7.1). En plus, de l'architecture applicative précédente, cette architecture applicative est constituée de services environnementaux « Recommandation pour la sortie », « Trafic » et « Météo », d'un service médical « Rappel du RDV » :

- les services environnementaux : « Recommandation pour la sortie », héritant de « Service environnemental », permettent de donner des informations concernant la sortie, telles que, la météo et l'état courant du trafic. « Météo » et « Trafic » héritant aussi de « Service environnemental », sont normalement disponibles comme services Web sur l'Internet. « Recommandation pour la sortie » peut synthétiser ces informations afin de donner une indication pratique pour la sortie de l'utilisateur. En général, cela fournit un service supplémentaire, mais pas essentiel, pour l'application de rappel des rendez-vous avec les médecins.
- le service médical : « Rappel du RDV », héritant de « Service médical », permet de configurer « Planification ». Il permet également d'avertir l'horaire du rendez vous à l'utilisateur final par « Haut-parleur » ou « Écran » et « Sonnette ». Il récupère des informations pratiques de « Recommandation pour la sortie », puis les affiche à l'utilisateur final.

Spécialisation de l'architecture de référence:

- en ajoutant les nouvelles dépendances : la connexion entre « Rappel du RDV » et « Recommandation pour la sortie » est ajoutée par la reprise de la dépendance entre « Service médical » et « Service environnemental ». De plus, le point de variation « pvp-o3-2 » selon la logique « optionnel » est gardé au sein de la connexion entre « Rappel du RDV » et « Recommandation pour la sortie ». Les nouvelles connexions entre « Recommandation pour la sortie », « Trafic » et « Météo » par la reprise de la connexion réflexive de « Service environnemental ».
- en ajoutant un point de variation : le point de variation primitif « pvp5 » est ajouté afin de définir la variation au sein des connexions entre « Recommandation pour la sortie », « Trafic » et « Météo ». Cet ajout est constitué de plusieurs modifications concrètes. Dans un premier lieu, « pvp-o3-3 » est séparé en deux points de variation suivant la logique « optionnel ». Ils décrivent la variation respective de la connexion de « Recommandation pour la sortie » à « Trafic », et de la connexion de « Recommandation pour la sortie » et « Météo ». Ensuite, ces deux variantes sont groupées dans un seul ensemble et puis le point de variation « pvp5 » est ajouté. A la fin, la logique de « pvp5 » est modifiée en « multiple » suivant la règle de la modification d'un point de variation. De fait, cette variation signifie que « Recommandation pour la sortie » peut dynamiquement ajouter les informations pratiques concernant « Trafic » ou (et) « Météo » dans l'application d'exécution.

Les points de variation « avp3 » et « pvp4 » sont définis par l'adoption des mêmes règles de raffinement que pour la définition de « avp2 » et « pvp3 » dans l'application précédente. Les points de variation, tels que « pvp-o1-1 », « pvp-o1-2 » et « pvp-o3-1 » dans l'architecture de référence, sont éliminés. Concrètement, les capteurs médicaux et environnementaux ainsi que leur médiateur dédié ne sont pas sélectionnés pour constituer cette architecture applicative. « Planification » est un élément obligatoire de l'architecture applicative.

3.3.2. Création et gestion de l'évolution de l'application

Un scénario dédié à cette application est développé avec trois événements successifs de changement dynamique sur la disponibilité des services.

Le contexte initial est le suivant : les implantations et les instances de service suivantes sont disponibles sur la plate-forme d'exécution:

- les implantations de service: « Rappel du RDV », « Recommandation pour la sortie » ;
- les instances de service: « Planification », « Haut-parleur », « Sonnette » ;
- les services «Trafic », «Météo », « Écran » ne sont pas disponible.

Création de l'application à partir du modèle de l'architecture :

Premièrement, le moteur d'interprétation prend l'architecture applicative en entrée. Puis il recherche les services disponibles incluant les instances et les implantations de service, dans le modèle d'exécution. Ces services doivent satisfaire les spécifications de service utilisées dans l'architecture applicative. Deuxièmement, ce moteur d'interprétation, selon la disponibilité des services et l'architecture applicative, utilise l'implantation de « Rappel du RDV » afin de créer une instance de service. Deuxièmement, ce moteur d'interprétation, selon la disponibilité des services et l'architecture, utilise l'implantation de « Rappel du RDV » afin de créer une instance de service. A la fin, il assemble les instances de service : telles que « Planification », « Rappel du RDV » créée par l'implantation de service correspondante et « Haut-parleur », pour construire l'application à l'exécution.

Malgré le fait que l'implantation de service « Recommandation pour la sortie » et l'instance de service « Sonnette » soient disponibles et satisfassent à une partie de l'architecture applicative, elles ne sont pas utilisées dans la configuration initiale à cause des raisons suivantes. Premièrement, « Recommandation pour la sortie » est un élément optionnel dans le modèle de l'architecture. En même temps, aucune dépendance de cette implantation de service, par exemple, « Météo » ou « Trafic » n'est pas disponible lors de la création de l'application. Troisième, le moteur d'interprétation n'a pas utilisé l'instance de service « Sonnette », puisque son utilisation requiert une instance de service « Écran ».

La gestion de l'évolution de l'application à l'exécution en fonction des variabilités définies au sein de l'architecture applicative:

Événement 1: l'instance de service « Haut-parleur » disparaît ; le service « Écran » fourni par un équipement UPnP apparaît sur le réseau.

Étant donné la déconnexion de « Haut-parleur », l'application ne peut plus utiliser ce service afin de rappeler et de montrer les informations à l'utilisateur final. D'autre part, le service « Écran » fourni par un équipement UPnP est dynamiquement détecté et devient disponible. Un « proxy » est automatiquement recherché, déployé et mémorisé dans le modèle d'exécution. A la fin, le moteur d'interprétation est notifié de ce changement. Par conséquent, le moteur d'interprétation emploie cette implantation de service comme une « factory » afin de créer une instance de service et d'enregistrer cette instance dans le modèle d'exécution. Puisque « Ecran » requiert le service « Sonnette » afin de

communiquer avec l'utilisateur final, le moteur d'interprétation intègre donc les deux instances de service dans la configuration précédente de l'application à l'exécution. Cela permet à l'application de s'adapter dynamiquement au changement de contexte provoqué par cet événement. De plus, il permet d'assurer l'intégrité de l'application.

Événement 2: « Météo » apparaît sur le réseau en tant que service Web.

Grâce à l'apparition du service Web « Météo », l'application de rappel des rendez-vous avec les médecins peut fournir l'information supplémentaire de météo. Premièrement, la plate-forme d'intégration de service et d'exécution peut le détecter et ensuite générer automatiquement un « proxy » correspondant à ce service Web en iPOJO. Le « proxy » permet aux autres services d'invoquer les opérations fournies par ce service Web. Deuxièmement, le modèle d'exécution enregistre ce proxy généré comme une implantation de service, tandis que le moteur d'interprétation est notifié de l'apparition de ce nouveau service disponible. Le « proxy » est alors instancier et enregistrer dans le modèle d'exécution. En même temps, le moteur d'interprétation utilise l'implantation de service « Recommandation pour la sortie » afin de créer une instance qui va utiliser le service de « météo ». L'implantation de ce service est disponible au sein du modèle d'exécution depuis le moment de la création de l'application à l'exécution.

Événement 3: « Trafic » apparaît sur le réseau comme un service Web

Le point de variation « pvp5 » selon le type de variabilité « multiple » définit que le service « Trafic » peut dynamiquement joindre l'application au cours de l'exécution. Premièrement, la plate-forme d'intégration de service et d'exécution le détecte et ensuite génère automatiquement un « proxy » afin de permettre aux autres services d'invoquer les opérations fournies par ce service Web. Comme la création de l'instance de service « Trafic», le modèle d'exécution enregistre ce proxy généré, tandis que le moteur d'interprétation est notifié l'apparition de ce nouveau service disponible. Le « proxy » est alors instancier et enregistrer dans le modèle d'exécution. En même temps, le moteur d'interprétation permet au service « Recommandation pour la sortie » d'utiliser le service de « Trafic ».

3.3.3. Résultat

De la même façon que pour l'évaluation de l'application précédente, nous avons développé le scénario décrit ci-dessus pour tester la deuxième application. La machine d'exécution et l'application fonctionnent dans le même environnement physique que lors de l'évaluation précédente. Les résultats sont obtenus sous forme de courbes de la consommation du processeur et de la mémoire au cours de l'exécution.

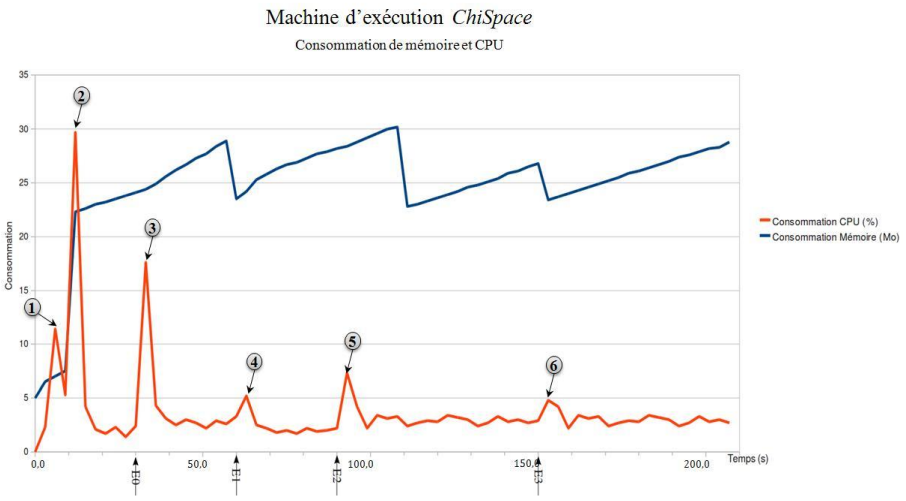


Figure 7.6. la consommation de mémoire et CPU

Le diagramme dans la figure 7.6 représente la consommation en ressources de la machine d'exécution :

- l'axe des abscisses représente le déroulement du temps d'exécution en seconde,
- l'axe des ordonnées représente la consommation en ressources de la machine d'exécution,
- la courbe bleue représente la valeur mesurée de la consommation de mémoire en Mégaoctet,
- la courbe orange représente la valeur mesurée du pourcentage d'utilisation du processeur.

Le moment du chargement de l'architecture applicative et les événements causant les changements sont pointés par les flèches numérotées, par exemple « E0 », « E1 », « E2 », etc., sur l'axe des abscisses du déroulement du temps de l'exécution. Les points numérotés sur la courbe orange représentent la valeur mesurée de l'utilisation du processeur en réaction aux changements de contexte selon le scénario :

- Le premier point représente le pic de consommation associé au chargement de la plate-forme d'exécution H-Omega et de la machine d'exécution.
- Le deuxième point représente le pic de consommation associé à l'établissement de l'état initial du scénario.
- Le troisième point représente le pic de consommation lié au chargement du modèle de l'architecture applicative et à la création de l'application. Le moteur d'interprétation est chargé d'interpréter l'architecture applicative en entrée, tout en analysant les instances et les implantations de service utiles et disponible afin de créer l'application à l'exécution.
- Le quatrième point représente le pic d'utilisation afin de réagir au premier événement– « Haut-parleur » disparu et « Écran » apparu. « Écran » étant fourni par un équipement UPnP, le moteur d'interprétation charge l'implantation de « proxy » correspondante. Ensuite, il crée une instance de ce « proxy » à partir de son implantation et l'intègre dans l'application. La courbe illustre ainsi le pourcentage d'utilisation du processeur correspondant à ce type d'adaptation. Nous considérons cette surcharge d'utilisation du processeur comme faible.
- Le cinquième point représente le pic d'utilisation au moment de l'apparition du deuxième événement : « Météo » apparu. Grâce à l'apparition de « Météo », le moteur d'interprétation prend en compte le fait que « Recommandation pour la sortie » devient utilisable dans l'architecture applicative. Cela permet de compléter la fonctionnalité de l'application. Le moteur d'interprétation crée ainsi une instance de service de cette implantation et une instance du « proxy » du service Web « Météo ». A la fin, il reconfigure l'application à l'exécution en intégrant ces deux instances de service. Puisque le moteur d'interprétation recalcule les variations au sein de l'architecture à « pvp-o3-2 » et « pvp5 », la valeur mesurée est un peu plus élevée que pour le premier événement, tout en restant très raisonnable.
- Le sixième point représente le pic d'utilisation au moment de l'apparition du troisième événement : « Trafic » apparu sur le réseau. Le moteur d'interprétation crée d'abord une instance de service du « proxy » correspondant. Puis il joint cette instance de service dans l'application à l'exécution en fonction de la variation définie à « pvp5 » de logique « multiple ». La courbe illustre une faible valeur d'utilisation du processeur correspondant à cet événement grâce aux raisons suivantes : a) le moteur d'interprétation utilise un mécanisme passif pour être notifié de la disponibilité de « Trafic »; b) l'instanciation du « proxy » de « Trafic » est une opération très peu couteuse ; c) ce changement impact seulement le point de variation « pvp5 ».

Le reste de la courbe de la consommation du processeur représente l'état stabilisé de la machine d'exécution et de l'application à l'exécution qui fonctionne continuellement et correctement. Nous pouvons observer que notre machine d'exécution n'engendre pas de surcoût constant à l'exécution. Ceci

est dû à la nature réactive de notre machine d'exécution, qui va uniquement réagir à des événements, plutôt que d'utiliser des mécanismes actifs pour s'assurer de l'intégrité de l'application.

La courbe bleue montre aussi que la valeur moyenne de la consommation du mémoire est stable au cours de l'exécution (environ 25Mo). Les décroissances soudaines représentent la libération de mémoire causée par l'exécution du ramasse miette de la « JVM ». Le constat sur l'utilisation mémoire est similaire au constat effectué pour la première application, à savoir que la machine d'exécution n'engendre pas de surcharge mémoire significative et que la consommation mémoire reste stable au cours de l'exécution.

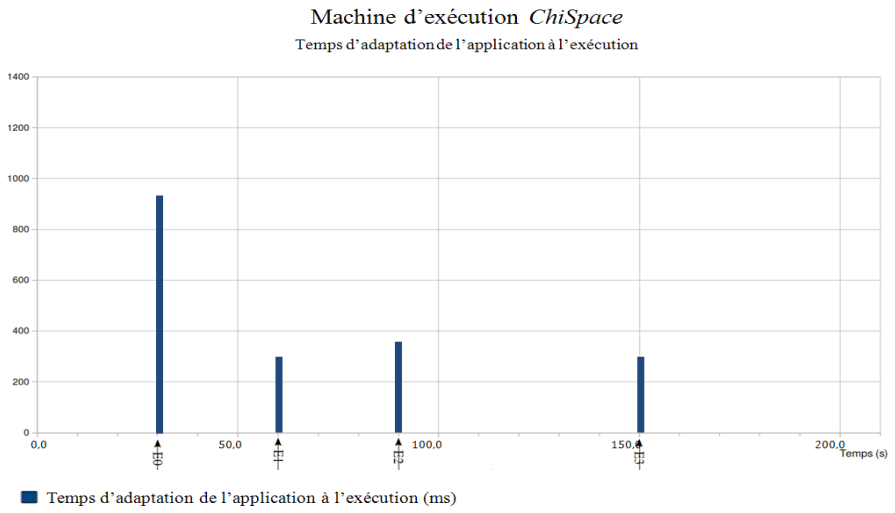


Figure 7.7. Le temps d'adaptation moyen de l'application à l'exécution

Le diagramme (dans la figure 7.7) illustre une moyenne du temps d'adaptation de l'application à l'exécution, réalisé sur 50 exécutions, correspondant aux différents événements de changements définis dans le scénario réalisé. L'axe des abscisses, dans le diagramme, correspond à la droite graduée du déroulement du temps de l'exécution en seconde. L'axe des ordonnées correspond au temps d'adaptation de l'application à l'exécution en microseconde. La création de l'application et l'adaptation aux événements de changement du contexte sont explicitement indiqués par les flèches numérotées avec « E0 », « E1 », « E2 », etc., comme pour la courbe précédente. La création de l'application à l'exécution indiquée par « E0 » prend la plus longue durée. Les autres adaptations prennent un temps inférieur à 350 ms, ce qui nous paraît être un temps de réaction plus qu'acceptable pour ce type d'application. De plus l'événement E2 est un événement engendrant une reconfiguration complexe car il met en jeu plusieurs instances de services et permet la création d'une branche complète de l'application avec un temps de réaction très similaire à la reconfiguration plus simple que sont les événements E1 et E3.

4. Conclusion

Ce chapitre a présenté les expériences pratiques que nous avons menées pour valider l'approche proposée dans cette thèse. Premièrement, nous avons étudié le domaine spécifique dans un contexte industriel afin de valider notre approche. Notre prototype a été validé dans le cadre du projet ITEA ANSO qui regroupait les grands acteurs industriels et académiques du domaine des passerelles résidentielles et de l'automatisation des maisons.

Deuxièmement, nous avons étudié deux applications pour assister des personnes à leur domicile dans des activités liées à la santé. Ces applications réalisées montrent l'intérêt de notre approche pour faciliter le développement des applications à services dans un domaine : les applications résidentielles. La conception d'application est réalisée sous la forme d'une architecture applicative en spécialisant l'architecture de référence dédiée aux applications médicales. Cette architecture de référence est définie par l'architecte du domaine lors de l'ingénierie du domaine grâce à notre outil. Nous avons effectué la spécialisation de l'architecture de référence en suivant les règles de raffinement d'architecture présentées dans le chapitre 5. Les architectures applicatives sont définies à l'aide de l'outil dédié généré après la phase d'ingénierie du domaine. Cet outil est ainsi dédié pour la conception de l'architecture applicative dans le domaine des applications médicale à la maison. Nous avons ainsi pu facilement concevoir nos deux applications. Cela nous a également permis de valider de façon expérimentale les règles de spécialisation de l'architecture. Ces règles permettent ainsi à la fois de s'assurer de la conformité de l'application par rapport à l'architecture de référence et apporte la flexibilité nécessaire pour la conception plus spécialisée d'applications.

Finalement, la création et la gestion des deux applications à services sont automatisées lors de la phase d'exécution à l'aide de la machine d'exécution. Nous avons ainsi testé ces applications dans les scénarios mettant en jeu les capacités de la machine d'exécution à créer les applications et à réagir à des événements entrainant des reconfigurations plus ou moins complexe. Cela nous a permis à la fois de vérifier de façon expérimentale le fonctionnement de notre machine d'exécution et d'évaluer les performances de ce machine d'exécution. Les performances évaluées de la machine d'exécution concernent à la fois l'impact sur les applications exécutées et le temps de réaction par rapport à des événements entrainant des reconfigurations. Les résultats sont satisfaisants. Cet outil permet ainsi de créer correctement et automatiquement les applications en assemblant les services en fonction de leur architecture applicative, tout en considérant l'état de la plate-forme d'exécution. De plus, la performance de la machine d'exécution a peu d'impact sur l'exécution des applications à services lors de l'exécution et le temps de réaction pour reconfigurer une application en réponse à un événement est suffisamment rapide compte tenu du domaine d'application. Pour finir, nous estimons que même si la machine d'exécution actuelle reste minimale et peut très largement être améliorée, les performances constatées montrent l'intérêt de l'approche globale. En effet, la machine d'exécution n'engendre pas de surcoût notoire lors de l'exécution. Dans le cas contraire, ces performances auraient rendu l'approche dénuée de sens. Ce prototype nous a ainsi permis de valider l'intérêt de notre approche.

Chapitre 8 Conclusion et Perspectives

Le but de ce dernier chapitre est de conclure cette thèse en apportant une synthèse de la contribution et en ouvrant certaines perspectives. Nous pensons en effet qu'il existe plusieurs possibilités d'extension du travail présenté ici.

1. Synthèse

1.1. Défis

Au travers des services, nous avons abordé dans cette thèse le développement d'applications pervasives. La production de telles applications demande de répondre à de multiples défis [110]. Il faut, en premier lieu, gérer l'hétérogénéité logicielle et matérielle des équipements électroniques et leur distribution. Il convient ensuite de gérer le dynamisme inhérent de ces applications de façon à les adapter à un contexte fortement évolutif. Cette thèse a montré que le développement d'applications par composition de services dynamiques et hétérogènes permettait de répondre à ces défis.

Le développement d'applications à services hétérogènes et dynamiques est ainsi le sujet principal de cette thèse. L'approche à services apporte des changements considérables dans le domaine du logiciel et peut amener des gains significatifs en termes de réduction des coûts, d'amélioration de la qualité et de réduction des temps de mise sur le marché. Les technologies à services ont dors et déjà pénétré de nombreux secteurs d'activité et répondent à certaines des attentes qu'ils suscitaient.

Cette thèse a néanmoins identifié les difficultés de développement d'applications à services dans un environnement distribué, dynamique et hétérogène sur plusieurs aspects :

- les diverses technologies existantes utilisent des mécanismes de déclaration, de recherche et de liaison très différents. Les services, eux-mêmes, sont décrits suivant des structures souvent éloignées. Des développements et des connaissances techniques très pointus sont ainsi nécessaires pour correctement associer des services utilisant des bases technologiques différentes. Un état de l'art de ces multiples technologies à services est présenté dans le chapitre 2.
- la gestion du dynamisme est difficile. Le principe de l'approche à service est de permettre la liaison retardée de service et, dans certains cas, le changement des liaisons en fonction de l'évolution du contexte. Cela demande des algorithmes de synchronisation très précis, difficiles à mettre au point et à tester. Nous nous sommes ainsi rendu compte que, dans de nombreux cas, les bénéfices de l'approche à service ne sont pas complètement obtenus, faute d'une gestion appropriée du dynamisme.
- les services sont essentiellement décrits suivant une logique syntaxique. On ne peut donc pas garantir, dans un cas général, la compatibilité de plusieurs services ou, plus simplement, la correction de leur comportement global. Cela est d'autant plus difficile lorsque des services ont des interactions complexes, non limitées à un unique appel pour obtenir une information.

1.2. Approche en trois phases outillées :

Nous apportons, dans cette thèse, une dimension « domaine » à la composition de service. La définition d'un domaine permet de restreindre les compositions possibles, aussi bien au niveau technologique qu'au niveau sémantique. C'est ainsi que nous avons trouvé une grande complémentarité entre les approches à service et les approches à base de lignes de produits. Les technologies à services apportent de façon naturelle le dynamisme, c'est-à-dire la capacité à créer des liaisons entre services de façon retardée à l'exécution. Les lignes de produits, quant à elles, définissent un cadre de réutilisation anticipée et planifiée. De façon plus précise, cette thèse défend une démarche outillée de composition de services structurée en trois phases, à savoir : la définition d'un domaine sous forme de services et

d'architectures de référence à services, la définition d'applications sous forme d'architectures à service, et l'exécution autonomique des applications en fonction de l'architecture applicative du contexte.

L'approche proposée permet de réunir les points forts des approches de lignes de produits et des approches à services. En effet, elle emprunte aux lignes de produits les notions d'ingénierie du domaine et applicatives et, de façon générale, la volonté de mettre en place une réutilisation planifiée et anticipée, de granularité élevée au sein d'un domaine. Elle a intégré également à tous les niveaux des notions de variabilité permettant de retarder des décisions de conceptions. Cette approche permet aussi de s'appuyer sur la propriété de dynamisme des approches à services de façon à retarder le plus tard possible à l'exécution des choix des services concrètes constituant l'application à l'exécution.

Le prototype *ChiSpace* a été développé pour supporter la démarche proposée dans cette thèse. Il comprend trois outils complémentaires :

1. **L'outil d'ingénierie du domaine** permet de définir le modèle du domaine sous la forme de services et d'architectures de référence à services.
2. **L'outil d'ingénierie applicative** permet de définir l'architecture applicative à base de services en conformité avec une architecture de référence précédemment définie. Les deux outils supportant les deux premières phases de la conception d'applications à services ont été testés dans divers contextes. Ils ont atteint un bon niveau de qualité.
3. **La machine d'exécution** se charge de gérer dynamiquement l'exécution de l'application suivant l'architecture applicative précédente. Pour cela, un moteur d'interprétation crée et maintien une application à services en fonction des services disponibles au moment de l'exécution et en conformité avec un modèle, celui constitué par l'architecture applicative. La machine d'exécution a été testée uniquement sur les applications du domaine résidentiel et demeure, aujourd'hui à l'état de prototype.

Nous avons validé l'approche proposée dans cette thèse dans le contexte du projet européen ITEA ANSO. Deux applications à services liées à la santé et au maintien de personnes à domicile ont été réalisées et testées dans le domaine résidentiel. Cela a démontré, en particulier, que l'« overhead » introduit par la machine d'exécution avait peu d'impact sur la performance des applications à services. Le temps de réaction pour reconfigurer une application en réponse à un événement par la machine d'exécution est suffisamment rapide compte tenu du domaine d'application. Ainsi, la machine d'exécution n'engendre pas de surcoût notoire lors de l'exécution.

2. Perspectives

Cette section décrit trois perspectives ouvertes par cette thèse sur les trois aspects ci-dessous :

- Les travaux autour de l'atelier de développement de services hétérogènes ;
- L'extension de la machine d'exécution dédiée à un domaine spécifique de façon générative ;
- L'extension de la machine d'exécution pour la gestion autonomique d'applications à services.

2.1. Environnement de développement de services

Nos travaux se sont concentrés sur le développement et l'exécution d'applications basées sur des services hétérogènes et dynamiques. La gestion précise de ces applications est mise en place de façon autonomique par la machine d'exécution. Cette gestion part de l'hypothèse que les implémentations de services utilisées sont déjà réalisées et mises à disposition sur la plate-forme d'exécution ou sur le réseau. Le développement de ces implémentations de services et leur déploiement n'a pas été envisagé dans le cadre de cette thèse.

Le développement de services suivant une technologie particulière demeure une tâche complexe pour les développeurs pour plusieurs raisons. Les développeurs doivent, en effet, avoir à la fois une bonne connaissance de la logique métier mais aussi une connaissance pointue de la technologie utilisée. Concrètement, ils doivent maîtriser le modèle d'implantation des services de façon, entre autres choses, à définir les interfaces spécifiques nécessaires et à développer le code suivant le langage et les patterns appropriés. Dans des environnements pervasifs, les développeurs peuvent être amenés à manipuler plusieurs technologies en fonction de leurs caractéristiques et des besoins applicatifs.

Une des perspectives à court terme de notre travail vise l'intégration au sein de *ChiSpace* d'un atelier de développement de services suivant différentes technologies à services. Un tel atelier devrait fournir une infrastructure de développement adaptée à chacune des technologies à utiliser, sous la forme de *Plug-In Eclipse* par exemple. Un tel atelier devrait également posséder des capacités de génération de code. Plus précisément, dès que nous avons choisi iPOJO pour modèle pivot, deux aspects seraient avantageusement générés. Tout d'abord, il serait intéressant de générer automatiquement les interfaces fonctionnelles de service en Java à partir de la spécification de service suivant la technologie d'implantation visée. Par ailleurs, il faudrait que le générateur produise les squelettes de code de type *proxy* permettant d'invoquer les interfaces des composants iPOJO. D'ailleurs, la gestion des implémentations de service concrètes et la compatibilité entre eux avec l'outil reste une question ouverte.

Un tel outil permettrait d'améliorer grandement la productivité de développement de services. Grâce au support du générateur des interfaces Java et du squelette des *proxies*, les développeurs pourraient se concentrer sur la mise en œuvre de code métier.

2.2. Extension « domaine » de la machine d'exécution

La machine d'exécution à l'état du prototype reste minimale et peut très largement être améliorée. Actuellement, la machine d'exécution permet uniquement de gérer les applications à services dans le domaine résidentiel de façon dynamique en fonction de l'architecture correspondant. Notamment, le moteur d'interprétation est chargé d'interpréter un modèle d'architecture applicative de façon à créer et maintenir des applications à services. En effet, le moteur d'interprétation est naturellement centré sur un

domaine spécifique pour plusieurs raisons. Chaque domaine peut comporter des types de donnée particuliers, des spécifications métier (par exemple le thermomètre médical) et des dépendances particulières entre eux. Les contraintes métier spécifiques également dépendent du domaine, par exemple, une contrainte des applications mobiles correspond à la limite de la taille d'affichage des équipements mobiles.

Ainsi, une des perspectives intéressantes est de la création de machine d'exécution dédiée à un domaine. En se basant sur le modèle du domaine, il est possible de créer automatiquement de façon générative un moteur d'interprétation spécifique au domaine. Effectivement, le moteur d'interprétation actuel est capable d'interpréter un modèle d'architecture spécifiant des structures communes et variables sans les notions d'un domaine métier. Le modèle du domaine peut être transformé comme le méta-modèle des architectures métier. Ce méta-modèle est considéré comme une extension des concepts métier par héritage du méta-modèle d'architecture. De ce fait, le moteur d'interprétation dédié au domaine peut être facilement étendu par la génération de code en ajoutant les concepts et les contraintes métier pour le moteur d'interprétation. Ainsi, la mise en œuvre d'infrastructure spécifique à un domaine serait grandement facilitée et fiabilisée grâce à ce processus de génération automatique.

2.3. Extension de la machine d'exécution pour la gestion autonomique d'applications à services

L'objectif de la gestion des applications à services est de les adapter aux changements d'un contexte dynamique. Nous avons considéré que la définition de la variabilité au sein de l'architecture applicative permet d'assister la gestion de l'exécution de l'application. Plus précisément, l'architecture applicative avec la définition de variabilité est utilisée pour vérifier et assurer la conformité du comportement de l'exécution de l'application correspondante.

En basant sur cette considération, une perspective intéressante est de lier plus profondément notre approche avec le domaine de l'informatique autonomique. L'informatique autonomique est de produire des applications autonomes qui s'administrent seules. 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-configuration, l'auto-optimisation, l'autoprotection ou l'autoréparation. Par exemple, a) la propriété pour maximiser la disponibilité de services de la plate-forme d'exécution ; b) la propriété pour minimiser la consommation de ressources de systèmes ; c) le temps de réponse pour s'adapter au dynamisme de contexte, etc.

Pour ce type d'applications, la machine d'exécution joue le rôle clé. Elle s'appuie pour cela sur le modèle d'architecture applicative. En effet, l'architecture applicative avec la variabilité permet de représenter l'ensemble des états acceptables de l'application à services lors de l'exécution. Cette spécification de l'ensemble des états de l'application lors de l'exécution peut être enrichie en considérant les propriétés précédentes. Plus précisément, la machine d'exécution permettra de favoriser tel(s) état(s) de l'application à services. Ces états de l'application à services doivent pouvoir mieux s'adapter aux conditions particulières du contexte d'exécution. De plus, les restrictions de telles propriétés autonomiques sur les applications dépendent fortement du domaine métier. Par exemple, les applications de supervision à distance d'un patient à la maison exigent souvent un temps de réponse immédiat. Nous pensons ainsi que les gestionnaires autonomiques sont naturellement dédiés à un domaine spécifique.

La machine d'exécution peut être largement étendue en ajoutant des politiques de gestion autonomique dans les applications à services. La machine d'exécution actuelle est réalisée en respectant des deux politiques basiques de la gestion autonomique a) maximiser la disponibilité de services et b) économiser les ressources de la plate-forme d'exécution. En effet, l'extension de la machine d'exécution en terme de gestion autonomique peut être réalisée de façon générative suivant le même principe de

l'extension domaine. Nous envisageons qu'un méta-modèle de la gestion autonomique est un élément clé pour cette extension. Ce méta-modèle permettra de définir et classifier les propriétés concernant la gestion autonomique. Une dépendance entre ce méta-modèle autonomique et le méta-modèle des architectures métier doit être établie afin d'apporter les contraintes autonomiques dans l'architecture applicative. Un générateur de code devra pouvoir générer automatiquement la machine d'exécution étendue en se basant sur le méta-modèle de la gestion autonomique dédiée au domaine spécifique. Cela permettra d'améliorer la productivité de développer l'extension de la machine d'exécution, tout en assurant la stabilité de cet outil de la phase d'exécution.

Chapitre 9 BIBLIOGAPHIE

[1] C.W. Krueger, “Software Reuse”, *ACM Computing Surveys*, Vol. 24, N°2, pp.131–183, 1992.

[2] H. Mili, A. Mili, S. Yacoub, E. Addy, *Reuse-based software engineering: techniques, organization, and controls*, Wiley-Interscience New York, NY, USA, 2001

[3] P. Naur and B. Randell, eds., *Software Engineering,Report on a Conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium.

[4] C. Szypersky, Component software - beyond object-oriented programming, Addison-Wesley, 1998.

[5] I. Crnkovic “Component-based software engineering-new challenges in software development”, in *Proceedings of the 25th Information Technology Interfaces International Conference*, pp.9-18, 2003

[6] M.P. Papazoglou “Service -Oriented Computing: Concepts, Characteristics and Directions”, in *Proceedings of the 4th International Conference on Web Information Systems Engineering*, Roma, Italy, pp.3-12, 2003.

[7] D. Schmidt, “Model-Driven Engineering”, *IEEE Computer*, Vol.39, 2006.

[8] R. France and B. Rumpe, “Model-driven Development of Complex Software: A Research Roadmap”, in *International conference on 2007 Future of Software Engineering* , Minneapolis, USA, pp.37-54, 2007.

[9] E. Gamma, R. Hem, R. Jahnson and J. Vissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Professional 1994

[10] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern Oriented Software Architecture-patterns for concurrent and networked objects, volume 2, John Wiley & Sons, 2000.

[11] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003*

[12] E.W. Dijkstra, *Notes on Structured Programming*, in Structured Programming, Dijkstra, E.W., Dahl, O.J. and Hoare, C.A.R., Ed(s). Academic Press, London, England, pp.1-82, 1972.

[13] D.L. Parnas, “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering*, vol2, no.1, pp.1-9, 1976

[14] D.M. Weiss, “Next Generation Software Product Line Engineering”, Software Product Lines, 9th International Conference, SPLC 2005, pp. 1, Rennes, France, September 26-29, 2005

[15] Frank van der Linden, Klaus Schmid, Eelco Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line engineering, Springer 2007

[16] J.D. McGregor, L.M. Northrop, S.Jarrad, K.Pohl , “Initiating Software Product Lines”, IEEE Software, Vol.19, pp.24-27, Issue 4, 2002

[17] OASIS: Reference Model for Service Oriented Architecture, October 2006. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf.

[18] Ali Arsanjani. Service-oriented Modeling and Architecture, November 2004. http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/.

[19] Marco Aiello, Ganna Frankova, and Daniela Malfatti. WhatŠs in an Agreement ? An Analysis and an Extension of WS-Agreement. *Service-Oriented Computing – ICSOC 2005*, pages 424–436, 2005.

[20] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[21] Mary Shaw and David Garlan. Software architecture: perspectives on an emerging discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[22] Sun Microsystems. Jini.org. http://www.jini.org/wiki/Main_Page.

[23] UPnP Forum. UPnP Device Architecture 1.0, April 2008. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20080424.pdf.

[24] OSGi Alliance. OSGi R4 Core Specification and Service Compendium, October 2005. http://www.osgi.org/Release4/Download.

[25] Service Centric System Engineering (SeCSE). http://www.secse-project.eu/.

[26] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10) :46–52, 2003.

[27] D. Austin, A. Babir, E. Peters, and S. Ross-Talbot. Web services choreography requirements 1.0, W3C Working Draft, August 2003.

[28] OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[29] W3C. Web Services Choreography Description Language Version 1.0, December 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/.

[30] OSOA. SCA Service Component Architectre, March 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[31] Cristina Marin. Une approche orientée domaine pour la composition de services. PhD thesis, Universtié Joseph Fourier, 2008.

[32] W3C. Web Services Description Language (WSDL) 1.1, March 2001. http://www.w3.org/TR/wsdl.

[33] OASIS. Web Services Security : SOAP Message Security 1.1 (WS-Security 2004), 2006. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[34] IBM. Web Services Atomic Transaction (WS-AtomicTransaction), November 2004. http://download.boulder.ibm.com/ibmdl/pub/software/dw/library/ws-atomictransaction200411.pdf.

[35] OASIS. Universal Description Discovery and Integration specification (UDDI) Version 3.0.2, October 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[36] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together (ACM Press)*. Addison-Wesley Professional, June 2001.

[37] R. Wuyts and S. Ducasse, “Composition languages for black-box components”, In First OOPSLA Workshop on Language Mechanisms for Programming Software Components, 2001.

[38] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[39] Felix Bachmann et al. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Software Engineering Institute, May 2000.

[40] Humberto Cervantes. Vers un modèle à composants orienté services pour supporter la disponibilité dynamique. PhD thesis, Université Joseph Fourier - Grenoble I, 2004.

[41] Sun Microsystems. Java SE Desktop Technologies. http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.

[42] Humberto Cervantes and Richard S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *ICSE’04 : Proceedings of the 26th International Conference on Software Engineering*, pages 614–623, Washington,DC, USA, 2004. IEEE Computer Society.

[43] Objectweb. JOnAS: Java Open Application Server, 2007. http://wiki.jonas.objectweb.org/xwiki/bin/view/Main/WebHome.

[44] Sun Microsystems. Glassfish: Open Source Application Server. https://glassfish.dev.java.net/.

[45] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The Eclipse 3.0 platform : adopting OSGi technology. *IBM Syst. J.*, 44(2) :289–299, 2005.

[46] Clément Escoffier and Richard S. Hall. Dynamically Adaptable Applications with iPOJO Service Components. *6th International Symposium on Software Composition(SC 2007)*, pages 113–128, March 2007.

[47] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO : an Extensible Service-Oriented Component Framework. *IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481, July 2007.

[48] ObjectWeb. The Fractal Project, 2004. http://fractal.objectweb.org/.

[49] M. Fowler. POJO: An acronym for: Plain Old Java Object, 2000. http://www.martinfowler.com/bliki/POJO.html.

[50] Object Management Group. CORBA 3.1, 2008. http://www.omg.org/spec/CORBA/3.1/.

[51] Jeff Magee, Andrew Tseng, and Jeff Kramer. Composing Distributed Objects in CORBA. In *ISADS’97 : Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems*, page 257,Washington, DC, USA, 1997. IEEE ComputerSociety.

[52] Manolis Marazakis, Dimitris Papadakis, and Christos Nikolaou. Aurora : An rchitecture for Dynamic and Adaptive Work Sessions in Open Environments. In *DEXA’98 : roceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 480–491, London, UK, 1998. Springer-Verlag.

[53] Elmar Zeeb, Andreas Bobek, Hendrik Bohn, and Frank Golatowski. Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services. In *AINAW ’07 : Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 956–963, Washington, DC, USA, 2007. IEEE Computer Society.

[54] François Jammes, Antoine Mensch, and Harm Smit. Service-Oriented Device Communications Using the Devices Profile for Web Services. In *MPAC'05 : Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–8, New York, NY, USA, 2005. ACM.

[55] Microsoft Corporation. Devices Profile for Web Services, February 2006. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.

[56] Microsoft Corporation. Web Services Dynamic Discovery (WS-Discovery), April 2005. http://schemas.xmlsoap.org/ws/2005/04/discovery/.

[57] W3C. Web Services Metadata Exchange (WS-MetadataExchange), August 2008. http://www.w3.org/Submission/2008/SUBM-WS-MetadataExchange-20080813/.

[58] W3C. Web Services Eventing (WS-Eventing), March 2005. http://www.w3.org/Submission/WS-Eventing/.

[59] W3C. Web Services Addressing (WS-Addressing), August 2004. http://www.w3.org/Submission/ws-addressing/.

[60] IBM. WebSphere Process Server, September 2005. http://www.ibm.com/developerworks/websphere/library/techarticles/0509_kulhanek/0509_kulhanek.html.

[61] BEA. BEA Aqualogic Service Bus 3.0, March 2008. http://www.bea.com/content/news_events/white_papers/BEA_AquaLogic_Service_Bus_ds.pdf.

[62] Apache Software Foundation. The Apache Tuscany Project, 2008. http://incubator.apache.org/tuscany/

[63] L.M. Northrop, "SEI's Software Product line Tenets", IEEE Software, Vol.19, pp.32-40, Issue 4, 2002

[64] http://www.softwareproductlines.com/introduction/introduction.html

[65] David M. Weiss, Chi Tau Robert Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison Wesley, 1999

[66] J. Bosch, Design & use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000

[67] http://www.sei.cmu.edu/productlines/

[68] IEEE recommended practice for architectural description of software intensive systems (IEEE-std-1471-2000), 2000.

[69] I. Jacobson, M. Griss and P. Johnson, Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley, 1997

[70] P. Clements, L. Northrop, *Software Product Lines: Practices and Patterns*, Boston, MA, Addison-Wesley, 2002.

[71] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Nowak, and A.S. Peterson, "Feature oriented domain analysis (FODA) feasibility study", *Technical Report (CMU/SEI-90-TR-21)*, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, 1990.

[72] P. Clements, "On the Importance of Product Line Scope", in Proceedings of the 4th International Workshop, Software Product-Family Engineering, Bilbao, Spain, pp.70-78, 2001

[73] K. Schmid, “Scoping software product lines – an analysis of an emerging technology.”, *in Proceedings of the first conference on Software product lines,* Denver, Colorado, United States, pp. 513 – 532,2000.

[74] J.O. Coplien, D.M. Hoffman and D.M. Weiss, “Commonality and variability in software engineering”, *IEEE Software,* Vol.15, No.6, pp. 37-45, 1998.

[75] J. van Gurp, J. Bosch and M. Svahnberg, “On the Notion of Variability in Software Product Line”, in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, The Netherlands, pp.45-54, 2001.

[76] B. Langlois, C.E. Jitia and E. Jouenne, “DSL Classification”, in *Proceedings the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM’07)*, Monréal, Canada, pp. 28-38, 2007.

[77] K. Pohl, G. B¨ockle and F. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.

[78] C. W. Krueger, “Variation Management for Software Product Lines”, in *Proceedings of the 2nd Software Product Line Conference* (SPLC2), San Diego, CA, USA, pp.37-48, 2002

[79] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice-Hall, 1996

[80] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley, Chichester, UK, 1996

[81] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice.* Second Edition, Addison-Wesley, 2003.

[82] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley Professional, 1998

[83] N. Medvidovic and R.N. Taylor, *“A Classification and Comparison Framework for Software Architecture Description Languages”*, Transaction on Software Engineering, Jan. 2000, Vol. 26, n°1: pp.70-93.

[84] M. Shaw, R. Deline, D. Klein, T. Ross, D. Young, G. Zelesnik, "Abstraction for Software Architecture and Tools to Support Them", *IEEE Transaction on Software Engeneering*, vol.21, n°4, pp. 314-335, 1995

[85] M. Moriconi, X. l. Qian, R. A. Riemenschneider and L. Gong: "Secure Software Architecture", in *procceedings of the IEEE Symposium on Security and Privacy*, pp. 84-93, CA, 1997.

[86] D. Garlan: "An Introduction to the Aesop System", Vesrsion of 11 July 1995.

[87] S. Vestal: "MetaH User's Manual" version 1.27, Honeywell Technologie Center.

[88] R.N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robins: "A Component- and Message-Based Architectural Style for GUI Software" *In Proceedings of the 17th International Conference on Software Engineering (ICSE17)*, pp. 295-304, Seattle WA, 1995.

[89] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, “Specifying Distributed Software Architectures”, in *Proc*eedings *5th European Software Engineering Conf. (ESEC 95)*, Sitges, Spain, vol. 989, Springer-Verlag, Berlin, pp. 137–153,1995.

[90] D.C. Luckham, J.L. Kenney, L.M. Augustin, J.Vera, D.Bryan, and W. Mann, “Specification and analysis of system architecture using rapide”, *IEEE Transactions on Software Engineering 21*, no. 4, 336–355,1995

[91] "Thje Standford Rapide Project", Home Page: http://pavg.stanford.edu/rapide/

[92] D. C. Lukham: "Rapide, A langage Toolset for Simulation of Distributed System By Partial Ordering of Events" In Proceedings of DIMACS Workshop on Partiel Order Methods in Verification (POMIV) page 329-358, July 25-26, 1996.

[93] R. Allen, D. Garlan, and R. Douence, Specifying Dynamism in Software Architectures, in *proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, 1997.

[94] S. D. Brookes, C. A. R. Hoare and A. W. Roscoe, " A Theory of Communicating Sequential Processes", Journal of ACM, Vol. 31, No. 3, pp. 560-599 , 1984

[95] J.S. Kim and D. Garlan, "Analyzing Architectural Styles with alloy", in Proceedings of the Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA 2006), Portland, ME, pp. 70-80, 2006

[96] http://www.sei.cmu.edu/architecture/start/glossary/

[97] http://www.j2ee.me/javaee/

[98] J. Andersson, J. Bosch, "Development and use of dynamic product-line architectures", IEEE Software, Vol.152, Issue 1, pp.15-28, 2005

[99] K. Kang, S. Kim, J. Lee, E. Shin and M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", Annals of Software Engineering, Vol. 5, pp.143-168, 1998.

[100] M. Griss, J. Favaro and M. d'Alessandro, "Integrating feature modeling with the RSEB", in Proceedings of the 5th International Conference on Software Reuse, Los Alamitos, CA, USA, pp.76-85, 1998.

[101] A. Harhurin and J. Hartmann, "Service-oriented Commonality Analysis Across Existing Systems", in Proceeding de 12th international Software Product Lines Conference, pp. 225-264, 2008.

[102] C. W. Krueger, "Easing the transition to software mass customization", in Proceedings of the 4th International Workshop on Software Product-Family Engineering, 2002.

[103] J. Bosch. G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl, "Variability issues in Software Product Lines", in Proceedings of the 4th International Workshop on Product Family Engineering, pp. 303-338, Springer Verlag, Germany, 2002.

[104] Jian-qi Yu, Philippe Lalanda and Stéphanie Chollet. "Development tool for service-oriented applications in smart homes", *In Proceedings of the IEEE International Conference on Services Computing (SCC'08)*, Honolulu, Hawaii, USA, 2008.

[105] Femi G. Olumofin, *A Holistic Method for Assesing Software Product Line Architectures*, VDM Verlag Saarbrücken, 2007

[106] Event Admin handler: http://felix.apache.org/site/event-admin-handlers.html

[107] K. Pohl and E. Sikora, *Software Product Line Engineering, Foundations, Principles, and Techniques*, Springer Berlin Heidelberg, pp. 39-52, 2005

[108] J. Bourcier, *Auto-Home, une plate-forme pour la gestion autonomique d'applications pervasives*, Thèse dans l'Université Joseph Fourier (Grenoble 1)

[109] Gordon Blair, Nelly Bencomo and Robert B. France, “Models@Run.Time”, *IEEE, Computer,* pp. 22-27, 2009

[110] Damien Cassou, Benjamin Bertran, Nicolas Loriant and Charles Consel, "A Generative Programming Approach to Developing Pervasive Computing Systems", In *GPCE '09: Proceedings of the 8th international conference on Generative programming and component engineering*, pp. 137-146, 2009.

[111] Y. Huang, JISGA,”A JINI-Based Service Oriented Grid Architecture”, *Int. High Perform. Comput. Appl.,* 17(3): 317-327,2003

[112] http://cxf.apache.org/

Résumé

Le développement d'applications par composition de services dynamiques et hétérogènes, c'est-à-dire implantés suivant des technologies différentes, est le sujet principal de cette thèse. Nous pensons, en effet, que l'approche orientée service apporte des changements considérables dans le domaine du logiciel et peut amener des gains significatifs en termes de réduction des coûts, d'amélioration de la qualité et de compression des temps de mise sur le marché. Les technologies à services ont dors et déjà pénétré de nombreux secteurs d'activité et répondent à certaines des attentes qu'ils suscitaient.

Le développement d'applications par composition de services hétérogènes demeure néanmoins très complexe pour plusieurs raisons. Tout d'abord, les diverses technologies existantes utilisent des mécanismes de déclaration, de recherche et de liaison très différents. Les services, eux-mêmes, sont décrits suivant des structures souvent éloignées. Des développements et des connaissances techniques très pointus sont ainsi nécessaires pour correctement associer des services utilisant des bases technologiques différentes. D'autre part, la gestion du dynamisme est complexe. Le principe de l'approche à service est de permettre la liaison retardée de service et, dans certains cas, le changement des liaisons en fonction de l'évolution du contexte. Cela demande des algorithmes de synchronisation très précis, difficiles à mettre au point et à tester. Nous nous sommes ainsi rendu compte que, dans de nombreux cas, les bénéfices de l'approche à service ne sont pas complètement obtenus, faute d'une gestion appropriée du dynamisme. Enfin, les services sont essentiellement décrits suivant une logique syntaxique. On ne peut donc pas garantir, dans un cas général, la compatibilité de plusieurs services ou, plus simplement, la correction de leur comportement global. Cela est d'autant plus difficile lorsque des services ont des interactions complexes, non limitées à un unique appel pour obtenir une information.

Nous apportons, dans cette thèse, une dimension « domaine » à la composition de service. La définition d'un domaine permet de restreindre les compositions possibles, aussi bien au niveau technologique qu'au niveau sémantique. C'est ainsi que nous avons trouvé une grande complémentarité entre les approches à service et les approches à base de lignes de produits. Les technologies à service apportent de façon naturelle le dynamisme, c'est-à-dire la capacité à créer des liaisons entre services de façon retardée à l'exécution. Les lignes de produits, quant à elles, définissent un cadre de réutilisation anticipée et planifiée. De façon plus précise, cette thèse défend une démarche outillée de composition de services structurée en trois phases, à savoir : la définition d'un domaine sous forme de services et d'architectures de référence à services, la définition d'applications sous forme d'architectures à service, et l'exécution autonomique des applications en fonction de l'architecture applicative du contexte.

Cette thèse est validée au sein d'un projet collaboratif dans le domaine de la santé dans l'habitat.

Mots-Clés : approche à services, Ingénierie de lignes de produits logiciels, application à services, OSGi, iPOJO.

Summary

Application development by composition of dynamic and heterogeneous services, that is to say, implemented according to different technologies, is the main subject of this thesis. We believe, actually, that service-oriented approach brings considerable changes in software computing and can bring significant benefits in terms of cost reduction, quality improvement and compression of time-to-market. The service-based technologies have already penetrated in many industries sections and answered certain expectations they aroused.

Application development by composing heterogeneous services is still very complex for several reasons. Firstly, the various existing technologies employ very different mechanisms for declaration, research and liaison. The services themselves are described following structures often different. Therefore, development and technical knowledge are necessary to correctly combine services using different technological bases. On the other hand, dynamism management is complex. The principle of the service-oriented approach is to allow late service binding and, in some cases, the change of bindings according to context evolution. This requires very precise synchronization algorithms and is difficult to develop and test. We are well aware that in many cases, the benefits of service-oriented approach are not fully achieved, lacking of appropriate dynamism management. Finally, services are essentially described using a logical syntax. Therefore, we cannot guarantee, in a general case, the compatibility of several services or, more simply, the correctness of their global behavior. This is even more difficult when services have complex interactions, not restricted to a single call to obtain information. In this thesis, we bring a domain-specific dimension into service composition. The domain definition allows restricting the possible compositions of services, both at technical and semantic level. Thus we have found great complementarities between software product lines approaches and service-oriented approaches. Dynamism is a natural characteristic of service-based technologies, that is to say, the ability to bind services as late as possible utile runtime. Software product-lines, in turn, define approaches for planned reuse. Specifically, this thesis provides a three-phase approach for development of service composition with tool support, namely: the definition of a domain in the form of services and reference architectures, the definition of application in the form of service-based architectures and the execution of autonomic applications following application architecture.

This thesis is validated in a collaborative project in the home healthcare domain.

Keywords : SOC, dynamic software product lines, application-based services, OSGi, iPOJO.