HAL
open science

Modèles, outils et plate-forme d'exécution pour les applications à services dynamiques

Diana Moreno

To cite this version:

Diana Moreno. Modèles, outils et plate-forme d'exécution pour les applications à services dynamiques. Génie logiciel [cs.SE]. Université de Grenoble, 2013. Français. NNT : . tel-00875053

HAL Id: tel-00875053

https://theses.hal.science/tel-00875053v1

Submitted on 21 Oct 2013

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Diana Guadalupe MORENO GARCIA

Thèse dirigée par **Jacky ESTUBLIER**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Modèles, outils et plate-forme d'exécution pour les applications à services dynamiques

Thèse soutenue publiquement le **22 février 2013**,
devant le jury composé de :

Mme. Laurence DUCHIEN
Professeur à l'Université de Lille 1, Rapporteur

M. Michele LANZA
Professeur à l'Université de Lugano, Rapporteur

Mme. Mireille BLAY-FORNARINO
Professeur à l'Université de Nice, Examinateur

M. Jean Pierre GIRAUDIN
Professeur à l'UPMF, Grenoble, Président

M. Jacky ESTUBLIER
Directeur de recherche au CNRS, Directeur de thèse

M. Germán Eduardo VEGA BAEZ
Ingénieur de recherche au CNRS, Co-encadrant de thèse

A Elva, Juan, Diego et Omar

« *Don't cry because it's over, smile because it happened.* »

T. Seuss Geisel

REMERCIEMENTS

Je tiens à remercier tous ceux qui ont contribué à l'aboutissement de ce travail. Je pense tout d'abord à ceux qui m'ont fait l'honneur d'être les membres de mon jury. Je remercie Laurence Duchien et Michele Lanza pour avoir évalué et reconnu mon travail, ainsi que Mireille Blay-Fornarino et Jean-Pierre Giraudin pour l'avoir examiné et jugé.

Je souhaite remercier Jacky Estublier pour m'avoir accueillie au sein de son équipe, pour avoir dirigé ce travail, pour son aide et ses encouragements. Je remercie également Germán Vega pour sa disponibilité, son écoute, ses commentaires et ses précieux conseils.

Je remercie vivement tous les membres de l'équipe Adèle pour leur soutien, les discussions, les conseils, la convivialité et la bonne ambiance qu'ils font régner. Je remercie également mes collègues de l'Université des Andes pour leurs commentaires et leurs conseils sur mon travail, ainsi que pour leur chaleureuse hospitalité et leur convivialité. Je souhaite exprimer ma sincère gratitude à Genoveva Vargas-Solar pour m'avoir offert l'opportunité de venir en France et pour m'avoir fait partager ses connaissances et son expérience.

Je remercie l'ensemble de mes amis en France pour leur aide et leur soutien ainsi que pour tous les bons moments que nous avons passés ensemble. Je tiens à ne pas oublier mes amis au Mexique pour m'avoir encouragée tout au long de ces années malgré la distance. Un grand merci donc à Carlos Hernán Prada, Walter Rudametkin, Marcia Marrón, Gabriel Pedraza, Lulu Martínez, Noé García, Eli González, Christiane Tron, Laura Tamayo, Jonathan Bardin, Mehdi Damou, Kiev Gama, Idrissa Dieng, Clément Escoffier, Luis et Marie Martínez, Carlos Jaime Barrios, Carlos Manuel López, Jennifer Zarate, Paola Durán, Andrés Quiroga, Luz María Priego, José Luis Aguirre, Amal Haddad, Alexandra Pomares, Sattisvar Tandabany, Hanh Tan, Giang Vu, Germán Aguilar, Javier Espinosa, Alex Téllez, Rafa García, Jennifer Rojas, Mauricio Romero, Juan José Cruz, Laura Téllez et à tous ceux que j'aurais pu oublier.

Je tiens à remercier profondément mon frère Juan et mes neveux Diego et Omar pour leur immense tendresse et leur soutien inconditionnel, ainsi que les familles García-Cano García, García Huerta, García Motolina, Castillo Caravantes, Escalante Caravantes, Téllez Rivas, Clark, Arbin et Bourret pour leurs inestimables encouragements.

Enfin, un très grand merci à Pierre pour avoir supporté mes nombreuses angoisses, pour tout son amour, son soutien, sa bonne humeur et sa folie contagieuse, ainsi que pour tous les tendres moments que nous avons passés ensemble.

RESUME

L'essor de l'Internet et l'évolution des dispositifs communicants ont permis l'intégration du monde informatique et du monde réel, ouvrant ainsi la voie à de nouveaux types d'applications, tels que les applications ubiquitaires et pervasives. Ces applications doivent s'exécuter dans des contextes hétérogènes, distribués et ouverts qui sont en constante évolution. Dans de tels contextes, la disponibilité des services et des dispositifs, les préférences et la localisation des utilisateurs peuvent varier à tout moment pendant l'exécution des applications.

La variabilité des contextes d'exécution fait que l'exécution d'une application dépend, par exemple, des services disponibles ou des dispositifs accessibles à l'exécution. En conséquence, l'architecture d'une telle application ne peut pas être connue statiquement à la conception, au développement ou au déploiement, ce qui impose de redéfinir ce qu'est une application dynamique : comment la concevoir, la développer, l'exécuter et la gérer à l'exécution.

Dans cette thèse, nous proposons une approche dirigée par les modèles pour la conception, le développement et l'exécution d'applications dynamiques. Pour cela, nous avons défini un modèle de composants à services permettant d'introduire des propriétés de dynamisme au sein d'un modèle de composants. Ce modèle permet de définir une application en intention, par un ensemble de propriétés, de contraintes et de préférences de composition. Une application est ainsi spécifiée de façon abstraite ce qui permet de contrôler la composition graduelle de l'application lors de son développement et de son exécution.

Notre approche vise à effacer la frontière entre les activités effectuées avant et pendant l'exécution des applications. Pour ce faire, le même modèle et les mêmes mécanismes de composition sont utilisés de la conception jusqu'à l'exécution des applications. A l'exécution, le processus de composition considère, en plus, les services disponibles dans la plate-forme d'exécution permettant la composition opportuniste des applications ; ainsi que la variabilité du contexte d'exécution permettant l'adaptation dynamique des compositions.

Nous avons mis en œuvre notre approche à travers le prototype nommé COMPASS, qui s'appuie sur les plates-formes CADSE pour la réalisation d'environnements logiciels de conception et de développement, et APAM pour la réalisation d'un environnement d'exécution d'applications à services dynamiques.

Mots-clés : applications dynamiques, architectures logicielles, approche à services, composition de services, environnements logiciels, ingénierie dirigée par les modèles.

ABSTRACT

The growth of the Internet and the evolution of communicating devices have allow the integration of the computer world and the real world, paving the way for developing new types of applications such as pervasive and ubiquitous ones. These applications must run in heterogeneous, distributed and open environments that evolve constantly. In such environments, the availability of services and devices, the preferences and location of users may change at any time during the execution of applications.

The variability of the execution context makes the execution of an application dependent on the available services and devices. Building applications capable of evolving dynamically to their execution context is a challenging task. In fact, the architecture of such an application cannot be fully known nor statically specified at design, development or deployment times. It is then needed to redefine the concept of dynamic application in order to cover the design, development, execution and management phases, and to enable thus the dynamic construction and evolution of applications.

In this dissertation, we propose a model-driven approach for the design, development and execution of dynamic applications. We defined a component service model that considers dynamic properties within a component model. This model allows defining an application by its intention (its goal) through a set of composition properties, constraints and preferences. An application is thus specified in an abstract way, which allows controlling its gradual composition during development and execution times.

Our approach aims to blur the boundary between development-time and runtime. Thus, the same model and the same composition mechanisms are used from design to runtime. At runtime, the composition process considers also the services available in the execution platform in order to compose applications opportunistically; and the variability of the execution context in order to adapt compositions dynamically.

We implemented our approach through a prototype named COMPASS, which relies on the CADSE platform for building software design and development environments, and on the APAM platform for building an execution environment for dynamic service-based applications.

Keywords: dynamic applications, software architectures, service-based approach, service composition, software environments, model-driven software engineering.

TABLE DE MATIERES

CHAPITRE 1. INTRODUCTION **1**
1 Contexte et problématique 1
2 Objectifs 3
3 Organisation du document 4

PREMIERE PARTIE : ETAT DE L'ART

CHAPITRE 2. CONTEXTE **7**
1 Composants et services 8
1.1 Approche à composants 8
1.1.1 Concepts de base 8
1.1.2 Technologies existantes 10
1.1.3 Avantages et limitations de l'approche à composants 13
1.2 Approche à services 14
1.2.1 Concepts de base 14
1.2.2 Service-Oriented Computing : SOC 15
1.2.3 Service-Oriented Architecture : SOA 16
1.2.4 Technologies existantes 16
1.2.5 Avantages et limitations de l'approche à services 19
1.3 Approche à composants à services 20
1.3.1 Principes 20
1.3.2 Technologies existantes 21
1.3.3 Avantages et limitations de l'approche à composants à services 25
1.4 Synthèse 25
2 Architectures logicielles 26
2.1 Concepts de base 26
2.2 Architectures à composants, à services, et à composants à services 27
2.3 Vues architecturales 28
2.4 Besoins et défis 30
2.5 Langages de description d'architecture existants 30
2.6 Synthèse 32

3 Ingénierie Dirigée par les Modèles 32
3.1 Concepts de base 33
3.2 Méta-modélisation 34
3.3 Séparation de préoccupations 35
3.4 Composition de modèles 36
3.5 Modèles à l'exécution : models@run.time 36
3.6 Synthèse 40

CHAPITRE 3. DYNAMISME ET ADAPTATION 41

1 Besoins de dynamisme et d'adaptation 42
2 Contexte d'exécution 43
3 Dynamisme et adaptation 44
3.1 Disponibilité dynamique 44
3.2 Adaptation dynamique 45
3.3 Logique d'adaptation 47
3.3.1 Reconfiguration dynamique 48
3.3.2 Emplacement 50
4 Approches existantes 51
4.1 Approche à services dynamique 51
4.1.1 OSGi 52
4.1.2 Les services Web 52
4.2 Architectures logicielles dynamiques 53
4.2.1 Darwin 53
4.2.2 Dynamic Wright 54
4.3 Modèles à composants dynamiques 55
4.3.1 ArchJava 55
4.3.2 SOFA 2.0 55
4.3.3 Fractal 56
4.4 Systèmes adaptatifs 57
4.4.1 K-Component 57
4.4.2 Rainbow 57
4.4.3 Jade 58
4.4.4 Adapt-Medium 59
4.5 Comparaison des approches 59
5 Synthèse 61

CHAPITRE 4. COMPOSITION DE SERVICES 63

1 Processus de composition 64
2 Approches de composition 65
2.1 Composition par procédés 65
2.2 Composition structurelle 67
2.2.1 SCA 68
2.2.2 iPOJO 70
3 Synthèse 73

DEUXIEME PARTIE : CONTRIBUTIONS

CHAPITRE 5. PROPOSITION **77**
1 Problématique 78
2 Objectifs 79
3 Notre approche 79
3.1 Phase de conception 82
3.2 Phase de développement 84
3.3 Phase d'exécution 88
4 Synthèse 90

CHAPITRE 6. MODELE DE COMPOSANTS A SERVICES **91**
1 Scénario d'application 92
2 Principes et mécanismes de base 93
2.1 Mécanismes de classification et de propagation 93
2.2 Groupes d'équivalence 97
2.2.1 Type de groupe 98
2.2.2 Résolution 99
2.3 Groupes de services 100
3 Métamodèle de composants à services 105
3.1 Composants à services primitifs 105
3.1.1 Spécification 105
3.1.2 Implémentation 107
3.1.3 Instance 110
3.1.4 Propriétés prédéfinies 111
3.2 Composants à services composites 111
3.2.1 Spécification composite 111
3.2.2 Implémentation composite 114
3.2.3 Instance composite 117
3.2.4 Propriétés prédéfinies 118
4 Construction d'applications 120
4.1 Conception 120
4.1.1 Langage de composition 120
4.1.2 Langage de sélection 121
4.2 Développement et exécution 122
4.2.1 Composition 122
4.2.2 Evolution dynamique 124
4.3 Visions top-down et bottom-up 124
5 Synthèse 124

CHAPITRE 7. REALISATION **127**
1 Conception et développement 128
1.1 CADSEs : Environnements dédiés 128
1.2 COMPASS-CADSE 131
1.3 Exemple : Mise en œuvre d'une application dans COMPASS-CADSE 133
2 Exécution 141
2.1 APAM 141
2.1.1 Concepts 141
2.1.2 Mécanismes 142
2.2 COMPASS-RT 145
2.2.1 Démarrage d'une application 145
2.2.2 Gestion d'une application 148
2.2.3 Evolution d'une application 149
3 Synthèse 150

CHAPITRE 8. CONCLUSION ET PERSPECTIVES **151**
1 Synthèse 151
2 Perspectives 153
2.1 Outils d'exécution intégrés aux environnements de développement 153
2.2 Calcul de « la meilleure composition possible » 154
2.3 Intégration d'autres préoccupations non-fonctionnelles 154
2.4 Montée en abstraction 154
2.5 Evolution proactive 155
2.6 Génie logiciel@run.time 155

ANNEXE A. LANGAGE DE COMPOSITION ET DE CONTRAINTES **157**
1 Langage de composition 157
2 Langage de séléction 158

ANNEXE B. PROPRIETES APAM **159**

REFERENCES **161**

TABLE DE FIGURES

Figure 1. Cycle de vie d'un composant et formes de composants 9
Figure 2. Une instance de composant et son conteneur 10
Figure 3. Un type de composant EJB 11
Figure 4. Un type de composant Fractal 12
Figure 5. Principe d'interaction de l'approche à services 15
Figure 6. Mécanismes d'un environnement d'intégration et d'exécution de services 16
Figure 7. Un *bundle* et son cycle de vie 17
Figure 8. Un composant SCA 22
Figure 9. Un composant iPOJO 22
Figure 10. Métamodèle de services SAM [28] 24
Figure 11. Architectures à différents niveaux d'abstraction pour des vues différentes 29
Figure 12. Architectures d'une application à différents niveaux 29
Figure 13. Modèle descriptif 34
Figure 14. Modèle prescriptif 34
Figure 15. Relations entre les concepts de l'IDM 34
Figure 16. Méta-métamodèle de modèles à l'exécution [57] 37
Figure 17. Types de modèles à l'exécution [58] 38
Figure 18. Classification des applications dynamiques 46
Figure 19. Reconfiguration issue de la disponibilité dynamique d'un composant 47
Figure 20. Reconfiguration dynamique d'une application 48
Figure 21. Remplacement dynamique d'un composant en utilisant un médiateur 49
Figure 22. Approche à services dynamique 51
Figure 23. Exemple dans Darwin 54
Figure 24. Exemple dans Dynamic Wright 54
Figure 25. Orchestration de services 65
Figure 26. Chorégraphie de services 65
Figure 27. Composition structurelle de services 67
Figure 28. Un composite SCA 69
Figure 29. Description d'un composite iPOJO 71
Figure 30. Instance d'un composite iPOJO 71
Figure 31. Exemple de résolution de dépendances dans iPOJO 72

Figure 32. Vision globale de notre approche 81
Figure 33. Spécification de service 82
Figure 34. Spécification composite de service 83
Figure 35. Implémentation de service 84
Figure 36. Instance de service 85
Figure 37. Implémentation composite et Instance composite de services 86
Figure 38. Modèle d'application 87
Figure 39. Modèle d'état 89
Figure 40. Exemple d'application de gestion multimédia pour des passerelles résidentielles 92
Figure 41. Matérialisation 94
Figure 42. Instanciation profonde 95
Figure 43. *Powertype* 95
Figure 44. Sous-typage du type partitionné d'un *powertype* 96
Figure 45. Mécanisme de groupes 97
Figure 46. Types de groupes et groupes 98
Figure 47. Types de groupes de services 101
Figure 48. Le groupe de services *MediaManager* 102
Figure 49. Exemples de résolution du groupe *MediaManager* 103
Figure 50. Types métier et types technologiques de groupes de services 104
Figure 51. Spécification de composant à services 105
Figure 52. Spécification de composant à services – Exemple *MediaManager* 107
Figure 53. Implémentation de composant à services 107
Figure 54. Implémentation de composant à services – Exemple *ADELE-MediaManager* 109
Figure 55. Instance de composant à services 110
Figure 56. Spécification composite 112
Figure 57. Spécification composite – Exemple *HomeMediaCenter* 113
Figure 58. Implémentation composite 114
Figure 59. Implémentation composite – Exemple *ADELE-MediaCenter* 116
Figure 60. Instance composite 117
Figure 61. Métamodèle de composant à services 119
Figure 62. Le système COMPASS 127
Figure 63. Métamodèle et modèle de données de CADSE 129
Figure 64. Métamodèle et modèle de correspondances de CADSE 129
Figure 65. Vues des modèles d'un CADSE 130
Figure 66. Approche générative de CADSE 130
Figure 67. Architecture de COMPASS-CADSE 132
Figure 68. Création de la spécification *MediaManager* 136
Figure 69. Création de l'implémentation *ADELE-MediaManager* 138
Figure 70. Spécification composite *HomeMediaCenter* 139
Figure 71. Implémentation composite ADELE-*MediaCenter* 140
Figure 72. Architecture du système APAM 141
Figure 73. Résultats des expérimentations APAM vs iPOJO 144

TABLE DE TABLEAUX

Tableau 1. Concepts du modèle à composants SCA 21
Tableau 2. Concepts du modèle iPOJO 23
Tableau 3. Concepts du modèle SAM 24
Tableau 4. Avantages et limitations des approches actuelles pour la création d'applications 25
Tableau 5. Positionnement de langages ADL par rapport aux critères 31
Tableau 6. Comparaison des approches 60
Tableau 7. Modes de résolution pour la composition d'une application 88
Tableau 8. Approches de classification et de propagation d'attributs 96
Tableau 9. Propriétés prédéfinies des composants à services primitifs 111
Tableau 10. Propriétés contextuelles prédéfinies des composants à services composites 118
Tableau 11. Propriétés contextuelles prédéfinies dans APAM 159

CHAPITRE 1
INTRODUCTION

1 CONTEXTE ET PROBLEMATIQUE

Dans les dernières années, l'informatique a considérablement évolué. Les systèmes logiciels sont devenus indispensables dans de nombreux domaines d'application, et leur construction est de plus en plus complexe à cause de besoins de plus en plus nombreux. En particulier, l'avènement de l'Internet et l'évolution des appareils communicants ont permis l'intégration du monde informatique et du monde réel, ouvrant ainsi la voie à de nouveaux types d'applications tels que les applications ubiquitaires et pervasives [1] [2]. De telles applications se caractérisent par l'utilisation de composants hétérogènes et distribués fournis par des tiers, et par des contextes d'exécution de plus en plus ouverts, variables et non-déterministes. Dans de tels contextes d'exécution, la disponibilité dynamique des dispositifs, la connectivité au réseau, la localisation et les préférences des utilisateurs peuvent changer de manière imprévisible pendant l'exécution des applications.

La variabilité des contextes d'exécution rendent encore plus complexe la construction des applications. En effet, il est difficile de connaître au moment de la conception et du développement d'une application quels sont les composants et les dispositifs qui seront disponibles à l'exécution. Même lorsque cela est possible, les variations du contexte d'exécution nécessitent, en général, d'adapter l'application afin de lui permettre de continuer à fonctionner correctement, ou de tirer le meilleur parti possible des composants disponibles en cours d'exécution.

Définir pour une telle application la liste exhaustive de tous ses composants avant son exécution est donc contraignant, inadéquat, voire même impossible : l'architecture d'une application ne peut plus être complétement figée avant son exécution. Une application devrait donc être définie de manière suffisamment flexible afin de pouvoir s'adapter dynamiquement au contexte d'exécution et aux variations de celui-ci, mais aussi de manière suffisamment précise afin de pouvoir contrôler l'exécution correcte de l'application. Une telle définition devrait spécifier un cadre abstrait qui définit l'ensemble, potentiellement infini, des exécutions valides d'une application.

La construction d'applications modernes présente donc des défis de conception, de développement, d'exécution et de maintenance, imposant de redéfinir comment une application est conçue, développée, exécutée et gérée à l'exécution.

De nombreux travaux de recherche s'intéressent à la construction d'applications logicielles afin de proposer des nouvelles approches qui répondent aux besoins et aux défis des applications. Les travaux dans le domaine du génie logiciel ont abouti à l'apparition de nouveaux paradigmes pour la construction d'applications, comme l'ingénierie dirigée par les modèles (IDM) qui considère le développement d'applications logicielles comme un processus de spécification, raffinement et intégration de modèles. Une vision plus ambitieuse de l'IDM est apparue récemment : les modèles

d'exécution (*models@run.time*) [3]. Dans cette vision, les modèles sont utilisés pour le développement des logiciels, mais aussi pendant la phase d'exécution afin de surveiller et gérer leur exécution.

D'autres paradigmes du génie logiciel conçoivent la construction d'applications comme une activité d'assemblage d'entités logicielles disponibles. L'approche à composants (*Component Based Software Engineering - CBSE*) [4], apparue autour du milieu des années 90, a été motivée d'un côté par des arguments économiques comme la réduction du temps et des coûts de développement des applications, et d'un autre côté pour éliminer les limitations de l'approche à objets [5]. Elle promeut la construction d'applications par réutilisation et par assemblage (composition) de composants. De façon simpliste, un composant peut être décrit comme une brique logicielle composable avec d'autres composants et réutilisable dans la construction de différentes applications. Pour permettre son utilisation, un composant précise, par des interfaces, les fonctionnalités qu'il fournit mais aussi ses dépendances fonctionnelles. L'approche à composants propose une séparation claire entre les interfaces fournies et le code réalisant l'implémentation. **L'approche à composants est centrée implémentation, abordant principalement le développement et la composition structurelle de composants**.

Une application est ainsi composée structurellement à travers les interfaces des composants : les implémentations de composants constituant l'application sont liées entre elles d'une interface requise à une interface fournie. En conséquence, différentes compositions d'une application (nommées configurations) peuvent être possibles en raison des différentes implémentations de composants disponibles au moment de la composition.

L'approche à composants a connu une grande popularité et des nombreux modèles de composants ont été proposés tant dans le monde académique qu'industriel [6]. Cependant, ces modèles ne traitent pas en général les besoins de flexibilité et de dynamisme des applications modernes : les applications sont généralement composées au développement ou au déploiement ; une fois une application composée, les changements dynamiques dans son architecture ne sont pas possibles ou sont difficiles à réaliser, soit parce que les modèles ne supportent pas la description d'architectures dynamiques, ou parce que les technologies sous-jacentes d'exécution ne supportent pas la réalisation de changements dynamiques. Les architectures des applications sont donc relativement statiques.

L'approche à services (*Service Oriented Computing - SOC*) [7] est apparue plus récemment pour répondre aux besoins de flexibilité, de dynamisme, d'hétérogénéité et de distribution des applications. Cette approche propose de construire des applications à partir d'éléments logiciels, nommés services, qui peuvent être fournis par des tiers et qui peuvent évoluer de façon dynamique. L'approche à services, comme l'approche à composants, propose une séparation claire entre la description des fonctionnalités fournies et l'implémentation d'un service. A partir d'une description, un client peut rechercher, découvrir, sélectionner et invoquer un service. L'approche à services met l'accent sur le fait que ce cadre d'interaction – recherche, découverte, sélection, invocation – peut être effectué à l'exécution afin de pouvoir utiliser les services disponibles à un moment précis. **L'approche à services est centrée instance, abordant principalement l'exécution de services**.

L'approche à services fournit ainsi deux propriétés importantes : le faible couplage entre clients et fournisseurs de services, et la liaison tardive. Grâce à ces propriétés, l'approche à services permet la construction d'applications flexibles et dynamiques par composition de services.

Les principes de l'approche à services ont été mis en œuvre par diverses approches académiques et industrielles. Certaines approches proposent des modèles pour le développement de services et/ou pour sa composition. Quelques-unes proposent des langages pour la description de l'architecture des applications. D'autres offrent des plates-formes pour supporter leur exécution. Cependant, en général, les approches permettant la composition ne permettent pas le dynamisme : la composition est relativement statique empêchant toute substitution de composants à l'exécution. Inversement, les approches proposant des mécanismes de gestion du dynamisme ne proposent pas de mécanismes de composition d'applications et les applications sont donc difficiles à administrer. Il existe en effet une dissociation entre le développement et l'exécution des applications : les informations présentes lors du

développement d'une application, notamment les concepts d'implémentation, de dépendance, d'architecture, ne sont pas connues à l'exécution. Cette séparation de la connaissance entre le développement et l'exécution rend difficile de contrôler et de garantir l'exécution des applications. La construction et la maintenance d'applications à services reste ainsi une tâche assez complexe.

L'approche à composants et l'approche à services peuvent être vues comme les deux extrêmes d'un spectre qui représente le dégrée de dynamisme d'une application, allant d'une composition statique, fortement déterministe et contrôlée, vers une composition, flexible, dynamique, non-déterministe, peu structurée et peu contrôlée. Ces deux approches apparaissent fréquemment dans la littérature comme étant concurrentes. Cependant, il n'y a pas d'antagonisme entre ces approches, tout au contraire, elles peuvent être vues comme des approches complémentaires afin de permettre la construction d'applications par composition de composants qui fournissent et requièrent des services.

Les travaux de cette thèse visent ainsi à fournir une approche, dirigée par les modèles, qui combine les avantages de l'approche à composants et de l'approche à services, afin de permettre la conception, le développement et l'exécution contrôlés d'applications à services où certaines parties sont déterministes et statiques, tandis que d'autres sont flexibles, dynamiques et non-déterministes.

Nos travaux se placent ainsi à l'intersection entre l'ingénierie dirigée par les modèles, l'approche à composants et l'approche à services.

2 OBJECTIFS

L'objectif principal de cette thèse est de proposer une approche pour **faciliter la construction d'applications flexibles et dynamiques à base de services** en couvrant leur cycle de vie, de leur conception à la gestion de leur exécution. De manière plus détaillée, nos objectifs sont les suivants :

- **proposer un formalisme permettant de définir une application à services en intention**, à partir de ses propriétés, ses contraintes et ses préférences de composition. Ce formalisme doit permettre de spécifier un cadre abstrait, structurel et sémantique, qui permette de guider la composition de l'application.
- **proposer des formalismes et des mécanismes permettant de réaliser la composition d'une application graduellement avant et pendant l'exécution de l'application**. Les mécanismes de composition, présents tout au long du cycle de vie des applications, doivent vérifier et garantir la conformité et la cohérence de la composition des applications vis-à-vis de leur définition.
- **fournir des environnements et des outils liés à chaque phase du cycle de vie des applications**. En particulier, nous considérons des environnements et des outils pour la conception, le développement, le déploiement de services et l'exécution d'applications à services. De tels environnements doivent être extensibles afin d'ajouter, par exemple, des mécanismes supportant la définition et la gestion de propriétés orthogonales à la fonctionnalité métier des applications.

3 ORGANISATION DU DOCUMENT

Après cette introduction, ce document est divisé en deux grandes parties : l'état de l'art et la contribution. L'état de l'art est présenté en trois chapitres :

- le **Chapitre 2** présente l'approche à composants, l'approche à services et l'approche à composants à services, ainsi que diverses technologies que les implantent. Ensuite, nous présentons la notion d'architecture logicielle vue comme le concept central de conception d'applications. Enfin, nous présentons les concepts de base de l'ingénierie dirigée par les modèles et le principe des modèles à l'exécution (*models@run.time*).
- le **Chapitre 3** présente les concepts liés à l'adaptation dynamique des applications. Nous définissons les concepts d'application dynamique et d'application adaptative. Nous montrons comment le dynamisme et l'adaptation d'applications sont traités par différentes approches.
- le **Chapitre 4** décrit la composition de services. Nous présentons les différentes approches de composition. Ensuite, nous présentons le dynamisme et l'adaptation dans la composition de services. Enfin, nous décrivons et classifions divers travaux pour la construction d'applications dynamiques et adaptatives par composition de services.

La contribution est présentée en trois chapitres :

- le **Chapitre 5** présente la vision globale de notre approche qui structure la construction d'applications en trois phases : la phase de conception, la phase de développement et de configuration, et la phase d'exécution des applications.
- le **Chapitre 6** détaille la mise en place de notre approche. Nous présentons les principes et les mécanismes sur lesquels notre approche est fondée. Ensuite, nous présentons le métamodèle de composants à services proposé qui définit les concepts nécessaires pour la construction d'applications, en couvrant leur cycle de vie, de leur conception jusqu'à la gestion de leur exécution.
- le **Chapitre 7** présente en détail la mise en œuvre de notre proposition pour la construction d'applications à services flexibles et dynamiques. Nous présentons notre système, appelé COMPASS, qui gère les phases de conception, de développement et d'exécution d'applications.

Enfin, le **Chapitre 8** synthétise les principales idées de notre proposition. Nous présentons les principales contributions et les perspectives possibles de nos travaux.

PREMIERE PARTIE :

ETAT DE L'ART

CHAPITRE 2
CONTEXTE

Dans une première partie, nous présentons différentes approches pour la construction d'applications. Nous introduisons tout d'abord l'approche à composants et présentons brièvement deux technologies qui la mettent en œuvre : EJB et Fractal. Nous présentons ensuite l'approche à services, apparue pour faire face à deux défis de l'informatique moderne : l'hétérogénéité et le dynamisme. Puis, nous décrivons deux technologies qui mettent en œuvre les principes de cette approche : la plate-forme à services OSGi et les services Web. Nous présentons ensuite l'approche à composants à services qui combine les avantages de l'approche à composants et de l'approche à services. Nous présentons trois technologies qui implémentent cette approche : SCA, iPOJO et SAM. Dans une deuxième partie, nous présentons la notion d'architecture logicielle vue comme le concept central de la conception d'applications. Nous introduisons, dans une troisième partie, les concepts et les propriétés de base de l'ingénierie dirigée par les modèles (IDM), et présentons la vision des modèles à l'exécution : *models@run.time*.

1 COMPOSANTS ET SERVICES

Construire une application par composition de pièces indépendantes et réutilisables est un défi majeur de recherche depuis le début du génie logiciel [8]. Des nombreuses approches ont été proposées pour le développement de logiciels par réutilisation et composition de pièces indépendantes. Dans cette section, nous nous intéressons aux approches à composants, à services et à composants à services. Nous nous intéressons à leurs avantages et surtout à leurs limitations qui nous permettront d'expliquer une partie de la problématique abordée par cette thèse.

1.1 APPROCHE A COMPOSANTS

L'approche à composants [4], apparue au milieu des années 90, a été motivée d'un côté par des raisons économiques – réduction du temps et du coût de développement, spécialisation des acteurs intervenant dans le cycle de vie du développement – et d'un autre coté par la proposition de solutions aux limitations de l'approche à objets [5]. Cette approche promeut la construction d'applications à partir d'assemblages de briques logicielles bien définies et indépendantes appelées composants. Dans la suite, nous détaillons la notion de composant. Nous présentons brièvement deux technologies qui mettent en œuvre l'approche à composants : EJB et Fractal. Enfin, nous exposons les avantages et les limitations de cette approche.

1.1.1 CONCEPTS DE BASE

Il n'existe pas de définition consensuelle de la notion de composant logiciel. Une définition largement citée dans la littérature est celle de Szyperski :

> « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* » [4]

Cette définition contient les concepts caractéristiques des composants qui sont l'auto-description et la réutilisation : un composant spécifie ses interfaces fonctionnelles et peut être utilisé et composé par des tiers en utilisant ces interfaces.

Une autre définition de composant citée fréquemment est celle donnée par Heineman et Councill :

> « *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.* » [9]

Selon cette définition, un composant doit être conforme à un modèle de composants. Cette définition précise ainsi la nécessité d'un modèle de composants, d'un mécanisme de composition et d'une infrastructure d'exécution. Heineman et Councill proposent la définition suivante d'un modèle de composants :

> « *A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model.* » [9]

Un **modèle à composants** définit la structure des composants et la manière de réaliser des assemblages. Une **infrastructure d'exécution** implémente un modèle à composants et fournit les mécanismes nécessaires pour exécuter des composants conformes au modèle.

D'après les définitions citées auparavant, nous constatons que les caractéristiques d'un composant sont réparties dans trois entités différentes : type de composant, unité de déploiement et instance de composant. En effet, dans la littérature, ces trois entités sont souvent confondues et sont toutes appelées composants, ou bien le terme composant est utilisé pour faire référence à une seule

d'entre elles. Pourtant, chacune représente un concept spécifique à un niveau du cycle de vie des composants. Notamment, au développement le concept de composant est une implémentation, tandis qu'à l'exécution il est une instance de composant. Dans ce travail nous faisons la distinction entre les concepts présents à chaque niveau du cycle de vie (voir la Figure 1). Nous distinguons donc les entités suivantes dans lesquelles sont réparties les caractéristiques d'un composant :

- **le type de composant** (similaire à une classe de la programmation à objets) qui est composé généralement par des interfaces fournies et requises, l'**implémentation**, des propriétés de configuration, des interfaces de contrôle. Un type de composant permet de créer des instances de composants. Il peut jouer le rôle de contrat de structure et de comportement qui doit être satisfait par les différentes implémentations.
- **l'unité de déploiement** qui contient tous les éléments nécessaires pour créer des instances du type de composant : l'implémentation du composant (le code binaire), des ressources (des librairies, des images), des fichiers de configuration, des informations d'assemblage, etc.
- **l'instance de composant** qui fournit, à l'exécution, les fonctionnalités de son type de composant. Une instance est créée en général à partir d'une fabrique associée au type de composant. La création d'instances peut suivre différentes politiques d'instanciation : instances multiples, partagées, limitées.

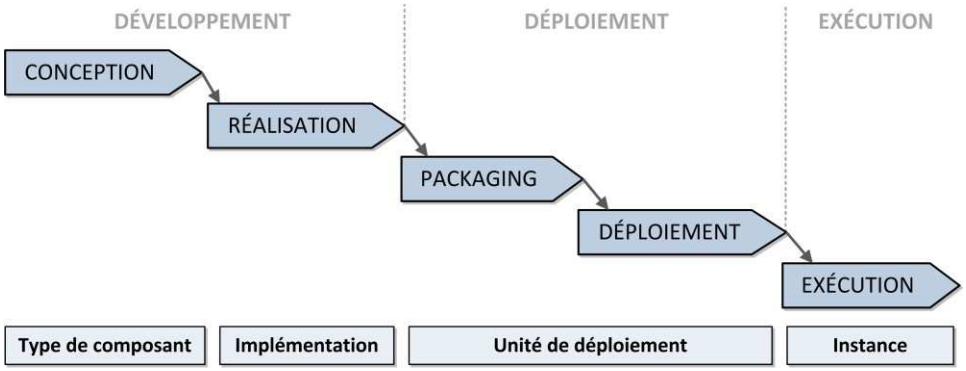


Figure 1. Cycle de vie d'un composant et formes de composants

La **composition** ou assemblage de composants est, en plus du développement, une activité fondamentale de l'approche à composants pour la construction d'applications. Toutefois, le fait que le concept de composant ne soit pas clairement défini dans la littérature rend vague le concept de composition.

En général, la composition est définie comme le processus de lier des composants existants entre eux par leurs interfaces requises et fournies. En particulier, la composition est centrée implémentation : elle résulte dans une architecture d'implémentations connectées entre elles, appelée configuration. L'information contenue dans une configuration est utilisée pour créer, configurer et connecter les instances de composant correspondantes. La composition peut être réalisée à différentes étapes du cycle de vie des composants (au développement, au déploiement, à l'exécution) et manipuler ainsi des entités différentes (des implémentations, des unités de déploiement, des instances).

La composition peut être définie par des langages déclaratifs de composition (des ADLs[1]), ou de façon impérative par des langages de programmation ou de script. La composition déclarative résulte généralement dans des architectures statiques obtenues au développement. La composition impérative permet plus facilement de réaliser des changements dans l'architecture pendant l'exécution (création et destruction d'instances et de connecteurs).

[1] *Architecture Description Language*

Une infrastructure d'exécution fournit les mécanismes nécessaires pour gérer l'exécution d'instances de composants, y compris les mécanismes de déploiement. Elle peut fournir aussi des mécanismes pour gérer des aspects non-fonctionnels, tels que la distribution, la sécurité ou les transactions. En effet, divers modèles à composants proposent des mécanismes qui permettent la séparation entre la logique métier et des aspects non-fonctionnels.

Une approche fréquemment utilisée pour réaliser la séparation de préoccupations est l'utilisation des conteneurs. Un conteneur, représenté schématiquement comme une coquille qui entoure le contenu d'une instance de composant (voir la Figure 2), gère le cycle de vie de l'instance, ses interactions avec d'autres instances (à travers les interfaces fournies et requises), ainsi que ses services techniques requis (à travers les interfaces de contrôle). Le contenu peut être un objet de l'implémentation définie dans le type de composant, ou d'autres instances de composants (composition hiérarchique).

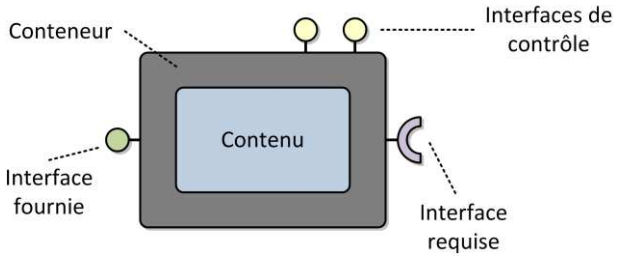


Figure 2. Une instance de composant et son conteneur

La relation entre le conteneur et le contenu s'effectue en utilisant les patrons d'inversion de contrôle et d'injection de dépendance (cf. Chapitre 3, section 3.3) : le conteneur injecte dans le contenu les objets requis, et gère l'accès au contenu (les appels depuis l'extérieur). Ainsi, différents mécanismes, d'introspection ou d'adaptation dynamique par exemple, peuvent être fournis par l'infrastructure d'exécution afin de gérer, à travers les conteneurs, différentes propriétés non-fonctionnelles des composants.

De plus, lorsque le modèle à composant permet la composition hiérarchique, une application est décomposée en plusieurs instances de composant, reliées par des connecteurs, s'exécutant à l'intérieur d'une instance de composant composite : le contenu de l'application peut ainsi être géré par le conteneur de l'instance composite.

1.1.2 TECHNOLOGIES EXISTANTES

De nombreux modèles à composants ont été proposés tant dans le monde académique qu'industriel. Nous pouvons citer COM[2], EJB et CCM[3] issus de l'industrie, ou ArchJava [10], Fractal, K-Component [11] et SOFA 2.0 [12] issus de la recherche.

Les modèles à composants existants ont des objectifs particuliers et suivent donc des principes différents. Certains modèles supportent seulement la phase de développement, bien que d'autres fournissent aussi des mécanismes pour supporter la phase d'exécution. Ces derniers possèdent des infrastructures d'exécution propres capables d'exécuter et de gérer des composants conformément au modèle. Divers modèles ont pour objectif de supporter le dynamisme : certains d'entre eux proposent des mécanismes capables de réaliser la mise à jour de composants de manière dynamique, tandis que d'autres proposent des mécanismes d'adaptation de l'architecture d'une application. Plusieurs modèles proposent des langages de description d'architectures (des ADLs), permettant d'architecturer un

[2] *Component Object Model* **http://www.microsoft.com/COM/**
[3] *CORBA Component Model* **http://www.omg.org/spec/CCM/**

système en décrivant ses composants et ses connecteurs. Enfin, certains modèles fournissent aussi des mécanismes pour le support d'aspects non-fonctionnels.

Les modèles existants ont de nombreuses similitudes, mais aussi des différences importantes, et dans beaucoup de cas leurs concepts sont peu clairs. Afin d'améliorer la compréhension des concepts et de différencier plus facilement les modèles existants, divers travaux ont essayé d'identifier les principes de base d'un modèle à composants et ont proposé des cadres de classification et de comparaison de modèles [6] [13].

Dans la suite, nous présentons brièvement les modèles EJB et Fractal qui nous semblent représentatifs des modèles à composants. D'autres modèles à composants qui considèrent le dynamisme des composants sont présentés dans le Chapitre 3 (cf. section 4.3).

1.1.2.1 EJB

La technologie EJB[4], acronyme de *Enterprise JavaBeans*, développée par *Sun MicroSystems*, fournit un modèle à composants (du côté serveur) pour la construction modulaire d'applications. EJB fournit un ensemble de services communs – transactions, persistance, concurrence, sécurité – permettant ainsi aux développeurs de se concentrer sur les problèmes liés à la logique métier des applications.

Un type de composant EJB est constitué d'une unique interface fonctionnelle fournie : l'interface *remote* (voir la Figure 3). EJB ne permet pas de décrire explicitement les interfaces requises d'un composant. L'implémentation du composant est réalisée par une seule classe Java qui implémente, en plus des méthodes de l'interface *remote*, un ensemble de méthodes de contrôle spécifiques au type de composant.

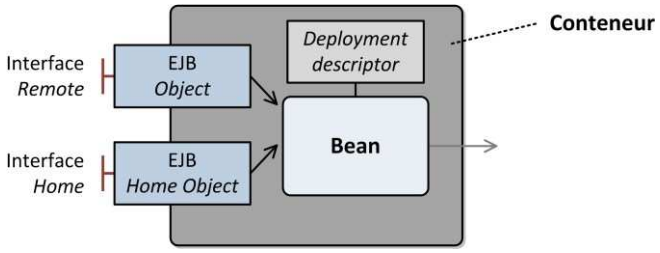


Figure 3. Un type de composant EJB

EJB propose trois types de composants : session (*SessionBeans*), entité (*EntityBeans*) et message (*MessageDrivenBeans*). Une instance de composant session est créée lors du premier accès par un utilisateur et sa durée de vie correspond à la durée de l'interaction avec cet utilisateur. Selon le type d'interaction avec l'utilisateur, une instance de composant session peut avoir un état (*stateful*) ou non (*stateless*). Une instance de composant entité représente une entité stockée dans un entrepôt de données persistant, comme une base de données. Une instance de composant message ressemble à une instance de composant session, mais elle traite les messages de manière asynchrone. Les instances de composants sont créées et détruites en utilisant l'interface *home*, qui permet aussi de retrouver des instances persistantes à partir de certains critères (par exemple, par leur identifiant).

Un type de composant EJB est packagé dans un fichier de type JAR, déployable sur des serveurs EJB, qui contient, en plus de l'implémentation du composant, un fichier de configuration dans lesquels sont déclarées des propriétés et des dépendances de déploiement.

Les instances de composants EJB résident dans des conteneurs (nommés *EJB containers*), installés dans des serveurs de l'environnement d'exécution, qui fournissent les services de gestion du cycle de

[4] **http://www.oracle.com/technetwork/java/javaee/ejb/**

vie, des transactions, de la persistance, de la sécurité, et aussi d'autres services. Les appels aux méthodes d'une instance de composant sont ainsi interceptés par son conteneur.

En général, la composition de composants EJB est faite de manière impérative : des nouveaux composants, chargés de créer et de connecter des instances de composants EJB, sont développés.

1.1.2.2 FRACTAL

Fractal[5] [14], développé par France Télécom R&D et l'INRIA, propose un modèle à composants générique et extensible qui permet la conception, l'implémentation, le déploiement et la gestion d'applications logicielles. Le modèle à composants Fractal permet de séparer les aspects fonctionnels des aspects non-fonctionnels (introspection, configuration, sécurité, transactions).

L'implémentation d'un composant Fractal est divisée en deux parties : le contrôleur et le contenu (voir la Figure 4). Le contrôleur peut implémenter des interfaces de contrôle, dont certaines sont définies dans la spécification du modèle à composants, qui permettent l'introspection et la manipulation du composant. Le contrôleur délègue les messages reçus depuis l'extérieur à son contenu, qui peut contenir du code fonctionnel (composant primitif), ou bien d'autres contrôleurs (composant composite) : le modèle Fractal supporte la composition hiérarchique.

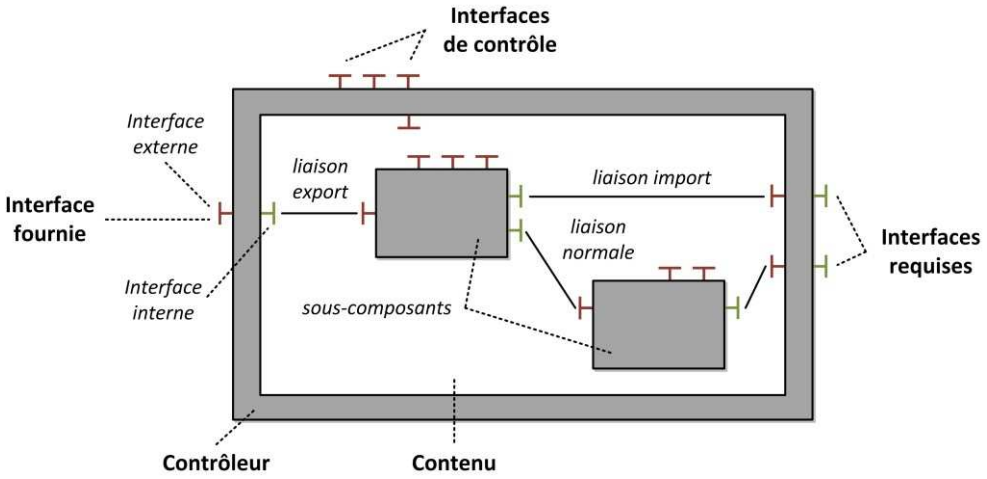


Figure 4. Un type de composant Fractal

Un type d'un composant Fractal (*ComponentType*) est constitué par un ensemble d'interfaces fournies (métier et de contrôle) et requises. Les interfaces métier sont les points d'accès externes aux interfaces des composants, tandis que celles de contrôle prennent en charge des aspects non-fonctionnels comme la gestion du cycle de vie ou des liaisons. Les interfaces requises peuvent être spécifiées comme obligatoires ou optionnelles, et avoir une cardinalité simple ou multiple.

Les instances de composants sont créées à partir de fabriques. Une fabrique fournit une méthode de création (*newFCInstance*) qui retourne une nouvelle instance de composant à chaque invocation. Le cycle de vie d'une instance peut être géré si son contrôleur implémente l'interface *LifeCycleController*.

Fractal fournit deux mécanismes pour la composition d'applications : les liaisons (*bindings*) entre composants, et le regroupement d'un ensemble fini de composants dans un composite. La composition peut être faite de manière déclarative à travers le langage ADL Fractal.

[5] **http://fractal.ow2.org/**

La composition de composants est réalisée au travers de différentes interfaces de contrôle permettant de réaliser la configuration d'instances (*AttributeController*), la liaison d'instances (*BindingController*), la création et la manipulation de composites (*ContentController*). Fractal offre, via son API, des mécanismes pour faire évoluer dynamiquement l'architecture d'une application : créer et détruire des instances et des liaisons.

Fractal ne définit pas de mécanisme particulier pour le packaging ou le déploiement de composants.

Il existe diverses implémentations du modèle Fractal pour des langages de programmation différents : Julia[6] [15] l'implémentation de référence en Java, Think[7] une implémentation en C pour le développement d'applications embarquées, AOKell[8] [16] une implémentation en Java à l'aide du langage AspectJ, entre autres. Le modèle à composants Fractal est aujourd'hui utilisé dans de nombreux projets de recherche.

1.1.3 AVANTAGES ET LIMITATIONS DE L'APPROCHE A COMPOSANTS

L'approche à composants introduit une méthodologie visant à faciliter la construction d'applications par réutilisation et composition de composants. Elle est basée sur l'idée que le développement et la composition de composants peuvent être réalisés de façon totalement séparée par des acteurs différents dans des emplacements différents. Cette approche propose une séparation claire entre les interfaces qui décrivent les fonctionnalités fournies d'un composant et leur implémentation. Elle promeut aussi la séparation des aspects fonctionnels et non-fonctionnels, afin de permettre aux développeurs de se concentrer sur l'implémentation de la logique métier des composants, et de faciliter l'administration et la maintenance des applications. L'approche à composants aborde principalement le développement de composants : la composition de composants est centrée implémentation est peut être définie par une description d'architecture.

Malgré les avantages de cette approche, les modèles à composants existants souffrent de limitations. Ces modèles ont des objectifs différents et suivent donc des principes différents : certains modèles supportent seulement la phase de développement, d'autres fournissent des mécanismes pour supporter aussi la phase d'exécution. En général, les concepts proposés par ces modèles sont confus et la distinction entre les concepts de type de composant, implémentation, unité de déploiement, et instance de composant n'est pas présente ou n'est pas évidente. En outre, les mécanismes de composition d'applications proposés par ces modèles souffrent de limitations. Les applications sont généralement composées lors de la phase de développement ou, au plus tard, lors de la phase de déploiement. Une fois la composition de composants effectuée, des changements dynamiques dans l'architecture d'exécution sont limités (création et destruction d'instances de types de composants connus et de connecteurs) ou ne sont pas possibles : la disparition dynamique de types de composants n'est pas une hypothèse de l'approche à composants.

Une limitation majeure des approches à composants est donc le manque de flexibilité et de dynamisme des applications composées. L'approche à services, présentée dans la section suivante, est apparue pour faire face aux besoins de flexibilité et de dynamisme des applications.

[6] **http://fractal.ow2.org/julia/**
[7] **http://think.ow2.org/**
[8] **http://fractal.ow2.org/aokell/**

1.2 APPROCHE A SERVICES

L'approche à services [7], en anglais *Service-Oriented Computing* (SOC), est un paradigme relativement récent conçu à l'origine pour permettre l'intégration de systèmes logiciels hétérogènes et distribués. Cette approche propose un style architectural qui cherche à faciliter la construction d'applications à partir d'entités logicielles faiblement couplées, hétérogènes et distribuées, nommées services. Dans cette section, nous présentons tout d'abord la définition du concept de service. Nous présentons ensuite les différents acteurs de cette approche ainsi que leurs interactions. Nous détaillons l'architecture à services et présentons deux exemples d'architectures à services : OSGi et les services Web.

1.2.1 CONCEPTS DE BASE

La notion de service constitue le concept de base de l'approche à services. Cependant, il n'existe pas une définition consensuelle du concept de service. Deux définitions fréquemment citées dans la littérature sont celles proposées par Papazoglou:

> « *Services are self-describing, platform agnostic computational elements.* » [7]

> « *Services are autonomous, platform-independent entities that can be described, published, discovery, and loosely coupled in novel way.* » [17]

D'après ces définitions, un service est une entité logicielle qui peut être utilisée grâce à sa description. Un consommateur utilise un service sans avoir connaissance de la technologie utilisée pour son implémentation ainsi que de sa plate-forme d'exécution. De plus, un service ne connaît pas le contexte dans lequel il sera utilisé par un consommateur. Cette indépendance offre un faible couplage entre les fournisseurs et les consommateurs de services.

Une autre définition de service couramment retenue est celle proposée par Arsanjani :

> « *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. The service provider realizes the service description implementation and also delivers the quality of service requirements to the service consumer. Services should ideally be governed by declarative policies and thus support a dynamically re-configurable architectural style.* » [18]

Cette définition décrit le principe d'interaction existant entre les fournisseurs et les consommateurs de services. Un consommateur recherche un fournisseur en utilisant une description de service. Une fois un fournisseur trouvé, le consommateur se lie à lui et l'invoque afin d'utiliser ses fonctionnalités. Cette définition indique aussi qu'un service devrait être géré par des propriétés exprimées de façon déclarative, et permettre ainsi la reconfiguration dynamique de services.

Sur la base de ces définitions, nous définissons le concept de service comme suit :

> *Un service est une entité logicielle constituée d'une spécification et d'un objet de service (une instance de service). La spécification de service contient la description des fonctionnalités offertes par le service, mais aussi des informations sur son comportement ou sur ses aspects non-fonctionnels. L'objet de service fournit l'ensemble des fonctionnalités définies dans la spécification du service. Une spécification de service est utilisée par les consommateurs pour rechercher un fournisseur et se lier à lui.*

Par abus du langage, nous dirons par la suite « service » pour « objet de service ».

1.2.2 SERVICE-ORIENTED COMPUTING : SOC

L'approche à services est un paradigme de programmation, un style architectural, qui utilise les services comme entités de base. Le but de cette approche est de permettre la construction d'applications en utilisant des entités faiblement couplées. L'approche propose ainsi un principe d'interaction dans lequel interviennent trois types d'acteurs (voir la Figure 5) :

- des **fournisseurs de services** qui offrent des services décrits dans des spécifications de services,
- des **consommateurs de services** qui requièrent et utilisent des services offerts par des fournisseurs, et
- un **annuaire de services** qui fournit un ensemble de mécanismes permettant la publication, la recherche, la découverte et la sélection de services.

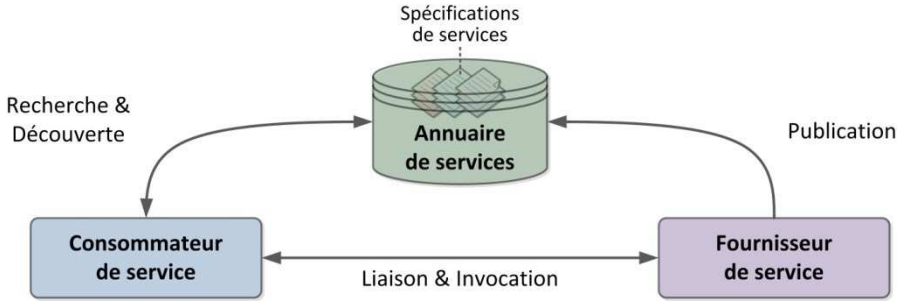


Figure 5. Principe d'interaction de l'approche à services

L'approche à services propose trois primitives d'interaction entre les différents acteurs :

- la **publication de services** par leurs fournisseurs : les fournisseurs s'enregistrent auprès de l'annuaire de services avec les spécifications des services fournis.
- la **recherche de services** par les consommateurs : les consommateurs interrogent l'annuaire de services afin de rechercher et découvrir les services qui correspondent à leurs besoins.
- la **liaison** entre un consommateur et un fournisseur de service. Une fois que le fournisseur d'un service est découvert, le consommateur peut se lier à lui et l'invoquer afin d'utiliser les fonctionnalités fournies.

Ces interactions peuvent avoir lieu à n'importe quel moment dans le cycle de vie de l'application : au développement, au déploiement ou à l'exécution. Grâce à ce principe d'interaction, les applications à services bénéficient des propriétés suivantes :

- un **faible couplage** : la seule information partagée entre un fournisseur et un consommateur est la spécification de service, qui ne contient pas les détails liés à la technologie utilisée pour la réalisation du service. Ainsi, tant l'hétérogénéité comme la distribution des services sont masquées : un consommateur n'a pas à connaitre l'implémentation du service utilisé, ni sa localisation.
- des **liaisons tardives** : la liaison entre un fournisseur et un consommateur de service est établie seulement lorsque le fournisseur est trouvé et lorsque le consommateur le demande. Cette propriété de liaison tardive offre la possibilité de sélectionner à l'exécution les fournisseurs des services requis par une application. Dans cette thèse, nous nous intéressons particulièrement à cette propriété.

Ce principe d'interaction fournit la base pour supporter le dynamisme dans l'approche à services. Les détails de cette approche, nommée approche à services dynamique, sont présentés dans le Chapitre 3 (cf. section 4.1).

1.2.3 SERVICE-ORIENTED ARCHITECTURE : SOA

Une architecture à services, en anglais *Service-Oriented Architecture* (SOA), est une réalisation particulière de l'approche à services, qui fournit un environnement d'intégration et d'exécution de services. Un tel environnement doit être capable de gérer les interactions entre les différents acteurs. Pour cela, il doit fournir des **mécanismes de base** qui permettent la publication, la recherche, la découverte, la liaison et l'invocation de services (voir la Figure 6). De plus, en fonction du domaine métier, l'environnement peut fournir des **mécanismes additionnels** pour supporter des aspects non-fonctionnels tels que la sécurité, la distribution, les transactions, ou l'administration.

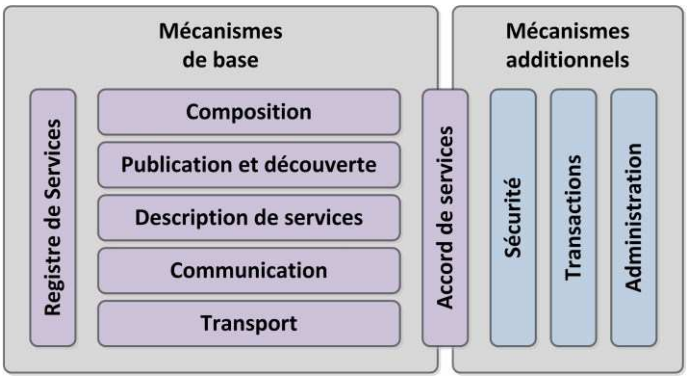


Figure 6. Mécanismes d'un environnement d'intégration et d'exécution de services

Les mécanismes de base sont nécessaires pour la réalisation d'applications à services ; cependant, ils ne sont pas suffisants pour répondre à tous les besoins d'une application. Dans [7], Papazoglou a proposé une architecture à services étendue, fondée sur les mécanismes de base, afin de définir des mécanismes de composition et d'administration d'applications. Les mécanismes de composition permettent la coordination de services et la gestion de compositions en assurant leur conformité. Les mécanismes d'administration permettent la gestion et la surveillance d'applications. Une architecture à services étendue considère aussi la gestion d'aspects non-fonctionnels. Le Chapitre 4 de ce manuscrit est consacré à la composition de services.

1.2.4 TECHNOLOGIES EXISTANTES

Il existe aujourd'hui de nombreuses architectures à services ainsi que de nombreuses architectures à services étendues. CORBA[9], Jini[10], OSGi, les services Web, UPnP[11] et DPWS[12] sont des exemples de SOA. Ils utilisent des technologies différentes pour mettre en place l'approche à services. Les choix des technologies dépendent du domaine métier et des objectifs visés par le SOA. Dans cette section, nous présentons les aspects principaux de la plate-forme à services OSGi et des services Web.

1.2.4.1 OSGI

OSGi [19], acronyme de *Open Services Gateway initiative*, est une spécification de plate-forme à services proposé par le consortium *OSGi Alliance*. A l'origine, la spécification OSGi visait des passerelles résidentielles et des équipements réseaux avec pour objectif d'administrer et mettre à jour de façon dynamique les applications. Aujourd'hui, la spécification OSGi est utilisée dans de nombreux domaines

[9] *Common Object Request Broker Architecture* **http://www.omg.org/spec/CORBA/**
[10] **http://river.apache.org/**
[11] *Universal Plug and Play* **http://www.upnp.org/**
[12] *Devices Profile for Web Services*

d'application : serveurs d'applications (comme Jonas[13] ou Glassfish[14]), systèmes embarqués (des téléphones portables par exemple), outils de développement intégrés (comme Eclipse[15]), objets industriels (automobiles, transports en commun).

La spécification OSGi définit un **modèle à services**, une **plate-forme de déploiement et d'exécution de services**, ainsi qu'un ensemble de services techniques : sécurité, monitoring, événements, journalisation, administration.

La spécification OSGi s'appuie sur le langage Java. Les services sont décrits par des interfaces Java et des propriétés. OSGi définit des unités de packaging/déploiement de services appelées ***bundles***. Un *bundle*, implémenté sous la forme d'un fichier JAR, peut contenir des classes Java implémentant les fonctionnalités des services fournis et d'autres ressources : fichiers de configuration, librairies natives, images. L'unité de code partagée entre les *bundles* est le package Java : un *bundle* décrit les packages fournis (exportés) et les packages requis (importés). Les *bundles* sont gérés – installés, démarrés, arrêtés, mis à jour, désinstallés – par la plate-forme d'exécution. La Figure 7 illustre le cycle de vie d'un *bundle*. Les applications sont ainsi composées par l'interconnexion de *bundles*.

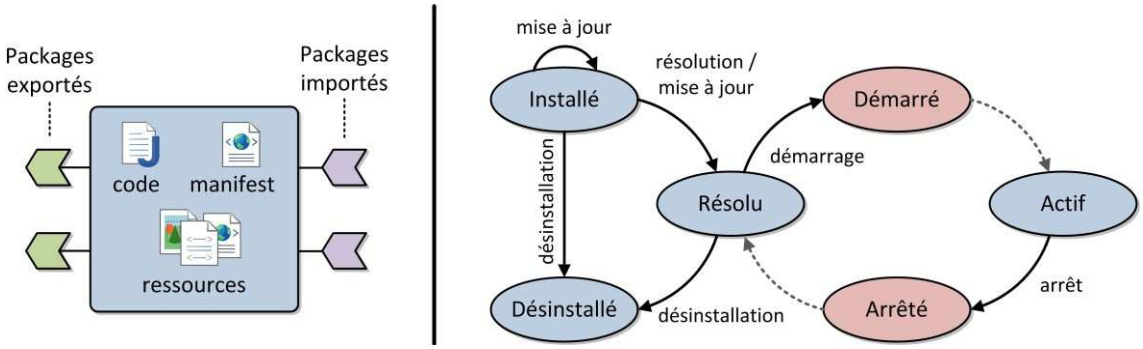


Figure 7. Un *bundle* et son cycle de vie

La spécification OSGi spécifie une plate-forme à services centralisée avec un annuaire de services. Les fournisseurs enregistrent leurs services fournis auprès de l'annuaire par le déploiement et l'activation des *bundles*. Le consommateur d'un service peut consulter l'annuaire afin de découvrir les fournisseurs disponibles du service recherché. Les résultats d'une telle requête peuvent être filtrés par des filtres LDAP portant sur les propriétés du service. En réponse, l'annuaire renvoie l'ensemble des services compatibles à la requête sous la forme de références de services. Lorsque le consommateur a choisi un service (sa référence), il peut demander à l'annuaire l'objet de service associé à la référence choisie. L'annuaire renvoie une référence directe sur le fournisseur que le consommateur peut utiliser.

De plus, la spécification OSGi définit des primitives et des mécanismes afin de procurer une plate-forme à services dynamique : les fournisseurs peuvent retirer leurs services de l'annuaire ou modifier leurs propriétés à tout moment, les consommateurs peuvent être notifiés lorsqu'un service utilisé n'est plus disponible ou lorsqu'un nouveau fournisseur de service apparaît ; toutefois, les consommateurs doivent explicitement libérer les services utilisés lorsqu'ils ne sont plus disponibles, ce qui constitue une contrainte forte pour les développeurs d'applications.

Il existe aujourd'hui plusieurs implémentations de la spécification OSGi, les principales étant Felix[16], Knopflerfish[17] et Equinox[18]. Théoriquement, les services OSGi sont interopérables et peuvent

[13] *Java Open Application Server* **http://jonas.ow2.org/**
[14] **http://glassfish.java.net/**
[15] **http://www.eclipse.org/**
[16] **http://felix.apache.org/**
[17] **http://www.knopflerfish.org/**
[18] **http://www.eclipse.org/equinox/**

être utilisés d'une implémentation à l'autre. Cependant, en pratique, il existe des différences qui rendent parfois nécessaires des adaptations.

En conclusion, OSGi considère le dynamisme du monde réel où les dispositifs peuvent apparaître ou disparaître à tout moment. OSGi propose ainsi, en plus des mécanismes de base de l'approche à services, des mécanismes de déploiement, d'administration, de notification,… qui en font aujourd'hui la plate-forme de référence pour la construction d'applications dynamiques.

Cependant, un défaut majeur de la spécification OSGi est qu'il n'existe pas de modèle de composition de services. De ce fait, la définition et la gestion de la composition d'une application doivent être faites par les développeurs. De plus, afin d'assurer le bon fonctionnement de l'application, les développeurs doivent prendre en compte explicitement le dynamisme des services (apparition, disparition, mise à jour) : le dynamisme des services peut valider ou invalider une composition à tout moment. Les développeurs doivent ainsi gérer l'apparition et la disparition de services, ainsi que relâcher explicitement les références vers les services disparus. Ces tâches sont délicates et demandent une grande connaissance des mécanismes OSGi pour bien traiter tous les cas possibles afin d'éviter les erreurs, rendant ainsi complexe le développement d'applications dynamiques. Diverses approches, comme iPOJO (présentée dans la section 1.3.2.2), ont été proposées pour résoudre des problèmes liés à la définition et à la gestion d'applications sur OSGi.

1.2.4.2 LES SERVICES WEB

Les services Web [20] sont aujourd'hui le SOA le plus connu et le plus populaire dans le monde industriel et académique pour la mise en place d'applications à services. Les services Web ont été conçus pour permettre à des applications hétérogènes s'exécutant au sein de différentes entreprises d'interopérer au travers de leurs services offerts. L'hétérogénéité des applications est considérée à différents niveaux : implémentation des applications, modèles d'interaction, protocoles de communication.

Les services Web sont décrits dans un langage standard appelé WSDL (*Web Service Description Language*). Cette description spécifie la fonctionnalité du service, les types de données employés dans les messages, le protocole de transport utilisé, ainsi que des informations pour localiser le service. Grâce à la description de service, les consommateurs s'abstraient des techniques d'implémentation des services, comme par exemple les langages de programmation utilisés ou leurs plates-formes d'exécution.

L'annuaire de services, appelé UDDI (*Universal Description, Discovery and Integration*), supporte l'enregistrement de descriptions de services, appelées type de services, ainsi que de fournisseurs de services, appelés entreprises. UDDI permet la publication de fournisseurs au moyen de pages blanches qui contient les noms des fournisseurs et leurs descriptions ; des pages jaunes qui catégorisent les fournisseurs ; et des pages vertes qui décrivent les moyens d'interaction avec les fournisseurs. Un fournisseur doit connaître la localisation de l'annuaire UDDI afin de communiquer avec lui et enregistrer un service. Le protocole de communication par défaut est SOAP[19]. L'enregistrement d'un service se fait en deux étapes : la publication du type de service, et la publication du fournisseur du service. L'annuaire UDDI est distribué et l'information est répliquée sur plusieurs sites. De ce fait, un fournisseur ne doit publier sa description de service que vers un seul nœud du réseau d'annuaires. Les fournisseurs peuvent se retirer de l'annuaire ainsi que retirer les types de services fournis.

Les consommateurs de services peuvent communiquer avec l'annuaire afin de découvrir un service fourni ou un fournisseur de service. L'annuaire UDDI fournit des opérations de recherche des fournisseurs et des services fournis par les fournisseurs enregistrés. La recherche d'un fournisseur de service peut s'effectuer lors du développement ou de l'exécution d'une application. Toutefois, le fait de connaître la localisation d'un fournisseur permet d'utiliser le service fourni sans passer

[19] *Simple Object Access Protocol*

nécessairement par l'annuaire. Ainsi, bien que les services Web soient fondés sur les concepts de l'approche à services, l'interaction propre à cette approche n'est pas suivie de manière systématique car la consommation de services Web ne passe pas forcément (voire rarement) par l'annuaire. En effet, la plupart des organisations qui fournissent des services Web indiquent directement leur localisation.

La composition de services Web peut être réalisée de manière comportementale par orchestration ou par chorégraphie de services. L'orchestration décrit la vision centralisée d'une composition de services, tandis que la chorégraphie décrit, d'un point de vue global, la façon dont un ensemble de services collaborent pour atteindre un but commun. Les dépendances des services étant inconnues, la composition structurelle de services Web n'est pas possible, et leur contrôle est plus compliqué. Les détails sur la composition de services sont présentés dans le Chapitre 4.

De nombreuses spécifications et technologies, comme par exemple WS-Transaction[20] et WS-Security[21], ont été créées au-dessus des concepts de base des services Web, transformant ce SOA en SOA étendu.

1.2.5 AVANTAGES ET LIMITATIONS DE L'APPROCHE A SERVICES

L'approche à services est un paradigme qui a pour but de faciliter la construction d'applications à partir de services. Elle favorise ainsi la réutilisation des services existants pour la construction d'applications. L'approche à services aborde spécialement la phase d'exécution de services : la composition d'applications est centrée instance (objet de service).

Une propriété importante de l'approche est le faible couplage entre les consommateurs et les fournisseurs de services. En effet, la seule information partagée entre ces acteurs est la spécification de service, permettant ainsi l'évolution indépendante des services, ainsi que le masquage de l'hétérogénéité et de la distribution des services aux consommateurs.

De plus, l'approche à services permet d'établir les liaisons entre les consommateurs et les fournisseurs de services de façon tardive : la liaison entre les fournisseurs et les consommateurs a lieu seulement lorsqu'un fournisseur est trouvé et lorsque le consommateur le demande. Grâce aux propriétés de faible couplage et de liaison tardive, l'approche à services permet la construction d'applications flexibles et dynamiques.

Néanmoins, les applications construites par composition de services sont difficiles à administrer. En effet, il existe une dissociation entre le développement et l'exécution d'applications : les informations présentes lors du développement d'une application, notamment les concepts d'implémentation, de dépendance, d'architecture, ne sont pas connues à l'exécution. Du point de vue d'une application, les services sont des entités logicielles « boite noire » et les concepts d'implémentation et de dépendance de services sont inconnus à l'exécution. L'application a donc une vue « utilisateur de services » ignorant leur mise en œuvre, tandis que les composants ont une vue « fournisseur de services » ignorant l'application à laquelle ils participent. Cette séparation de la connaissance entre le développement et l'exécution rend difficile de contrôler et de garantir l'exécution des applications.

L'approche à composants à services vise à résoudre les problèmes liés à la construction d'applications à services en associant les concepts de composant et de service.

[20] **https://www.oasis-open.org/committees/ws-tx/**
[21] **https://www.oasis-open.org/committees/wss/**

1.3 APPROCHE A COMPOSANTS A SERVICES

L'approche à services a donné naissance à un nouveau type de modèle à composants, appelé modèle à composants à services [21], qui combine les avantages de l'approche à composants : description d'architectures ; et de l'approche à services : faible couplage, dynamisme, afin de faciliter la construction d'applications dynamiques.

L'approche définit un modèle à composants à services qui est un modèle à composants dans lequel les concepts de l'approche à services sont introduits. Cette section décrit les principes de l'approche et présente quelques technologies qui la mettent en œuvre.

1.3.1 PRINCIPES

L'approche à composants à services propose d'utiliser des composants pour implémenter des services. Cette approche emprunte aux composants l'idée de séparation des aspects fonctionnels et non-fonctionnels, afin de séparer le code de gestion des aspects services (publication, recherche, sélection, liaison, invocation, ...) du code implémentant la logique métier des services. Le but est ainsi de déléguer les mécanismes liés à l'approche à services aux conteneurs des composants, qui interagiront avec l'annuaire de services de la plate-forme sous-jacente.

Les principes de cette approche, décrits dans [22], sont les suivants :

- **un service est une fonctionnalité fournie**. Un service est un ensemble d'opérations réutilisables.
- **un service est caractérisé par une spécification de service** qui peut contenir des informations syntaxiques, comportementales et sémantiques, ainsi des dépendances vers d'autres spécifications de services.
- **un composant implémente une ou plusieurs spécifications de service** en respectant leurs contraintes. En plus des dépendances spécifiées dans les spécifications implémentées (dépendance de spécification), un composant peut déclarer des dépendances additionnelles vers d'autres services (dépendances d'implémentation). Les services sont ainsi le seul moyen d'interaction entre les instances de composants.
- **le principe d'interaction de l'approche à services est utilisé pour résoudre les dépendances de services**. Les services, fournis par des instances de composants, sont enregistrés auprès d'un annuaire de services. L'annuaire de services est utilisé pour découvrir des services de façon dynamique afin de résoudre des dépendances de services.
- **les compositions de services sont décrites en termes de spécifications de services**. Une composition (abstraite) est un ensemble de spécifications qui seront utilisées pour sélectionner des composants concrets. Des liaisons explicites ne sont pas nécessaires car elles seront inférées à l'exécution à partir des dépendances déclarées dans les spécifications et les composants participant à la composition.
- **les spécifications de services sont la base de la substitution**. Dans une composition, tout composant implémentant une spécification donnée peut être remplacé par un composant alternatif qui implémente la même spécification.

Le modèle à composants à services résultant facilite ainsi la construction d'applications flexibles et dynamiques. En effet, les applications sont définies à un niveau d'abstraction plus élevé, en termes de spécifications de services (en non pas en termes d'implémentations comme dans les modèles à composants traditionnels).

Le principe d'interaction de l'approche à services est utilisé pour résoudre les dépendances de services, permettant de retarder la sélection des implémentations (i.e., des fournisseurs de services) jusqu'à l'exécution (contrairement aux modèles à composants traditionnels qui effectuent la sélection avant l'exécution).

Les applications peuvent être définies par une composition comportementale comme l'orchestration, mais aussi par une composition structurelle. De plus, les applications peuvent être composées statiquement avant l'exécution (comme dans les modèles à composants traditionnels), mais aussi dynamiquement, à l'exécution, et par opportunisme. Entre ces deux extrêmes, il est possible de définir des applications dans lesquelles certaines parties de l'architecture sont définies statiquement, et d'autres sont définies dynamiquement et autorisées à varier pendant l'exécution.

1.3.2 TECHNOLOGIES EXISTANTES

Il existe aujourd'hui plusieurs implémentations de modèles à composants à services : *Service Binder* [22], *Declarative Services* [23], *Dependency Manager* [24], *Spring Dynamic Modules* [25], SCA, iPOJO, SAM en sont quelques exemples. Dans la suite, nous présentons les modèles SCA, iPOJO et SAM, en nous concentrant uniquement sur les concepts de conception/développement de composants définis par ces modèles. Les concepts liés à la définition, l'exécution et l'administration d'applications seront présentés plus tard dans le Chapitre 4 de ce document.

1.3.2.1 SCA

SCA [26], acronyme de *Service Component Architecture*, est un ensemble de spécifications définies par le consortium OSOA[22]. SCA spécifie un modèle pour la création de composants, et un modèle pour la construction d'applications à composants. Dans le Tableau 1, nous présentons les concepts du modèle à composants en les classifiant comme des concepts de niveau développement ou exécution.

	Concept	***Définition***
Développement	***Service***	*Un service définit les fonctionnalités métiers fournies par une implémentation. La description des services est faite selon la technologie utilisée pour réaliser la logique métier : Java, WSDL.*
	Référence	*Une référence représente un service requis (une dépendance) d'un composant.*
	Implémentation	*Une implémentation réalise un ou plusieurs services. Elle peut être écrite avec différents langages : Java, BPEL, C++.*
	Propriété	*Une propriété permet la configuration d'une implémentation. Les propriétés sont typées et peuvent être de type simple ou complexe.*
	Type de composant	*Un type de composant représente les aspects configurables d'une implémentation : services, références et propriétés.*
Exécution	***Composant***	*Un composant est une instance configurée d'une implémentation. Il fournit et utilise des services. Plusieurs composants peuvent utiliser et configurer la même implémentation ; chaque composant configure différemment une implémentation.*

Tableau 1. Concepts du modèle à composants SCA

La vue externe d'un composant SCA, présentée dans la Figure 8, est représentée par les services fournis, les références de services et les propriétés associées. Les services, références et propriétés sont des aspects configurables qui constituent le type d'un composant. Le type correspond à une implémentation réalisant, entre autres, la fonctionnalité des services fournis. En indiquant des valeurs concrètes pour les services, les références et les propriétés, un composant réalise la configuration d'une implémentation qui se traduira à l'exécution par une instance de cette implémentation.

[22] *Open Service Oriented Architecture*

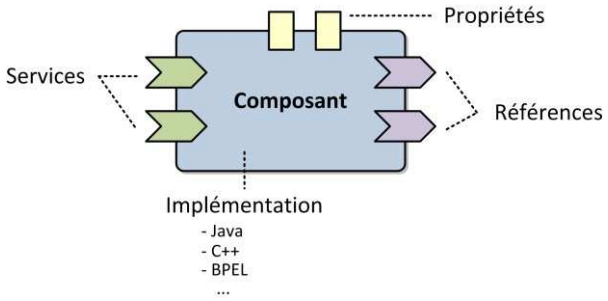


Figure 8. Un composant SCA

Un des avantages de SCA est qu'il supporte plusieurs technologies pour l'implémentation de services : Java, BPEL[23], C++. Les composants ayant la même vue externe, ils suivent le même traitement, indépendamment de la technologie utilisée pour leur implémentation. Le modèle de composition défini par SCA (présenté dans la section 2.2.1 du Chapitre 4) permet ainsi d'assembler des composants SCA de nature hétérogène afin d'obtenir une fonctionnalité plus complexe à partir des services fournis par les différents composants impliqués.

1.3.2.2 iPOJO

La technologie iPOJO [27], acronyme de *injected Plain Old Java Object*, développée au sein de l'équipe Adèle du laboratoire LIG[24] et intégrée actuellement au projet Apache Felix[25], est une implémentation des principes de l'approche à composants à services. iPOJO est considéré comme le successeur de *Service Binder* [22].

En suivant les principes de l'approche à composants à services, iPOJO propose un modèle de développement pour implémenter les services sous la forme de composants, et un langage pour leur description. Il fournit une plate-forme pour l'exécution et la composition dynamique de services. Cette plate-forme est développée au-dessus de la technologie OSGi, comportant ainsi les mêmes caractéristiques : centrée sur Java, centralisée, support du dynamisme.

Dans le modèle iPOJO, une spécification de service est une description de fonctionnalités fournies et de propriétés (une interface Java). Un type de composant décrit les spécifications de services fournies et requises, ainsi que différentes informations sur le contenu du composant et sur les aspects non-fonctionnels (voir la Figure 9). Une fabrique est attachée à un type de composant afin de permettre la création d'instances de composant. Les instances de composant interagissent à l'exécution en suivant les principes de l'approche à services.

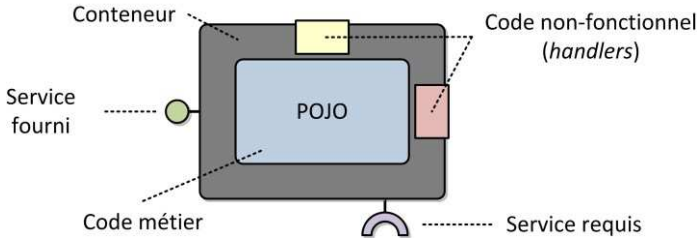


Figure 9. Un composant iPOJO

[23] *Business Process Execution Language*
[24] Laboratoire d'Informatique de Grenoble
[25] Apache Felix est une implémentation *open source* de la spécification OSGi

iPOJO utilise le concept de conteneur afin de séparer la logique métier d'un composant des aspects non-fonctionnels. Un conteneur iPOJO gère le cycle de vie d'une instance de composant ainsi que ses communications avec le monde extérieur. Un conteneur iPOJO est composé d'un ensemble de composants appelés *handlers*, chacun chargé de gérer un aspect non-fonctionnel. La plate-forme d'exécution iPOJO fournit un ensemble de *handlers* de base, par exemple, pour la gestion de dépendances de services (*Service Dependency Handler*), pour la gestion du cycle de vie d'instances de composants (*Lifecycle Callback Handler*), pour la reconfiguration d'instances de composants (*Configuration Handler*).

L'utilisation de conteneurs iPOJO simplifie la tâche des développeurs : un conteneur peut gérer tous les aspects liés à la technologie de services (publication, recherche, sélection, liaison). Ainsi, le développeur d'un composant iPOJO doit « seulement » implémenter la logique métier du composant (via un simple POJO) et configurer son conteneur. Cette facilité de développement constitue un des avantages majeurs d'iPOJO.

De plus, les conteneurs iPOJO sont extensibles : divers *handlers* peuvent être ajoutés aux conteneurs afin de gérer différents aspects non-fonctionnels tels que la sécurité ou les transactions. Cette capacité d'extensibilité est un autre avantage majeur du modèle iPOJO. Les concepts du modèle iPOJO sont résumés dans le Tableau 2.

	Concept	***Définition***
Développement	***Spécification de service***	*Une spécification de service est la description des fonctionnalités fournies et des propriétés d'un composant (une interface Java).*
	Type de composant	*Un type de composant (aussi appelé implémentation de service) implémente une ou plusieurs spécifications de services. Il peut avoir des dépendances vers d'autres spécifications.*
	Conteneur	*Un conteneur est composé par des composants nommés handlers, chargés de gérer des aspects non-fonctionnels. Les conteneurs sont extensibles.*
Exécution	***Instance de composant***	*Un composant peut posséder plusieurs instances qui peuvent être configurées différemment : chaque instance de composant (aussi appelée instance de service) possède une configuration qui lui est propre.*

Tableau 2. Concepts du modèle iPOJO

iPOJO définit des types de composants composites qui permettent de structurer d'autres types de composants, atomiques ou composites (cf. Chapitre 4, section 2.2.2).

1.3.2.3 SAM

La technologie SAM [28], acronyme de *Service Abstract Machine*, développée au sein de l'équipe Adèle du laboratoire LIG, propose un métamodèle pour la spécification, l'implémentation et l'exécution de composants à services, et une plate-forme d'exécution associée.

Le concept de service correspond au concept central du métamodèle SAM (voir la Figure 10). Un service possède trois matérialisations : spécification, implémentation et instance. Lors de la phase de spécification, un service est la spécification abstraite d'un ensemble de fonctionnalités logicielles. Une spécification peut indiquer des propriétés du service spécifié et des dépendances vers d'autres spécifications. Au développement, un service est une implémentation qui réalise les fonctionnalités définies par une spécification, hérite des propriétés et des dépendances définies, et peut définir des interfaces fournies, des propriétés et des dépendances propres. A l'exécution, un service est une instance d'une implémentation (voir le Tableau 3).

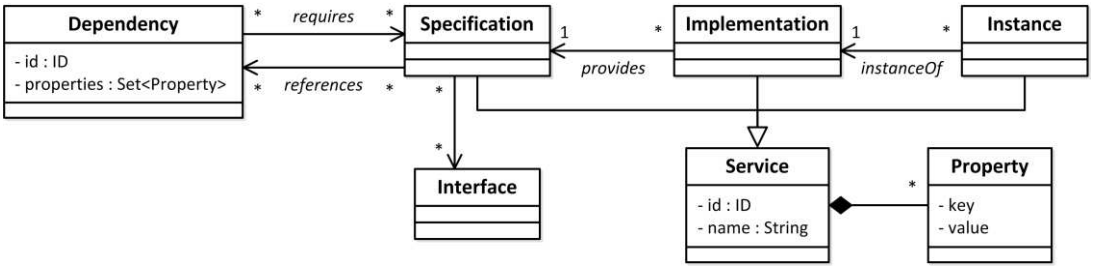


Figure 10. Métamodèle de services SAM [28]

	Concept	Définition
Développement	**Spécification**	*Une spécification définit les interfaces fournies, des propriétés et des dépendances vers d'autres spécifications. Elle est indépendante d'une technologie particulière.*
	Implémentation	*Une implémentation est une entité qui fournit une spécification en réalisant les fonctionnalités définies dans les interfaces associées. Elle hérite des propriétés et des dépendances définies par sa spécification, et peut définir en plus des interfaces fournies, des propriétés et des dépendances vers d'autres spécifications.*
	Propriété	*Un service dispose d'un ensemble de propriétés appelées des propriétés intrinsèques de service.*
Exécution	**Instance**	*Une instance représente l'exécution d'une implémentation de service. Elle hérite des propriétés et des relations définies par son implémentation, et par transitivité, celles de sa spécification.*

Tableau 3. Concepts du modèle SAM

L'approche SAM propose une plate-forme d'exécution, appelée SAM-RT, qui fournit les mécanismes propres à l'approche à services (publication, recherche, sélection, liaison). SAM-RT est définie comme une machine abstraite à services car elle ne fournit pas directement les services qu'elle expose. En effet, l'exécution de services est réalisée par des plates-formes à services hétérogènes utilisées par SAM-RT au travers de machines concrètes appelées *Service Concrete Machines.* SAM-RT permet ainsi l'interopérabilité de services hétérogènes.

SAM-RT fournit un modèle qui représente de manière homogène l'état d'exécution des services disponibles dans les plates-formes sous-jacentes (en termes de spécifications, d'implémentations et d'instances). Ce modèle est un modèle descriptif causalement lié au système représenté : toute modification du modèle engendre une modification de l'architecture en cours d'exécution, et toute modification de cette architecture provoque une modification du modèle. SAM-RT fournit des opérations de manipulation de spécifications, implémentations et instances permettant de manipuler l'état d'exécution des services.

SAM-RT supporte la distribution de services sur plusieurs machines SAM-RT distantes qui communiquent entre elles. SAM-RT gère de façon transparente la distribution des services, ce qui constitue l'un de ses avantages. SAM-RT supporte ainsi l'interopérabilité de services distribués. De plus, SAM-RT est extensible permettant l'intégration de nouveaux concepts et fonctionnalités.

L'approche SAM ne définit pas de modèle de composition de services. Toutefois, le métamodèle et l'environnement d'exécution étant extensibles, des concepts liés au concept d'application ainsi que des fonctionnalités pour la gestion d'applications peuvent y être intégrés. Les travaux de cette thèse ont comme point de départ l'approche SAM, dans le but de fournir un modèle pour la conception, le développement, l'exécution et le contrôle d'applications à services dynamiques.

1.3.3 AVANTAGES ET LIMITATIONS DE L'APPROCHE A COMPOSANTS A SERVICES

L'approche à composants à services combine les principes de l'approche à services et de l'approche à composants dans le but de faciliter le développement d'applications à services dynamiques. Elle propose ainsi d'utiliser des composants pour implémenter des services.

L'approche propose de décrire les compositions en termes de spécifications de services. La composition est centrée spécification, permettant ainsi de retarder la sélection d'implémentations de services jusqu'à la phase d'exécution, et aussi la substitution dynamique d'implémentations de services. De ce fait, l'approche fournit un plus grand degré de flexibilité pour la composition d'applications dynamiques. Cependant, les modèles à composants à services actuels n'implémentent pas tous les principes de l'approche, limitant la construction et/ou l'administration des applications.

1.4 SYNTHESE

Les approches à composants, à services et à composants à services proposent d'assembler des entités logicielles préexistantes dans le but de faciliter et de réduire les temps et les coûts de développement des applications. Toutefois, ces approches ne permettent pas de créer des applications dynamiques aisément ni de les administrer.

Le Tableau 4 résume les avantages et les limites de ces approches, exposés tout au long des sections précédentes.

	Avantages	***Limitations***
Approche à composants	*Modèle de développement* *Composition structurelle* *Reconfiguration*	*Manque de flexibilité* *Dynamisme difficile*
Approche à services	*Faible couplage* *Liaison tardive* *Dynamisme*	*Pas de modèle de composition structurelle* *Administration difficile*
Approche à composants à services	*Modèle de développement* *Composition structurelle* *Reconfiguration* *Faible couplage* *Liaison tardive* *Dynamisme* *Substitution*	*Modèles actuels :* *mécanismes de composition et d'administration limités/inexistantes*

Tableau 4. Avantages et limitations des approches actuelles pour la création d'applications

Cette thèse vise à fournir un modèle à composants à services pour la conception, le développement, l'exécution et l'administration d'applications à services flexibles et dynamiques. Un tel modèle combine ainsi les concepts de l'approche à composants et de l'approche à services avec les concepts des architectures logicielles dans le but de proposer un modèle de développement simple pour les développeurs et un modèle de composition intuitif pour les architectes des applications. Dans la suite, nous présentons les concepts des architectures logicielles.

2 ARCHITECTURES LOGICIELLES

Une architecture logicielle décrit la structure globale d'un système en termes d'entités logicielles et de connecteurs entre elles. Son but est d'augmenter le niveau d'abstraction du système lors de sa conception, et de faciliter ainsi sa compréhension, son développement, son évolution [29]. La description d'une architecture logicielle a ainsi une importance particulière.

Dans cette section, nous présentons tout d'abord les concepts de base des architectures logicielles. Ensuite, nous présentons la notion d'architecture logicielle pour les approches présentées dans la section précédente : architectures à composants, à services, et à composants à services. Nous soulignons l'importance de structurer une architecture logicielle sur différentes dimensions centrées sur des préoccupations particulières, et nous présentons ainsi les vues et les niveaux architecturaux que nous avons identifiés. Nous exposons ensuite les besoins et les défis auxquels sont confrontées les architectures logicielles. Enfin, nous présentons succinctement des langages et des outils existants, en les confrontant avec les besoins et les défis identifiés.

2.1 CONCEPTS DE BASE

De nombreuses définitions ont été proposées, tant dans le monde académique qu'industriel, afin de caractériser le concept d'architecture logicielle[26]. Shaw et Garlan définissent une architecture logicielle comme la description des éléments constituant un système logiciel, de leurs interactions, des patrons de composition utilisés, ainsi que des contraintes associées :

> « *Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.* » [30]

Une autre définition intéressante, citée fréquemment dans la littérature, est celle proposée par Bass, Clements et Kazman :

> « *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.* » [31]

Ici, l'architecture logicielle est définie comme une structure de haut niveau constituée d'un ensemble d'éléments logiciels et de relations entre eux. Cette définition insiste sur les propriétés extérieurement visibles des composants constituant le système. En effet, ces propriétés font partie de l'architecture car elles permettent d'établir les interactions entre les composants. Cette définition souligne aussi qu'un système peut avoir plusieurs structures, chacune correspondant par exemple à un point de vue particulier du système.

Le standard *IEEE 1471-2000* pour la description d'architectures propose une définition d'architecture qui ajoute la notion d'évolution :

> « *Architecture is defined by the recommended practice as the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.* » [32]

Taylor, Medvidovic et Dashofy définissent une architecture logicielle comme l'ensemble de décisions capitales prises lors du processus de conception du logiciel :

[26] Le site Web **http://www.sei.cmu.edu/architecture/start/community.cfm** de *The Software Engineering Institute* énumère plus de 90 définitions du concept d'architecture logicielle recueillies auprès de la littérature et des professionnels.

> « *A software system's architecture is the set of principal design decisions about the system.* » [33]

Les décisions de conception représentent l'architecture prescriptive d'un système. Une architecture prescriptive représente l'intention (ou le but) d'un système, alors qu'une architecture descriptive représente la matérialisation – la réalisation ou l'exécution – d'un système [33]. Une architecture, comme tout modèle, peut ainsi avoir deux rôles bien distincts : définir le système souhaité ou refléter le système concret.

Malgré les nombreuses définitions proposées, il n'existe pas de définition consensuelle du concept d'architecture logicielle. Néanmoins, les concepts de composant, connecteur et configuration sont généralement acceptés comme les concepts fondamentaux d'une architecture logicielle.

- un **composant** réalise une fonctionnalité spécifiée et peut être assemblé avec d'autres composants. Pour cela, un composant expose ses interfaces fournies et requises : la seule façon d'utiliser un composant est à travers ses interfaces fournies. La notion de composant prend différentes formes : type, implémentation, instance de composant.
- un **connecteur** modélise les interactions entre les composants d'une architecture. Un connecteur contient les informations concernant les règles d'interconnexion entre les composants : protocole de communication, spécification des échanges de données.
- une **configuration** est formée par de composants connectés à partir de leurs interfaces. Une configuration décrit ainsi l'ensemble de composants nécessaires pour le fonctionnement d'un système, ainsi que leurs interactions. Les façons d'assembler des composants dans une configuration sont spécifiées par des règles de composition.

L'architecture logicielle joue un rôle prépondérant dans la phase de conception de logiciels. Toutefois, l'architecture logicielle est confrontée à des défis majeurs comme la description, l'évaluation et la conception d'architectures. De nombreux langages de description d'architecture (ADLs) ont apparus pour répondre à ces défis. Medvidovic et Taylor définissent un langage de description d'architecture comme :

> « *a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation. An ADL must support the building blocks of an architectural description.* » [34]

Un langage de description d'architecture est ainsi une notation formelle ou semi-formelle qui permet de décrire la structure de conception d'un système logiciel par assemblage de composants. Le but des ADLs est de faciliter l'analyse et la compréhension des architectures logicielles à partir de leur description, et de permettre leur évaluation.

Une description d'architecture réalisée à l'aide d'un ADL peut être utilisée pour produire du code pour une plate-forme d'exécution cible (approche générative), ou peut être interprétée à l'exécution (approche interprétée).

2.2 ARCHITECTURES A COMPOSANTS, A SERVICES, ET A COMPOSANTS A SERVICES

Dans l'approche à composants (cf. section 1.1), l'architecture d'un système est définie en termes de types de composant, de connecteurs, et de leurs configurations. Une configuration peut être utilisée comme un type de composant à l'intérieur d'autres configurations, résultant ainsi dans une configuration hiérarchique. La description de l'architecture d'un système à composants est utilisée pour créer (par génération ou interprétation) les instances de composants correspondantes et leurs connecteurs.

Dans l'approche à services (cf. section 1.2), le manque d'expression des dépendances entre services et l'absence de la notion d'encapsulation constituent des limitations majeures. En effet, les systèmes à services ne peuvent pas être conçus (par une architecture prescriptive) ni représentés

(architecture descriptive) de façon structurelle. Les services ne peuvent pas être structurés de manière hiérarchique selon leur granularité.

Dans l'approche à composants à services (cf. section 1.3), l'architecture d'un système est définie en termes de spécifications de composant, de connecteurs, et de leurs configurations. Comme dans l'approche à composants traditionnelle, les configurations peuvent être hiérarchiques. Une description est utilisée pour créer ou sélectionner à l'exécution les services correspondants et les lier.

Nous nous intéressons aux architectures permettant la conception, le développement, l'exécution et l'administration d'applications à composants à services. Toutefois, la construction de telles architectures s'avère difficile. En effet, leur conception et leur description sont confrontées aux mêmes défis de la production de logiciels : modularité, abstraction, séparation de préoccupations, réutilisation, composition incrémentale, évolution. La notion de vue architecturale, présentée ci-dessous, est un concept primordial qui vise à répondre aux défis de la construction d'architectures logicielles [35].

2.3 VUES ARCHITECTURALES

Il y a quatre décennies, Parnas [36] a souligné qu'un système logiciel est constitué de plusieurs structures, définies comme des descriptions partielles du logiciel. Un système est donc une collection de parties et de relations entre ces parties. Parnas a identifié différentes structures génériques présentes fréquemment dans les logiciels : structures de modules, structures d'utilisation, et structures de processus.

Plus tard, Perry et Wolf [37] ont reconnu le besoin d'avoir des vues diverses pour un système logiciel. Une vue est la représentation d'un ensemble d'éléments d'un système avec leurs relations. Chaque vue met l'accent sur certains aspects architecturaux et est utilisée pour une finalité différente, souvent par des acteurs différents.

Kruchten [38] a proposé plus tard un modèle (nommé « 4+1 ») qui identifie quatre vues principales d'une architecture logicielle qui peuvent être utilisées pour la construction de systèmes : vue logique, de processus, de développement et physique ; avec une cinquième vue qui ensemble les quatre dernières.

A peu près à la même époque, Soni, Nord et Hofmeister ont proposé un modèle, appelé *Siemens Four View*, qui distingue quatre vues d'une architecture logicielle : vue conceptuelle, vue d'interconnexion de modules, vue d'exécution et vue de code.

D'autres propositions d'ensembles de vues ont fait leur apparition [39] [40]. Toutefois, aucun ensemble de vues n'est approprié pour tous les systèmes logiciels. En effet, le standard *IEEE 1471-2000* pour la description d'architectures [32], préconise la création de vues spécifiques à un système logiciel qui assistent au mieux ses acteurs et aux préoccupations associées au système.

Dans ce travail, nous privilégions les vues associées aux différents aspects présents dans les différentes phases du cycle de vie des systèmes logiciels. Nous considérons ainsi qu'un système peut être représenté par une ou plusieurs structures architecturales (modèles), chacune présentant un point de vue (préoccupation) du système à un certain niveau d'abstraction (phase du cycle de vie). La notion d'abstraction d'un système est relative à un point de vue (voir la Figure 11). Pour un point de vue, métier par exemple, un modèle peut être transformé/raffiné vers un autre modèle de niveau d'abstraction différent : le développement d'un système peut ainsi être automatisé par une série de transformations de modèles ; plus un modèle est raffiné, plus sa précision augmente et sa granularité devient fine. Pour un niveau d'abstraction donné des modèles correspondants à des points de vue différents peuvent être composés assemblant ainsi des préoccupations différentes.

Afin de contrôler l'exécution d'une application, une partie des informations produites lors de sa conception et de son développement doivent être présentes à l'exécution : les informations constituant les modèles prescriptifs d'une application (ses buts) doivent ainsi être transmises d'activité en activité jusqu'à la phase d'exécution afin de pouvoir gérer son exécution (modèles descriptifs). De plus, cette continuité de la connaissance peut permettre d'effectuer à l'exécution un certain nombre d'activités

réalisées habituellement dans les phases de conception et de développement des applications : sélection de composants, évaluation de la cohérence et de la conformité de la composition, changements des buts, etc.

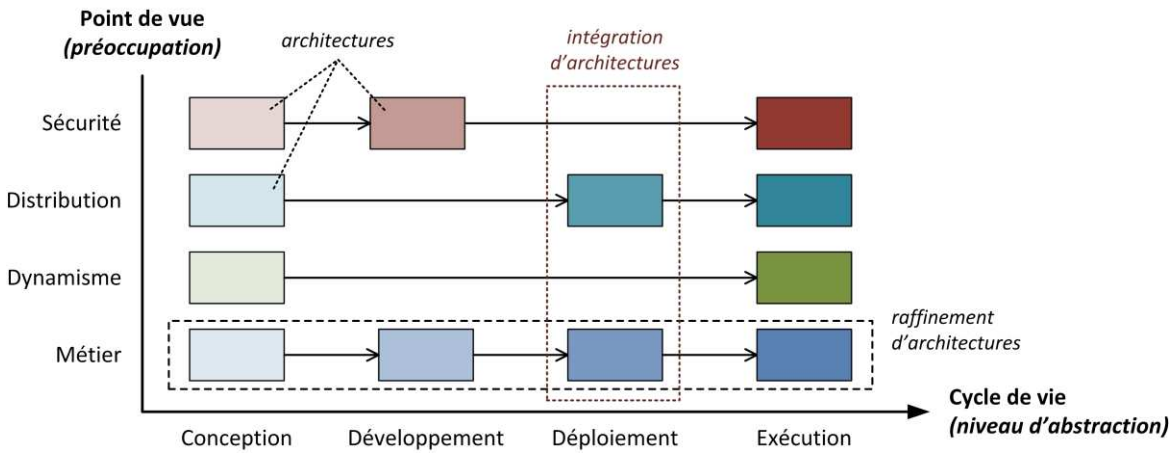


Figure 11. Architectures à différents niveaux d'abstraction pour des vues différentes

Dans ce travail, nous distinguons et caractérisons différentes architectures présentes dans le cycle de vie d'une application pour un point de vue particulier (métier par exemple). Nous identifions les similitudes et les différences entre ces architectures afin de déterminer l'information qui est ajoutée, modifiée, supprimée à chaque phase pour établir l'information qui doit être transmise à la phase d'exécution.

Dans la Figure 11, nous illustrons différentes architectures pour les différents points de vue ; chacune de ces architectures correspond à une phase du cycle de vie et donc un niveau d'abstraction différent : conception, développement, déploiement, exécution. Pour chaque niveau, le concept d'application raffine le concept d'application de base introduisant les concepts relatifs au niveau architectural. L'architecture métier d'une application au niveau conception est un objet qui contient des spécifications de composants et des relations entre elles. Une architecture au niveau exécution est un objet qui contient des spécifications, des implémentations et des instances de composants et les relations entre elles (voir la Figure 12).

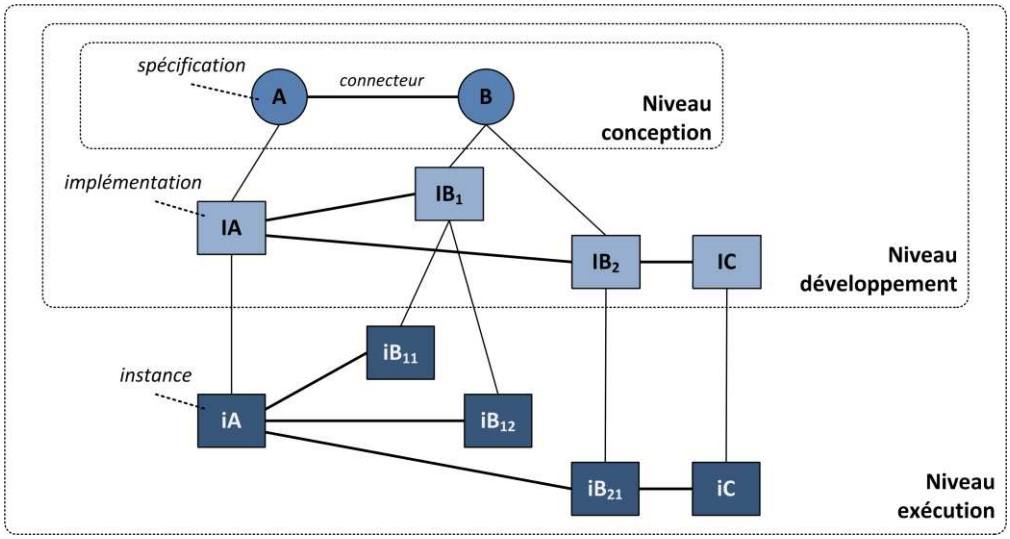


Figure 12. Architectures d'une application à différents niveaux

En effet, certains éléments de conception/développement (notamment les spécifications et les implémentations de composant) restent visibles comme éléments d'exécution. Autrement dit, quelques notions sont préservées à l'exécution en tant qu'entités identifiables. L'identification de ces éléments à l'exécution permet d'administrer l'exécution des applications.

2.4 BESOINS ET DEFIS

Les architectures logicielles sont confrontées à divers besoins et défis dans le but de satisfaire les besoins de conception, de développement, et de contrôle de l'exécution des applications. Nous identifions l'ensemble des besoins et défis suivants :

- **la spécification abstraite de l'architecture** en termes de composants logiciels (réutilisables) et de connecteurs entre eux. Les fonctionnalités des composants sont indépendantes des technologies de réalisation et des plates-formes d'exécution. De ce fait, la spécification de l'architecture doit être aussi abstraite – indépendante des technologies de réalisation des composants – afin de permettre aux architectes de manipuler les constructions essentielles de l'architecture.
- **la structuration de l'architecture dans différentes vues** afin de considérer différentes préoccupations, fonctionnelles et non-fonctionnelles, ainsi que les différentes phases du cycle de vie du logiciel, simplifiant ainsi la manipulation des architectures. Les différents acteurs (concepteurs, développeurs, architectes, administrateurs) d'une application manipulent des vues différentes en fonction de leurs rôles, de leurs préoccupations, de leurs connaissances.
- **la construction incrémentale de l'architecture** à partir de sa spécification, passant par sa réalisation concrète jusqu'à son exécution. Le processus de construction incrémentale d'architectures est une tâche complexe car la cohérence et la conformité des architectures doivent être assurées. De plus, lors du processus de construction d'une architecture, d'autres architectures correspondantes à des vues architecturales différentes peuvent être intégrées, assemblant ainsi des préoccupations différentes du système.
- **le support de l'évolution de l'architecture** afin de considérer les changements du système (dynamisme des composants, nouveaux besoins des utilisateurs). Une architecture de conception doit préciser la variabilité d'un système.
- **la validation de l'architecture** afin d'assurer sa cohérence et sa conformité suite à la construction incrémentale ou à l'évolution.

Ces besoins doivent être pris en compte lors de la construction des architectures logicielles, afin de supporter l'analyse, la validation, l'administration, l'évolution des applications. En outre, nous ressentons le besoin de fournir des supports logiciels pour faciliter ces tâches.

2.5 LANGAGES DE DESCRIPTION D'ARCHITECTURE EXISTANTS

De nombreux ADLs ont été conçus. Chacun se distingue par ses capacités de modélisation d'architectures qui proviennent directement des objectifs recherchés. Certains langages se veulent génériques, fournissant la flexibilité suffisante pour capturer tous les aspects potentiellement pertinents d'une application ; d'autres langages sont spécialisés tandis que d'autres, considérés comme langages noyau, sont extensibles permettant ainsi leur adaptation à de nouveaux besoins. Un cadre de classification et de comparaison de langages de description d'architectures et des outils associés a été publié dans [34].

Nous pouvons citer les langages ACME [41] qui se veut un ADL générique ; Rapide [42] qui permet de spécifier le comportement dynamique des applications et de réaliser des simulations afin de valider les architectures, ou Wright [43] qui supporte la spécification formelle d'architectures, la description de reconfigurations dynamiques, et la vérification d'interactions entre composants.

Dans [44], les langages de description d'architectures sont différenciés des langages de configuration. Les langages de configuration fournissent un modèle où un composant est considéré comme une entité instanciable : la configuration est décrite en termes d'instances de composants. Des exemples de ces langages sont Darwin [45] qui permet d'effectuer des instanciations (de façon paresseuse et dynamique) à partir d'une description afin de former une architecture spécifique exécutable ; OCL (*Olan Configuration Language*) [46], inspiré de Darwin, qui repose sur un modèle d'assemblage de composants avec les concepts de connecteur et de collection d'instances de composants ; Fractal ADL [47], un langage ouvert et extensible, qui permet la définition d'architectures à partir de composants Fractal ; le langage fournit par iPOJO [27] qui permet la composition structurelle d'applications en termes de composants à services.

En particulier, nous nous intéressons aux langages de description d'architectures qui permettent de générer une image exécutable de l'application. Nous identifions certains critères, issus des besoins et des défis cités précédemment, que nous jugeons pertinents pour évaluer les capacités des langages de description d'architecture :

- **vues multiples** : permet la spécification d'architectures logicielles à travers diverses structures, chacune satisfaisant les besoins et les préoccupations de différents acteurs (packaging, déploiement, distribution, sécurité). La cohérence entre les vues doit être assurée.
- **construction hiérarchique** : permet la spécification d'architectures à différents niveaux de granularité : une architecture peut être un composant d'une autre architecture.
- **construction incrémentale** : permet la construction valide de systèmes exécutables au travers du raffinement (et de l'intégration) des architectures.
- **validation** : supporte la validation de la syntaxe et de la sémantique des architectures.
- **vérification de la cohérence et la conformité** : permet la validation (en permanence) des propriétés du système afin d'assurer les propriétés souhaitées.
- **instanciation** : permet la spécification de propriétés d'instanciation (paresseuse, dynamique) des composants du système.
- **évolution** : supporte l'évolution de l'architecture au niveau de propriétés, composants, connecteurs : modification, ajout, suppression.
- **dynamisme** : supporte la spécification du dynamisme au niveau de composants et connecteurs, afin de permettre la reconfiguration des systèmes à l'exécution.

Le Tableau 5 positionne certains langages de description d'architecture par rapport aux critères énoncés précédemment.

	Vues Multiples	***Construction hiérarchique***	***Construction incrémentale***	***Validation : Syntaxe & Sémantique***	***Vérification : Cohérence & Conformité***	***Instanciation***	***Evolution***	***Dynamisme***
Darwin	x	✓	✓	x	x	✓	x	✓
OCL	x	✓	✓	✓	x	✓	x	✓
Fractal ADL	x	✓	✓	✓	✓	✓	x	✓
iPOJO	x	✓	✓	✓	x	✓	x	x

Tableau 5. Positionnement de langages ADL par rapport aux critères

2.6 SYNTHESE

Les architectures logicielles, prescriptives et descriptives, constituent des informations essentielles pour la conception, le développement, l'exécution et la gestion des applications. Cependant, en général, la description de l'architecture prescriptive d'une application n'est pas manipulable à l'exécution : il existe un véritable écart entre la conception et l'exécution des applications.

Notre travail vise à modéliser les différentes architectures présentes dans le cycle de vie des applications : conception, développement, déploiement, exécution ; pour des vues architecturales différentes : métier, distribution, sécurité, etc. ; et à réduire ainsi l'écart existant entre la conception et l'exécution des applications.

Dans la section suivante, nous présentons l'Ingénierie Dirigée par les Modèles (IDM) et les modèles à l'exécution (*models@run.time*), deux paradigmes qui proposent des cadres pour la conception, le développement, l'exécution d'applications ; permettant de réduire l'écart entre l'espace du problème et l'espace de solution.

3 INGENIERIE DIRIGEE PAR LES MODELES

L'*Object Managment Group* (OMG) a proposé une approche nommée *Model Driven Architecture* (MDA) qui a pour objectif la modélisation d'applications séparant les aspects indépendants des plates-formes d'exécution (PIM[27]) et ceux qui sont dépendants de plates-formes spécifiques (PSM[28]). Cette séparation rend indépendante la spécification de l'application, facilitant la génération du code de l'application vers une technologie cible particulière au travers d'une succession de transformations de modèles.

L'Ingénierie Dirigée par les Modèles (IDM) reprend les concepts et les principes de l'approche MDA dans l'intention de faire face à la complexité, l'hétérogénéité et l'évolution rapide des applications. L'IDM propose ainsi de concevoir des applications logicielles à partir d'un ensemble de modèles (représentant des vues différentes) afin de générer automatiquement des artefacts exécutables, réduisant ainsi le temps de développement des applications. L'IDM utilise un ensemble de technologies qui supportent des transformations de modèles qui vont du niveau d'abstraction du problème jusqu'à l'implémentation logicielle de la solution. De ce fait, l'approche de l'IDM permet de réduire l'écart entre le problème et l'implémentation logicielle de la solution.

Des travaux récents, comme [3], [48] et [49], présentent une vision plus ambitieuse de l'IDM. Leur objectif est d'étendre l'utilisation des modèles produits dans la phase de développement à l'exécution. L'utilisation de modèles à l'exécution (*models@run.time*) permet de raisonner sur les applications dans leur contexte d'exécution. Les modèles à l'exécution fournissent donc un support pour la gestion de l'exécution des applications.

Dans cette section nous présentons les concepts de base de l'approche IDM : modèle, métamodèle, langage. Nous présentons l'importance de l'abstraction et de la séparation de préoccupations dans la méta-modélisation. Nous présentons ensuite le principe de composition de modèles. Enfin, nous présentons les principes des modèles à l'exécution.

[27] *Platform Independent Model*
[28] *Platform Specific Model*

3.1 CONCEPTS DE BASE

Le concept de modèle est le concept principal de l'IDM. Plusieurs définitions de modèle ont été proposées, cependant, il n'existe pas une définition consensuelle de ce concept. Nous nous appuyons sur les définitions suivantes pour expliquer le terme de modèle :

> « *Un modèle est une abstraction… une représentation… une simplification d'un système étudié.* » [50]
>
> « *A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.* » [51]
>
> « *A model is a description or specification of that system and its environment for some certain purpose.* » [52]
>
> « *A model is an abstraction of some aspect of a system. The system described by a model may or may not exist at the time the model is created. Models are created to serve particular purposes.* » [3]

Selon ces définitions, **un modèle est une représentation simplifiée d'un système**. Une telle représentation peut être la spécification d'un système à développer ou la description d'un système existant. Un modèle est une abstraction d'un aspect d'un système : un système peut être représenté par un ou plusieurs modèles, chacun représentant un aspect du système (une vue). Un modèle, étant la description d'un système, doit pouvoir répondre à certaines questions, à la place du système.

Nous constatons d'après ces définitions que les modèles répondent aux trois critères proposés par Ludewing [53] :

- le **critère de représentation** (*mapping criterion*) : il existe un objet ou phénomène original représenté par le modèle.
- le **critère de réduction** (*reduction criterion*) : toutes les propriétés de l'objet original ne sont pas représentées dans le modèle (le modèle est réduit). Le modèle représente au moins quelques propriétés de l'objet original.
- le **critère de pragmatisme** (*pragmatic criterion*) : le modèle peut remplacer l'objet original pour un certain usage (le modèle est utilisable).

A partir de ces critères, deux aspects fondamentaux de la modélisation de systèmes logiciels peuvent être dérivés : **l'abstraction** et **la séparation de préoccupations**. Le principe d'abstraction ne considère que les informations d'un système jugées pertinentes à un moment donné, en omettant les autres informations. La séparation de préoccupations est la séparation des différents aspects d'un système : chaque aspect du système est pris en compte par un modèle particulier [50]. La séparation de préoccupations doit être associée à un mécanisme d'intégration (ou composition) de préoccupations afin d'avoir une vision globale du système. Un système peut donc être représenté par plusieurs modèles, chacun représentant une préoccupation du système à un certain niveau d'abstraction.

D'après les définitions citées auparavant, nous constatons qu'il existe divers « types » de modèles. En effet, les modèles peuvent être distingués en fonction de la relation existante entre le modèle et le système modélisé. Dans [52], [53] et [54], le **modèle descriptif** d'un système est distingué du **modèle prescriptif**. Un modèle descriptif décrit de façon abstraite un système ou un aspect particulier d'un système (voir la Figure 13). Un modèle prescriptif spécifie les caractéristiques attendues d'un système (voir la Figure 14). Ainsi, un modèle prescriptif permet de définir des spécifications afin de réaliser un système, alors qu'un modèle descriptif est construit à partir d'un système existant. Cette distinction est importante car elle permet d'établir les relations de correction et de validité entre un modèle et le système modélisé : un modèle descriptif est *correct* si toutes les assertions spécifiées par le modèle sur

le système sont effectivement vraies ; un système est *valide* par rapport à son modèle prescriptif si aucune assertion définie par le modèle n'est fausse pour le système.

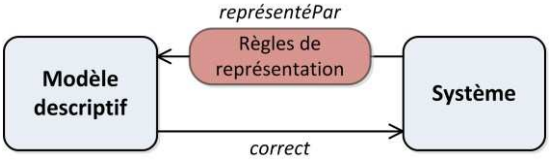


Figure 13. Modèle descriptif

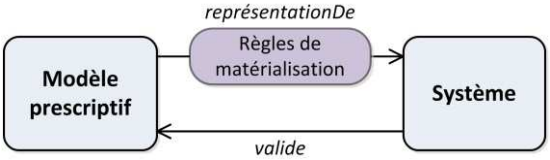


Figure 14. Modèle prescriptif

En outre, les modèles peuvent être catégorisés en fonction des différentes phases du cycle de vie d'un logiciel. Dans [55], Fowler propose trois niveaux de modèles : modèles conceptuels, modèles de spécification et modèles d'implémentation. Dans [3], France et Rumpe classifient les modèles en modèles de développement et modèles d'exécution (cf. section 3.5).

Indépendamment de leur type (descriptif, prescriptif) ou de la phase du cycle de vie qu'ils représentent (conception, développement, exécution), les modèles doivent être formalisés dans un langage de modélisation bien défini. Afin de définir un tel langage, il est nécessaire de spécifier tous ses aspects : le langage devient ainsi un sujet de modélisation. Le modèle qui spécifie le langage de modélisation est appelé **métamodèle**.

« *A metamodel is a model that defines the language for expressing a model.* » [52]

Un métamodèle définit donc une grammaire et un vocabulaire afin de réaliser des modèles conformes. La relation entre un modèle et son métamodèle est une relation de conformité. Un modèle est conforme à son métamodèle si les concepts des éléments du modèle et des relations qui les relient sont définis dans le métamodèle, et si les contraintes du métamodèle sont respectées par le modèle. La Figure 15 décrit les relations entre les concepts de langage, métamodèle, modèle et système modélisé.

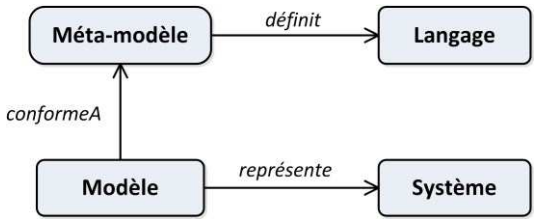


Figure 15. Relations entre les concepts de l'IDM

La modélisation de systèmes est confrontée à plusieurs défis : **la définition des langages** (la méta-modélisation) qui permettent de décrire les modèles d'une application ; **la séparation des préoccupations** d'une application en différents modèles ; et **le traitement de modèles** (composition, transformation, manipulation).

3.2 META-MODELISATION

Un métamodèle permet la description des modèles qui représentent un système. La définition de métamodèles, ou méta-modélisation, consiste à définir explicitement et formellement un langage de modélisation : les concepts qui composent les modèles et les relations entre ces concepts (la syntaxe abstraite du langage), la notation utilisé pour la représentation des modèles (la syntaxe concrète du langage) et le sens des modèles (la sémantique du langage). La représentation explicite et formelle de métamodèles permet de produire de façon fiable, voire automatisée, les mécanismes nécessaires pour le traitement de modèles. Cette représentation permet ainsi de créer des outils pour l'édition, la vérification, la composition et la transformation automatisées de modèles.

Parmi les approches de méta-modélisation nous pouvons citer MOF[29] et EMF[30]. MOF est le formalisme pour la spécification de métamodèles dans MDA. MOF est situé au sommet d'une architecture de méta-modélisation en quatre couches : la couche de base M0 représente les objets du monde réel ; la couche M1 contient les modèles du système modélisé ; la couche M2 contient les métamodèles utilisés pour la définition des modèles de la couche M1 ; et la couche M3 contient le méta-métamodèle MOF qui est utilisé pour décrire tous les métamodèles utilisés dans M2 (ce méta-métamodèle est auto-descriptif). Le métamodèle du langage UML[31] a été défini en utilisant MOF.

EMF est un canevas de modélisation et une infrastructure de génération de code pour la construction d'outils et d'applications. EMF est constitué d'un métalangage orienté objets pour la spécification de métamodèles : *Ecore,* d'une API pour le traitement de ces métamodèles et de leurs modèles ; et des outils pour la génération de code (Java) à partir de la définition de métamodèles. EMF a une ample utilisation grâce à deux raisons principales : il met à disposition des outils (contrairement à MOF qui est juste une spécification) et il a été construit au-dessus de l'environnement Eclipse.

D'autres approches de méta-modélisation, Kermeta[32] par exemple, ajoutent explicitement la sémantique opérationnelle dans les métamodèles. Ces approches de méta-modélisation sont utiles pour la création de langages dédiés exécutables, car la définition explicite de la sémantique opérationnelle permet de simuler l'exécution et de tester les langages au moment de la conception. Toutefois, ces approches doivent inclure des machines d'exécution au niveau métamodèle capables d'interpréter la définition de la sémantique, rentrant donc dans les problèmes de définition de la sémantique d'exécution.

3.3 SEPARATION DE PREOCCUPATIONS

Le principe de la séparation de préoccupations vise à réaliser une décomposition (et puis une composition) des différentes préoccupations logicielles qui composent une application. Dans l'approche MDA, la préoccupation métier (modelée par des PIMs) est séparée de la préoccupation technologique (modelée par des PSMs). Dans l'IDM, cette séparation peut s'appliquer à tous les aspects particuliers d'une application : chacun des aspects particuliers d'une application est pris en compte par un modèle spécifique [50].

Un système peut donc être représenté par un ou plusieurs modèles, chacun représentant une préoccupation du système à un certain niveau d'abstraction. La relation de représentation entre un modèle et le système modélisé est donc relative à un point de vue. Les différents modèles d'un même système peuvent être produits et traités à la fois en fonction des différentes perspectives et des acteurs impliqués.

Dans le cas des modèles descriptifs, les différents aspects du système modélisé sont liés (composés) : la composition de vues différentes est toujours envisageable. Dans le cas des modèles prescriptifs, la construction du système consiste à effectuer la composition de tous les aspects des modèles : la cohérence des modèles doit être assurée.

Même si la séparation des préoccupations est une caractéristique de l'ingénierie dirigée par les modèles, elle représente un des défis de la modélisation de systèmes : les aspects utilisés pour la décomposition d'une application doivent être identifiés ; chacun de ces aspects peut être modélisé en utilisant un langage de modélisation (métamodèle) différent ; la façon de composer les modèles doit être spécifiée.

[29] *Méta-Object Facility*
[30] *Eclipse Modeling Framework*
[31] *Unified Modeling Language*
[32] **http://www.kermeta.org/**

3.4 COMPOSITION DE MODELES

Dans la section précédente, nous avons montré qu'un système peut être représenté par plusieurs modèles, chacun représentant un aspect différent du système (cf. section 2.3). La composition de modèles permet ainsi d'intégrer des aspects différents du système. La **composition de modèles** est définie par Herrmann et al. comme :

> « *Model composition in its simplest form refers to the mechanism of combining two models into a new one.* » [56]

La composition de modèles est un mécanisme qui permet de combiner deux (ou plus) modèles dans un modèle nouveau. Dans le cas où les modèles à composer ainsi que le modèle composé ont le même métamodèle, la composition, appelée composition endogène, peut se réaliser au niveau modèle : des éléments du même type sont composés ; des mises en correspondance sont nécessaires au niveau modèle. Dans le cas où les modèles à composer et le modèle composé ont des métamodèles différents, la composition, appelée composition exogène, est définie au niveau des métamodèles : des relations entre les éléments des métamodèles sont établies. La définition de ces relations n'exprime pas la sémantique de la composition : des règles de traitement des éléments des modèles doivent donc être définies. La composition exogène applique des opérateurs de composition au niveau des éléments des métamodèles.

La composition de modèles peut donc être réalisée par application d'opérateurs ou par création de relations. Dans le premier cas, la composition est réalisée en appliquant des opérateurs de composition – fusion, remplacement, union, tissage –, définis au préalable, sur les modèles à composer. Ces modèles ainsi que les opérateurs définis ne font pas partie du modèle composite résultant. Dans le deuxième cas, des relations – association, agrégation, héritage – sont établies entre les éléments des modèles à composer. Ces modèles ainsi que les relations construites entre eux font partie du modèle composé résultant.

La composition des modèles d'un système réalisée au développement permet de composer un modèle intégrant des aspects différents du système à développer. La composition de modèles peut aussi être réalisée à l'exécution, fournissant ainsi un cadre de haut niveau pour la création et l'administration d'applications de plus en plus dynamiques.

3.5 MODELES A L'EXECUTION : MODELS@RUN.TIME

Les nouveaux types d'applications tels que les applications ubiquitaires, embarqués et mobiles, sont de plus en plus dynamiques. Ils s'exécutent dans des contextes évolutifs et non déterministes, leur imposant ainsi de s'adapter. Pour ce faire, l'exécution des applications doit être suivie afin de surveiller les systèmes et de les adapter dynamiquement aux évolutions de leur contexte.

La recherche sur les systèmes auto-adaptatifs a produit des résultats significatifs pour l'adaptation d'applications, cependant les problèmes à affronter restent encore nombreux. Un problème particulièrement difficile est la gestion de la complexité issue de la quantité d'information qui peut être associée à des phénomènes à l'exécution. Une approche prometteuse pour gérer cette complexité est de développer des mécanismes d'adaptation qui utilisent des modèles à l'exécution, nommés *models@run.time*. Le but des modèles à l'exécution est d'étendre à l'exécution l'utilisation de modèles produits lors de la conception des systèmes.

Blair, Bencomo et France proposent la définition suivante d'un modèle à l'exécution :

> « *A model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective.* » [48]

Un modèle à l'exécution fournit une représentation (en termes de structure, comportement, ou objectifs) d'un système en exécution. Le **modèle est connecté causalement au système**, fournissant

ainsi des informations à jour du système, et permettant de réaliser des adaptations par des modifications du modèle.

Un modèle à l'exécution sert ainsi de base pour **la surveillance, l'analyse et l'adaptation d'un système en exécution**. Grâce à leur connexion causale, les modèles à l'exécution décrivent des systèmes et, en même temps, spécifient comment les systèmes doivent se comporter : les modèles à l'exécution sont descriptifs et prescriptifs.

Les modèles à l'exécution évoluent dans le temps. Les modifications des modèles peuvent être effectuées de façons différentes : par des transformations, par des opérations prédéfinies, par des outils spécialisés. Selon que la partie prescriptive ou descriptive d'un modèle est modifiée, les conséquences sont différentes. Les modifications des éléments prescriptifs d'un modèle à l'exécution provoquent des changements (adaptations/reconfigurations) dans le système. Les modifications dans les éléments descriptifs d'un modèle sont occasionnées par le système : lors d'un changement dans le système, sa représentation dans le modèle change aussi ; et doivent donc être valides vis-à-vis du modèle prescriptif.

Les métamodèles des modèles à l'exécution doivent ainsi fournir des mécanismes de construction de modèles permettant la définition de :

- **la partie prescriptive du modèle** spécifiant comment le système doit être,
- **la partie descriptive du modèle** spécifiant l'état du système à l'exécution,
- **les modifications valides de la partie prescriptive du modèle** (des adaptations) exécutables à l'exécution,
- **les modifications valides de la partie descriptive du modèle** exécutables à l'exécution, et
- **la connexion causale** sous forme de flux d'information entre le modèle et le système.

Dans [57], Lehmann et al. proposent un processus de méta-modélisation de modèles à l'exécution qui répond aux besoins ci-dessus. La Figure 16 présente les concepts derrière ce processus sous la forme d'un **méta-métamodèle**. Ce méta-métamodèle fournit les mécanismes nécessaires pour la formalisation de métamodèles de modèles à l'exécution.

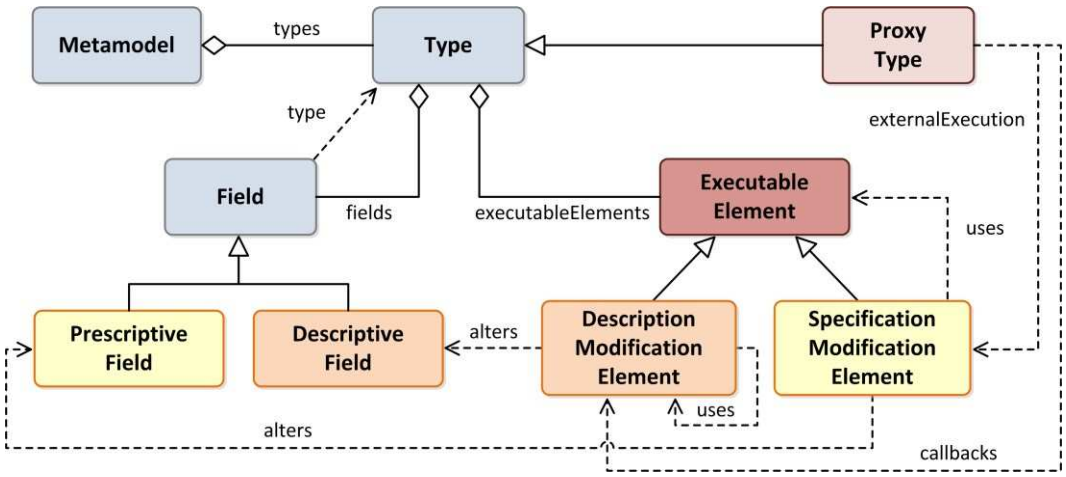


Figure 16. Méta-métamodèle de modèles à l'exécution [57]

D'après ce méta-métamodèle, chaque métamodèle conforme définit des types (*Types*) composés de champs (*Fields*) et d'éléments exécutables (*ExecutableElements*). Les champs représentent des relations entre les types et sont classifiés comme prescriptifs (*PrescriptiveField*) ou descriptifs (*DescriptiveField*). Intuitivement, les éléments d'un modèle contenu dans les champs prescriptifs sont des éléments prescriptifs, ceux qui sont contenus dans les champs descriptifs sont des éléments

descriptifs. La distinction de champs permet d'identifier ainsi les parties descriptives et prescriptives des modèles lors du processus de méta-modélisation.

Les éléments exécutables (*ExecutableElements*) représentent des opérations permettant la modification des éléments d'un modèle. Les éléments exécutables sont classifiés comme des éléments de modification de la partie descriptive (*DescriptionModificationElements* – DME) ou des éléments de modification de la partie prescriptive (*SpecificationModificationElements* – SME). Les DMEs encapsulent les mécanismes de synchronisation d'état des modèles, les SMEs représentent des adaptations possibles (reconfigurations) du modèle et du système.

Les éléments descriptifs d'un modèle sont contenus dans les champs descriptifs. Ainsi, chaque DME définit les champs descriptifs qu'il modifie, en utilisant la relation d'association *alters*. Un DME peut être associé à d'autres DMEs, à travers la relation *uses*, afin d'exprimer que son exécution implique l'exécution des DMEs associés.

L'adaptation d'un modèle peut influencer sa partie prescriptive, et en conséquence son état. Ainsi, les SMEs peuvent définir des relations *alters* et *uses* avec des champs prescriptifs et des éléments exécutables respectivement.

Enfin, le proxy (*Proxy type*) permet la formalisation de la connexion causale des modèles à l'exécution avec le système modélisé. Un proxy classifie les éléments d'un modèle afin de le connecter avec son système associé. A l'exécution, le proxy interagit avec un élément externe au travers d'une interface de communication bien définie. Cette interface est définie sous la forme d'élément exécutable, et est donc appelée lors de l'adaptation du modèle afin de modifier le système (*externalExecution*), ou disponible aux proxys pour envoyer des informations du système en exécution au modèle (*callbacks*).

D'après ce processus de méta-modélisation de modèles à l'exécution, nous pouvons constater que différents modèles, descriptifs et prescriptifs, peuvent représenter un même système ou des aspects différents d'un système en exécution.

Dans la section précédente, nous avons évoqué qu'il est difficile de développer un système logiciel complexe à partir d'un seul modèle : différents aspects du système doivent être considérés. De manière similaire, l'administration de l'exécution d'un système ne peut pas être couverte par un seul modèle à l'exécution. En effet, Blair et al. ont déclaré que :

> « *in practice, it is likely that multiple [runtime] models will coexist and that different styles of models may be required to capture different system concerns.* » [48]

A cet égard, dans [58], Vogel, Seibel et Giese proposent une catégorisation de différents types de modèles à l'exécution et présentent les relations entre eux (voir la Figure 17). Ces types de modèles, classifiés par rapport à leur usage et aux éléments qu'ils représentent, sont :

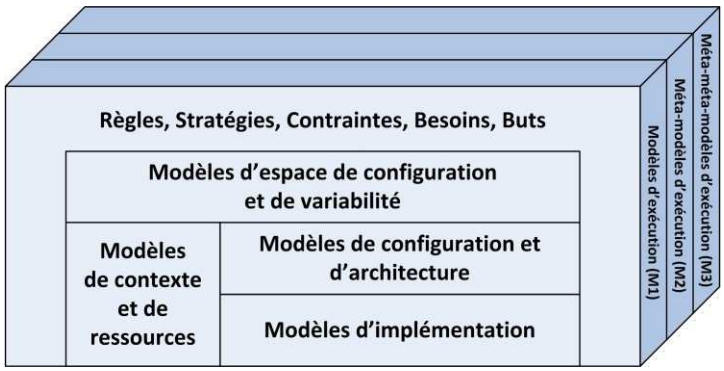


Figure 17. Types de modèles à l'exécution [58]

- **les modèles d'implémentation** sont similaires aux modèles utilisés dans le domaine de la réflexion pour représenter et modifier un système en exécution par une connexion causale. Ces modèles sont ainsi dynamiques car ils évoluent systématiquement avec un système en exécution. Tels modèles sont basés sur l'espace de solution d'un système : ils sont couplés à l'implémentation du système et au modèle computationnel.
- **les modèles de configuration et d'architecture** sont à un niveau supérieur des modèles d'implémentation, mais ils fournissent aussi une représentation causalement connectée du système. Ces modèles reflètent la configuration et l'architecture courantes d'un système. Ils sont la base pour la supervision et l'adaptation du système.
- **les modèles de contexte et de ressources** décrivent l'environnement opérationnel d'un système en exécution. Cela contient le contexte d'un système : les informations qui caractérisent une entité – personne, emplacement ou objet – considérée pertinente pour l'interaction entre un utilisateur et une application ou pour le fonctionnement de l'application ; ainsi que les ressources du système.
- **les modèles d'espace de configuration et de variabilité** spécifient les variantes possibles du système, tandis que les modèles de configuration et d'architecture représentant l'état du système courant. Ces modèles définissent – par intention ou par extension – l'ensemble des états possibles et autorisés du système. Ainsi, en utilisant ces modèles, des points d'adaptation d'un système ainsi que des alternatives d'adaptation peuvent être identifiés.
- **les règles, stratégies, contraintes, besoins et buts** peuvent faire référence aux modèles des autres catégories et par conséquent, leurs niveaux d'abstraction sont similaires ou supérieurs aux niveaux des modèles référés. Les modèles dans cette catégorie définissent quand et comment un système doit être adapté et sous quelles conditions.

Cette catégorisation permet de montrer que des types différents de modèles à l'exécution peuvent être utilisés au même temps. Les catégories, ainsi que les types et le nombre de modèles à utiliser dépendent des objectifs du système modélisé (propriétés fonctionnelles et non-fonctionnelles, activités d'administration : surveillance, analyse, adaptation, etc.) et de son domaine métier (systèmes embarqués, mobiles, etc.).

Cependant, l'utilisation de multiples modèles lors de l'exécution d'un système emporte des problèmes de gestion des modèles et des relations entre eux. Vogel et al. proposent d'affronter de tels problèmes en utilisant des méga-modèles [58]. Un **méga-modèle** est défini comme :

> « *A megamodel is a model that contains models and relations between those models or between elements of those models.* » [58]

Un méga-modèle fournit un langage qui supporte la modélisation de modèles et de relations entre eux. La gestion des modèles et de leurs relations est réalisée ainsi d'une façon orientée modèles, permettant l'utilisation des techniques de l'IDM existantes.

Les méga-modèles fournissent ainsi un haut niveau d'automatisation pour utiliser des modèles à l'exécution et des relations. Ils peuvent être utilisés pour analyser automatiquement l'impact d'un changement d'un modèle sur ses modèles liés. Les relations sont utilisées pour synchroniser un changement d'un modèle avec ses modèles liés, et ces modèles synchronisés sont donc analysés pour connaître l'impact du changement initial. Ce processus d'analyse d'impacts peut être utilisé, par exemple, pour valider une adaptation affectant différents modèles avant d'adapter réellement le système.

3.6 SYNTHESE

L'approche IDM vise à utiliser des modèles pour la construction d'applications logicielles. L'IDM fournit deux propriétés fondamentales pour la construction d'applications : l'abstraction et la séparation de préoccupations. En effet, la méta-modélisation préconise l'utilisation de langages de haut niveau d'abstraction qui peuvent être exploitables de façon automatisé.

De plus, une application peut être représentée (prescrite et/ou décrite) à travers plusieurs modèles, chacun représentant une préoccupation différente de l'application. Des mécanismes de composition de modèles (et de métamodèles) sont utilisés afin d'intégrer des préoccupations différentes de l'application. Grâce à ces propriétés, l'IDM contribue à la maîtrise de la complexité des applications.

L'IDM offre des supports pour gérer tout le cycle de vie des applications, de la conception à l'exécution. En effet, l'approche *models@run.time* part des principes de l'IDM et propose d'utiliser à l'exécution les modèles produits à la conception des applications afin de supporter la gestion de leur exécution. L'utilisation des modèles de développement à l'exécution permet ainsi d'effacer l'écart entre les modèles de développement et les modèles à l'exécution.

Dans cette thèse, l'ingénierie dirigée par les modèles dans sa vision étendue (*models@run.time*) est utilisée comme paradigme pour la modélisation et l'administration d'applications. Notre objectif est d'étendre à l'exécution l'utilisation des modèles produits à la conception des applications afin de supporter la gestion de leur exécution, réduisant ainsi l'écart entre la conception et l'exécution des applications.

CHAPITRE 3
DYNAMISME ET ADAPTATION

Dans une première partie nous mettons en évidence les besoins de dynamisme et d'adaptation des applications modernes face aux variations de leurs contextes d'exécution. Nous soulignons les défis que ces besoins entraînent pour la conception ainsi que pour l'exécution des applications.

Nous présentons, dans une deuxième partie, le concept de contexte d'exécution et les différentes variations pouvant requérir l'adaptation d'une application pendant son exécution. Dans une troisième partie, nous mettons l'accent sur le concept de disponibilité dynamique, une variation que nous considérons particulièrement importante pour l'adaptation des applications. Ces adaptations, pouvant être effectuées à différents moments du cycle de vie d'une application, permettent de distinguer différents types d'applications, parmi eux les applications dynamiques et les applications adaptatives. Nous présentons les différentes techniques permettant d'implanter la logique d'adaptation des applications (quand et comment adapter), ainsi que les différentes opérations permettant d'effectuer des reconfigurations dynamiques.

Dans une quatrième partie, nous montrons comment le dynamisme et l'adaptation d'applications sont traités par différentes approches. Nous présentons des travaux existants pour chacune de ces approches, et nous définissions des critères de comparaison afin d'identifier les avantages et les limitations de ces travaux.

1 BESOINS DE DYNAMISME ET D'ADAPTATION

L'évolution de l'informatique a ouvert la voie à de nouveaux types d'applications. En effet, l'avènement de l'Internet et l'évolution d'objets informatiques communicants ont permis l'intégration du monde réel avec le monde informatique. Notamment, la vision de l'informatique ubiquitaire [1] conçoit des petits et peu coûteux dispositifs de traitement en réseau, intégrés et distribués dans notre environnement de la vie quotidienne afin de fournir des fonctionnalités diverses de manière continue et imperceptible. Par exemple, un environnement informatique ubiquitaire domestique pourrait relier les contrôles de l'éclairage et des radiateurs avec des capteurs de présence afin de moduler en conséquence l'éclairage et le chauffage dans une chambre. Dans l'environnement de la voiture, un scénario communément envisagé consiste à régler la destination du GPS en fonction des activités enregistrées dans l'agenda du PDA. Dans un aéroport, un utilisateur pourrait souhaiter être dirigé à l'aide de son PDA jusqu'à sa porte d'embarquement ou être informé en cas de modification de l'heure ou du lieu d'embarquement. Le but commun de ces différents environnements ubiquitaires est d'assister les utilisateurs dans leur vie quotidienne, en leur offrant divers services dont ils peuvent naturellement se servir en utilisant les dispositifs de l'environnement.

Nous pouvons constater ainsi que les applications modernes sont de plus en plus intégrées dans un monde ouvert en constante évolution : disponibilité dynamique de dispositifs, mobilité et préférences des utilisateurs, changements des besoins et des requis, etc. Cette **variabilité des contextes dans lesquels s'exécutent ces applications** rend difficile, voire impossible, d'avoir une connaissance complète, à leur conception, des conditions précises dans lesquelles elles seront utilisées et des services qui seraient les mieux adaptés à un instant donné. Les décisions prises à la conception d'une application peuvent changer après le déploiement du système au cours de son exécution. En effet, la variabilité du contexte d'exécution fait que l'architecture à l'exécution d'une application dépend, par exemple, des services disponibles ou des dispositifs accessibles. De ce fait, l'architecture de ce type d'applications n'est plus figée ni prévisible à la conception, rendant leur développement et leur exécution plus complexes.

En conséquence, ces **applications doivent être définies de manière flexible, permettant leur adaptation aux divers contextes d'exécution** auxquels elles seront confrontées et aux évolutions dynamiques de ceux-ci. Ces adaptations doivent permettre aux applications de continuer à fonctionner correctement malgré les variations de leur contexte, et de profiter au mieux des services apparus en cours d'exécution.

Le besoin de construire des applications qui s'adaptent à leur contexte n'est pas nouveau. Cependant, les caractéristiques des nouveaux environnements, telles que l'hétérogénéité et la distribution des plates-formes d'exécution, la disponibilité de ressources, le dynamisme et la diversité des utilisateurs, rendent cet objectif de plus en plus important et de plus en plus difficile à atteindre. Bien que de nombreux travaux aient étudié ce besoin, le développement d'applications flexibles qui peuvent s'adapter dynamiquement à leur contexte d'exécution, ainsi que leur administration restent très complexes.

Le défi est ainsi de trouver le dégrée de flexibilité, de dynamisme et d'adaptation requis pour les applications logicielles modernes. Ces besoins de flexibilité, de dynamisme et d'adaptation doivent être considérés par les infrastructures de développement et d'exécution d'applications. En effet, l'adaptation d'applications présente des défis importants qui doivent être traités : la cohérence (intégrité) et la conformité des applications après leur adaptation, l'arrêt sécurisé des composants impactés lors de l'adaptation des applications, le transfert d'état lors de la mise à jour ou la migration de composants.

Dans les sections suivantes nous présentons les concepts liés à l'adaptation dynamique d'applications.

2 CONTEXTE D'EXECUTION

La notion de contexte d'exécution est un concept fondamental pour l'adaptation d'applications. En effet, avec le passage de l'informatique de bureau traditionnelle aux environnements mobiles et ubiquitaires, la nécessité d'exploiter des informations implicites – qui font partie du contexte d'exécution – est de plus en plus accentuée afin de contrôler et d'adapter le comportement des applications. Cependant, cet ensemble d'informations varie selon l'application considérée et ses objectifs recherchés : des applications différentes ne sont pas « sensibles » aux mêmes informations. Le concept de contexte d'exécution a ainsi été largement étudié. En effet, la caractérisation, la représentation et l'acquisition du contexte d'exécution constituent un domaine de recherche en soi.

De nombreuses définitions du concept de contexte ont été proposées. Une définition citée fréquemment, et que nous considérons adaptée à notre travail, est celle donnée par Dey et Abowd :

> « *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.* » [59]

D'après cette définition, le contexte est construit à partir de toute information qui peut être utilisée pour caractériser la situation d'une entité. Une entité est une personne, un endroit ou un objet considérée pertinente pour l'interaction entre l'utilisateur et l'application. Cette définition considère la pertinence des entités pour l'interaction entre un utilisateur et une application, permettant d'utiliser des informations explicitement fournies par l'utilisateur ainsi que des informations calculées.

Différentes classifications des contextes d'exécution ont été proposées dans la littérature. Dans [60], les auteurs présentent la classification de contextes dans deux dimensions : **externe** (ou physique) et **interne** (ou logique). La dimension externe fait référence au contexte qui peut être mesuré par des senseurs matériels. La dimension interne est spécifiée par l'utilisateur ou capturée à travers la surveillance des interactions de l'utilisateur. Une autre classification est présentée dans [61], où les contextes sont classifiés dans trois catégories :

- **le contexte informatique** (*computing context*) qui regroupe les aspects techniques liés aux capacités de calcul et aux ressources matérielles utilisés : la capacité du réseau, les périphériques et ressources accessibles, etc.
- **le contexte physique** (*physical context*) qui rassemble les aspects représentant le monde réel et qui sont accessibles par de senseurs : la localisation de dispositifs et d'utilisateurs, la température, le niveau sonore et de luminosité, etc.
- **le contexte de l'utilisateur** (*user context*) qui regroupe l'ensemble des informations sur l'utilisateur : son profil, des informations sur les personnes à proximité, sa situation sociale actuelle, etc.

Les contextes d'exécution des applications sont de plus en plus variables. En considérant les classifications précédentes, les variations des contextes peuvent être classifiées comme :

- **variations dans le temps** : la disponibilité des éléments logiciels (par exemple à cause de la mobilité des utilisateurs) et des ressources matérielles (processeur, mémoire, réseau) peut varier pendant l'exécution d'une application.
- **variations dans l'espace** : les applications peuvent être utilisées dans une grande diversité de plates-formes hétérogènes (cartes à puce programmables, téléphones portables, machines multiprocesseurs) et distribuées qui ont des ressources différentes (en termes de capacité de traitement, de stockage, d'affichage).
- **variations des utilisations et des utilisateurs** : les applications peuvent être utilisées par différents utilisateurs, avec des niveaux d'expertise et des besoins différents, demandant ainsi aux applications des modes de fonctionnement différents.
- **variations des objectifs de l'application** : les objectifs d'une application (ses propriétés, ses contraintes) peuvent être redéfinis par son concepteur/administrateur.

Ces variations peuvent affecter le fonctionnement des applications, imposant ainsi leur adaptation. En outre, les applications devant assurer de plus en plus une disponibilité quasiment ininterrompue, leurs adaptations doivent donc être réalisées de manière dynamique, i.e., pendant leur exécution.

3 DYNAMISME ET ADAPTATION

Dans cette section, nous présentons un type de variation des contextes d'exécution que nous considérons particulièrement important pour les applications à services : la disponibilité dynamique. Nous présentons ensuite la notion d'adaptation dynamique ainsi que différentes techniques qui permettent sa mise en œuvre.

3.1 DISPONIBILITE DYNAMIQUE

La disponibilité dynamique concerne les éléments logiciels qui font partie ou qui peuvent faire partie d'une application pendant son exécution. L'informatique ubiquitaire, par exemple, est caractérisée par la disponibilité dynamique des équipements : un dispositif mobile peut entrer et sortir d'une zone contrôlée à cause des changements de localisation de son utilisateur, il peut être allumé ou éteint, volontairement ou involontairement (à cause du manque d'énergie par exemple).

Cervantes définit la disponibilité dynamique, dans le contexte des applications à composants à services, comme :

> « *le fait qu'une instance ou une classe de composant puisse devenir disponible ou indisponible à tout moment pendant qu'une application qui l'utilise ou qui pourrait l'utiliser est en train d'être exécutée.* » [21]

Cette définition prend en compte la distinction entre classe de composant (implémentation) et instance de composant. De ce fait, la disponibilité dynamique peut être provoquée par :

- **des créations ou des destructions d'instances de composants** durant l'exécution.
- **des changements dans la validité d'une instance** vis-à-vis de l'état de ses dépendances, permettant à l'instance de fournir (ou non) ses fonctionnalités.
- **des départs ou des arrivées de types de composants** grâce au déploiement continu pendant l'exécution.

Cependant, Simon considère que la définition de disponibilité dynamique donnée par Cervantes est une définition partielle. Il définit donc différents types de disponibilité en les caractérisant par rapport à la notion de gestion de cycle de vie. La **disponibilité dynamique** est ainsi définie comme :

> « *le fait qu'un élément logiciel puisse devenir* ***indisponible*** *ou* ***disponible*** *durant l'exécution d'une application, et ce* ***indépendamment de la volonté de l'administrateur*** *de l'application.* » [28]

Cette définition considère que le cycle de vie des éléments logiciels n'est pas maitrisé par l'administrateur de l'application. Par exemple, un opérateur téléphonique peut géo-localiser un client à partir de son téléphone portable seulement si le téléphone est allumé et s'il se trouve dans une zone couverte. La décision d'éteindre un téléphone ou de sortir un client de la zone couverte n'est pas maîtrisée pour l'opérateur. Ainsi, les téléphones sont considérés par l'opérateur comme des éléments logiciels qui ont une disponibilité dynamique, pouvant apparaître ou disparaître arbitrairement durant l'exécution de l'application. Les plates-formes d'exécution permettant aux applications de dépendre de ce type d'éléments doivent fournir des mécanismes de notification de la disponibilité ou de l'indisponibilité des éléments afin de pouvoir les administrer.

Un élément logiciel peut avoir une **disponibilité semi-dynamique**, qui est définie comme :

> « *le fait qu'un élément logiciel puisse devenir* ***indisponible*** *ou* ***disponible*** *durant l'exécution d'une application,* ***partiellement sous le contrôle de l'administrateur*** *de l'application.* » [28]

Cette définition considère que l'administrateur peut contraindre partiellement des éléments logiciels et les masquer aux applications. L'exemple donné est une application qui intègre des équipements ubiquitaires réifiés par des *proxies*. L'administrateur peut décider de masquer des équipements à l'application. La disponibilité de ces éléments, étant partiellement contrôlée par l'administrateur, est donc dite semi-dynamique. Les plates-formes d'exécution d'applications doivent définir des mécanismes de notification de la disponibilité ou de l'indisponibilité des éléments, ainsi que permettre l'accès à la gestion de leurs cycles de vie (création, destruction, configuration).

Un élément peut avoir une **disponibilité semi-statique**, qui est définie comme :

> « *le fait qu'un élément logiciel puisse devenir* ***indisponible*** *ou* ***disponible*** *durant l'exécution d'une application,* ***totalement sous le contrôle de l'administrateur*** *de l'application.* » [28]

Cette définition considère que les cycles de vie des éléments logiciels, ainsi que de l'application, sont contrôlés par l'administrateur, comme dans le cas d'utilisation basique d'une plate-forme OSGi qui fournit des opérations d'administration du cycle de vie des composants (*bundles*).

Enfin, un élément peut avoir une **disponibilité statique**, qui est définie comme :

> « *le fait qu'un élément logiciel est* ***invariablement disponible*** *durant l'exécution d'une application, selon la volonté de l'administrateur de l'application.* » [28]

La disponibilité statique est le cas des applications dont la composition de l'application est définie avant le déploiement ; la modification d'un composant requiert donc l'arrêt de l'application.

Dans toutes ces définitions, l'administrateur d'une application est un des facteurs responsables de la disponibilité des éléments logiciels durant l'exécution de l'application. Nous considérons que cette distinction entre les différents types de disponibilité des éléments logiciels par rapport au contrôle de l'administrateur est importante car elle permet de prévoir et de définir des adaptations (reconfigurations) à réaliser face à la disponibilité dynamique des éléments logiciels afin de permettre à l'application de continuer à fonctionner.

En effet, si la disponibilité des éléments logiciels est contrôlée par l'administrateur de l'application, les adaptations à réaliser sont facilement concevables et l'adaptation est d'une certaine façon plus contrôlée. Dans le cas où l'administrateur n'a pas de contrôle sur la disponibilité dynamique des éléments logiciels, les adaptations sont plus difficiles à envisager.

Considérer la disponibilité dynamique des éléments logiciels permet ainsi de spécifier – lors de la conception d'une application – des **points de variabilité** de l'application, qui représentent des positions dans l'architecture de conception d'une application où des changements peuvent apparaître lors de l'exécution, et face auxquels des adaptations (choix d'autres variantes) doivent être réalisées.

3.2 ADAPTATION DYNAMIQUE

L'adaptation consiste à rendre un système apte à assurer ses fonctions dans des conditions particulières ou nouvelles. Il existe plusieurs moments pour adapter une application. L'adaptation réalisée plus tardivement (par exemple à l'exécution) est plus puissante (par rapport à une adaptation réalisée au développement par exemple), mais aussi plus complexe à mettre en place et à garantir.

Dans [62], McKinley et al. définissent une classification d'applications par rapport au moment où les compositions ou les adaptations sont réalisées : développement, compilation, déploiement, exécution (voir la Figure 18).

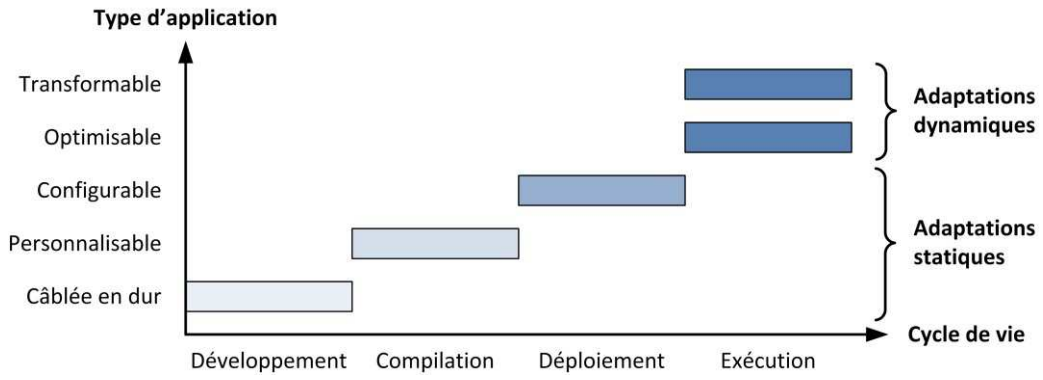


Figure 18. Classification des applications dynamiques

Une **adaptation statique** a lieu au développement, à la compilation ou au déploiement. Pour une application composée complétement au niveau développement (*hardwired*), le comportement adaptif est câblé dans le logiciel. L'adaptation aux changements de l'environnement requiert donc la modification du code d'adaptation de l'application.

Les applications personnalisables (*customizable*) sont adaptées à la compilation (ou à la création de liens) en les configurant pour un environnement particulier. Par exemple, les langages de programmation à aspects comme AspectJ permettent d'intégrer des aspects aux applications au moment de la compilation. Afin de s'adapter à un nouvel environnement, les applications personnalisables doivent être recompilées.

Les applications configurables sont adaptées durant leur déploiement. Les applications sont configurées pour satisfaire certains besoins : lorsqu'une application demande le déploiement d'un nouveau composant, le composant le plus adapté aux besoins actuels est sélectionné à partir d'une liste de composants avec des propriétés et des implémentations différentes. Par exemple, lorsqu'un utilisateur démarre une application sur son *smartphone*, le système d'exécution peut sélectionner et charger un composant d'affichage minimal afin de garantir la bonne présentation de l'application. Bien que l'adaptation au déploiement soit considérée comme statique, elle offre plus de dynamisme que d'autres méthodes statiques.

Pour ces trois types d'applications – câblées en dur, personnalisables et configurables – les adaptations sont statiques. En effet, une fois une adaptation effectuée, l'architecture de l'application et sa configuration ne sont plus modifiables à l'exécution. De ce fait, les adaptations statiques ne considèrent pas la disponibilité dynamique ni la disponibilité semi-dynamique des éléments logiciels.

D'un autre côté, lorsque l'adaptation a lieu à l'exécution, elle est considérée comme une **adaptation dynamique**. Une adaptation dynamique peut être effectuée sans besoin d'arrêter et de redémarrer une application. Deux types d'applications peuvent être distingués dans cette phase : les applications optimisables (*tunable*) et les applications transformables (*mutable*).

Les applications optimisables supportent l'adaptation des aspects d'une application en fonction de l'évolution de son contexte d'exécution. Toutefois, le code métier de l'application n'est pas modifié. Par exemple, quand la résolution d'un écran change, l'application peut reconfigurer le composant d'affichage utilisé, ou utiliser un autre composant plus adapté à la nouvelle résolution. Ces adaptations n'ont pas d'impact sur le code de l'application, mais sur son architecture.

En revanche, les applications transformables permettent la modification du code et de l'architecture de l'application. L'adaptation permet ainsi de changer la logique métier de l'application,

résultant ainsi en une application fonctionnellement différente. Par exemple, dans la plate-forme OpenORB[33], tous les composants et les applications ont des interfaces réflexives, permettant ainsi à une application de changer ses composants en modifiant leurs interfaces ou leurs implémentations.

Lorsque une adaptation modifie l'architecture d'une application à l'exécution, cette dernière est qualifiée **d'architecture logicielle dynamique** [63] et l'action de modifier l'architecture est appelée **reconfiguration dynamique**.

La reconfiguration dynamique des applications est en général nécessaire afin de réagir à la disponibilité dynamique, semi-dynamique et semi-statique des composants logiciels. La Figure 19 présente un exemple de reconfiguration d'une application issue de l'indisponibilité dynamique d'une des instances qui la composent.

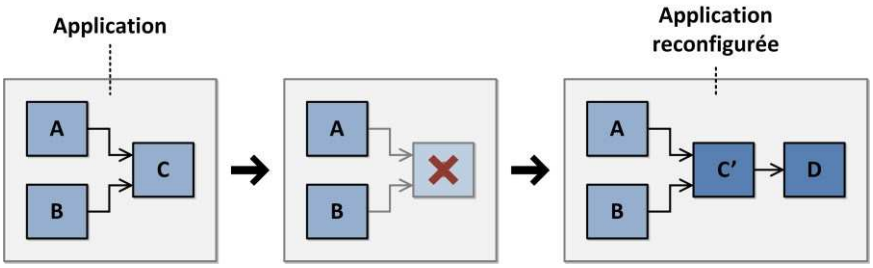


Figure 19. Reconfiguration issue de la disponibilité dynamique d'un composant

Dans cet exemple, nous considérons que l'application est composée de trois instances de composants, A, B et C. A et B sont connectés à C par une relation de dépendance. A un moment donné, l'instance C devient indisponible. Une solution d'adaptation possible consiste à utiliser une instance disponible et compatible avec C. Pour cela, les liaisons de A et B vers C sont détruites, et de nouvelles liaisons sont créées vers l'instance compatible (C'). Dans le cas où aucune instance compatible ne serait disponible, l'application peut se mettre en attente ou s'arrêter.

En résumé, une **application dynamiquement adaptative** est une application qui est optimisable ou transformable lors de son exécution. Une telle application fournit des moyens permettant à un acteur externe (par exemple, un gestionnaire d'adaptation) de réaliser des adaptations dynamiques face aux variations de son contexte d'exécution. Une **application auto-adaptative** est capable de s'adapter elle-même, lors de son exécution, face aux variations de son contexte d'exécution.

3.3 LOGIQUE D'ADAPTATION

La logique d'adaptation d'une application définit (i) **les informations à superviser** afin de détecter des situations d'adaptation, et (ii) **les opérations d'adaptation** à effectuer face aux situations détectées. La logique d'adaptation définit ainsi **quand adapter** et **comment adapter**. Elle peut être exprimée à travers différents types de politiques [64] :

- **des politiques d'action** qui dictent les actions à effecteur face à une situation prédéterminée afin d'assurer le comportement souhaité d'une application. Typiquement, les politiques d'action sont exprimées sous forme de règles ECA (Evénement, Condition, Action) permettant ainsi de spécifier un ensemble d'actions à effectuer quand un événement a lieu et que certaines conditions sont remplies. La simplicité et la rapidité d'expression de ces politiques les rend très populaires, surtout pour des applications dont les situations sont peu nombreuses et connues à l'avance. En effet, ce type de politiques est utilisé depuis longtemps dans les systèmes adaptatifs.

[33] OpenOBR est un OBR (CORBA 2.4.2) *open source* pour Java. **http://openorb.sourceforge.net/**

- **des politiques de but** qui définissent un ensemble de situations ou des propriétés désirables d'une application : les buts. Les actions à effectuer pour satisfaire ces buts sont déterminées par le système. Ces politiques offrent ainsi une alternative lorsque les politiques d'action sont inadaptées à cause du grand nombre de situations possibles. Les politiques de but sont bien adaptées aux systèmes qui raisonnent sur l'architecture du système.
- **des politiques de fonctions d'utilité** qui permettent de caractériser le degré de satisfaction d'une situation par rapport au but global de l'application. Une situation n'est plus considérée comme valide ou invalide : elle reçoit une valeur indiquant le degré de satisfaction, calculée en fonction d'un ensemble de paramètres qui caractérisent la situation. En utilisant ce type de politiques, un système peut chercher à optimiser sa configuration : deux situations peuvent être comparées. Ces politiques, tout comme les politiques de buts, permettent de gérer des cas non prévus par le concepteur du système. Cependant, leur difficulté est d'inférer les actions à effectuer pour atteindre une situation dont le degré de satisfaction serait meilleur.

3.3.1 RECONFIGURATION DYNAMIQUE

La reconfiguration dynamique est le processus de modification de l'architecture d'une application durant son exécution. La reconfiguration dynamique d'une application est possible grâce à la représentation de son architecture d'exécution, i.e., à son modèle descriptif.

Une reconfiguration dynamique peut être faite par des *scripts* de reconfiguration. Les modifications peuvent être effectuées par un gestionnaire d'adaptation (voir la Figure 20) qui doit manipuler la structure de l'application tout en minimisant la durée de l'interruption, et en assurant l'intégrité de l'application (i.e., un état cohérent) avant et après la reconfiguration.

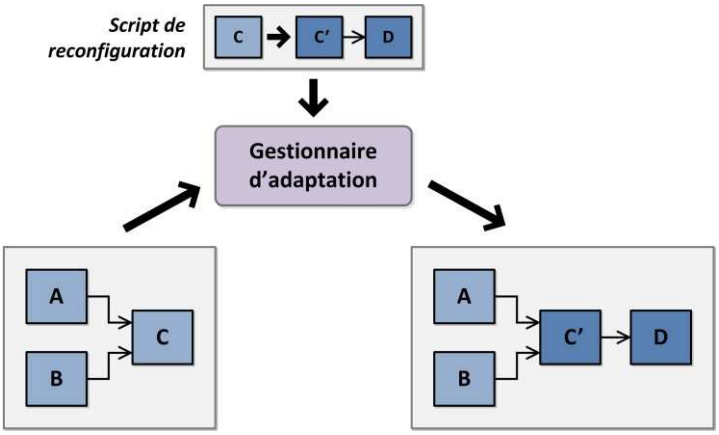


Figure 20. Reconfiguration dynamique d'une application

Les opérations de base sur les composants (types et instances) qui permettent de réaliser la reconfiguration dynamique des applications sont :

- l'ajout d'un composant
- le retrait d'un composant
- la création d'une liaison entre composants
- la destruction d'une liaison entre composants
- la mise à jour d'un composant
- la relocalisation physique d'un composant

D'autres opérations sont nécessaires afin de garantir l'intégrité des applications lors de la reconfiguration des applications [27] :

- la passivation d'un composant
- l'activation d'un composant

Une des approches les plus utilisées pour assurer l'intégrité des applications est la **quiescence**, proposé par Kramer et Magee [65]. Ils proposent différents états (*status*) pour les composants d'une application. Un composant dans un état *actif* peut initier, accepter et traiter des transactions. Un composant dans un état *passif* continue à traiter les transactions en cours et peut initier des transactions requises par d'autres transactions en cours afin de permettre sa complétion. Cependant, la passivation d'un composant bloque toutes les nouvelles transactions entrantes du composant afin de permettre sa manipulation. Un composant peut être manipulé une fois qu'il est dans un état *quiescent* : il est passif et il ne traite plus des transactions. Un composant dans un état quiescent est dans un état cohérent car il ne contient pas de résultats de transactions incomplètes. Une fois une reconfiguration terminée, les composants passivés peuvent être réactivés.

Toutefois, la quiescence demande un contrôle total de l'application afin de détecter les transactions entrantes des composants. De plus, les transactions imbriquées sont très coûteuses en termes de temps d'interruption de l'application : un grand nombre de composants doivent être passivés. D'autres approches, comme la **tranquillité** (*tranquility*) [66], ont été proposés pour assurer l'intégrité d'une application toute en réduisant le temps d'interruption lors de son adaptation.

Une opération de reconfiguration peut être décrite comme une séquence ou une combinaison de patrons, chacun définissant un ensemble d'opérations de reconfiguration. Parmi ces patrons nous pouvons mentionner **le remplacement**, **l'interposition** et **la migration dynamiques**.

Dans le remplacement dynamique, le but est de remplacer un composant par un autre composant compatible, sans arrêter l'exécution de l'application et en minimisant le temps d'interruption du fonctionnement de l'application. Considérons l'exemple présenté dans la Figure 21, où le composant *C* doit être remplacé par le composant *C'*.

Pour réaliser le remplacement dynamique d'un composant, l'interposition peut être utilisée. Elle consiste à insérer un intercepteur devant un composant pendant l'exécution d'une application. Tous les appels à ce composant seront donc capturés par l'intercepteur, qui est lui-même lié au composant. Dans le cas du remplacement dynamique, un tel intercepteur est appelé médiateur. Une fois le médiateur disponible, le remplacement d'un composant est effectué en deux phases (illustrées dans la Figure 21) :

- **la phase de blocage** : les nouveaux appels au composant *C* sont bloqués par le médiateur, les appels en cours sont autorisés à se terminer. Lorsque le composant *C* est dans un état quiescent, la phase de transfert est démarrée.
- **la phase de transfert** : le médiateur réalise le transfert d'état (si nécessaire) entre *C* et *C'*. Ensuite, les liaisons vers *C* sont mises à jour (*rebinding*) afin de pointer sur *C'*. Enfin, les appels bloqués sont repris en utilisant le composant *C'*.

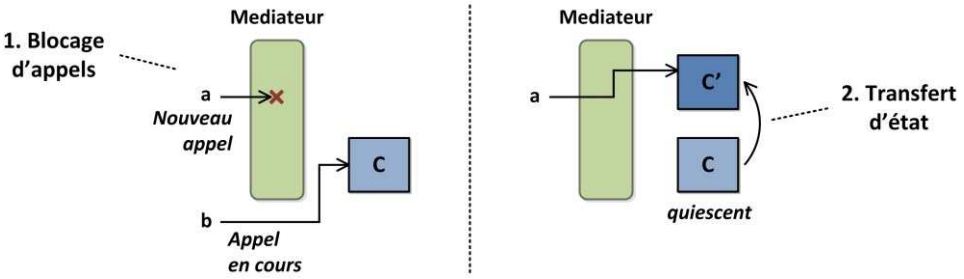


Figure 21. Remplacement dynamique d'un composant en utilisant un médiateur

La migration dynamique d'un composant C d'un site S à un site S', utilise le patron de remplacement dynamique : C est remplacé pour une copie de lui-même placée sur le site S'. La copie du composant est faite quand le composant C est dans un état quiescent.

La reconfiguration dynamique est donc primordiale pour l'adaptation dynamique d'une application. Un dysfonctionnement du processus de reconfiguration peut avoir des effets indésirables sur l'application. De ce fait, assurer l'intégrité de l'application lors des reconfigurations dynamiques est un défi majeur.

3.3.2 EMPLACEMENT

La logique d'adaptation (quand et comment adapter) d'une application peut être implantée selon différentes approches :

La logique d'adaptation est mélangée avec la logique métier de l'application. L'adaptation (gestion du dynamisme, détection de situations, exécution de reconfigurations) est directement gérée dans le code de l'application. Malgré la contradiction avec le principe de séparation des préoccupations, cette approche est la plus simple et l'une des plus utilisées aujourd'hui car l'adaptation est définie et implémentée en même temps que le code fonctionnel. L'avantage de cette approche est que le code d'adaptation est complétement spécifique et intégré à l'application, permettant une gestion très spécialisée du dynamisme sans avoir besoin d'une infrastructure spécifique. Cependant, la modification du comportement de l'adaptation requiert la modification des éléments de l'application, complexifiant ainsi sa maintenance. En outre, toutes les sources de dynamisme ne peuvent être gérées de cette façon. Ainsi, cette approche peut être utilisée pour des applications avec des besoins de performance importants et de dynamisme limité.

La logique d'adaptation est fusionnée avec la logique métier de l'application. Le code d'adaptation est séparé du code de l'application, respectant ainsi le principe de séparation de préoccupations. Une approche couramment utilisée est la programmation par aspects (AOP[34]), qui permet de séparer le code des aspects du code métier d'une application. L'intégration des aspects est réalisée à la compilation de l'application. Cette approche permet d'avoir une meilleure maintenabilité de l'application. Cependant, le code d'adaptation reste spécifique aux composants de l'application.

La logique d'adaptation est définie dans le conteneur de l'application. Cette approche est utilisée dans les systèmes basés sur des modèles à composants. Cette approche respecte aussi le principe de séparation des préoccupations, et par conséquent, l'application n'est pas impactée par les modifications du code d'adaptation. Cependant, le code d'adaptation est généralement dépendant du composant pour lequel il a été conçu : le code d'adaptation doit connaître le fonctionnement interne du composant pour savoir comment l'adapter. De ce fait, le code d'adaptation est spécifique au composant et généralement il n'est pas réutilisable. De plus, la maintenance des deux parties n'est pas aussi indépendante qu'il y parait : un changement dans le code de l'application peut impacter le code d'adaptation. Bien que cette approche comporte de nombreux avantages, elle reste aujourd'hui peu utilisée car elle est généralement difficile à mettre en œuvre.

La logique d'adaptation est définie séparément de l'application. Cette approche suit aussi le principe de séparation de préoccupations. Elle découple totalement la logique d'adaptation de l'application. Cette approche se base sur les interfaces d'administration des composants de l'application préalablement définies, permettant ainsi de s'abstraire de l'implémentation des composants de l'application. L'avantage de cette approche est que le code d'adaptation de l'application est indépendant de toute contrainte liée à l'application (technologie, topologie, architecture), cependant elle requiert que les interfaces nécessaires à l'adaptation soient définies au préalable.

En outre, la logique d'adaptation peut être implantée de façon centralisée ou distribuée.

[34] *Aspect Oriented Programming*

4 APPROCHES EXISTANTES

Le dynamisme et l'adaptation d'applications ont été abordés selon différentes perspectives. Certains travaux se focalisent sur la spécification d'architectures et de reconfigurations dynamiques. D'autres se concentrent sur les mécanismes de bas niveau qui fournissent un support pour effectuer des adaptations dynamiques. D'autres se focalisent sur la conception de systèmes auto-adaptatifs capables d'adapter dynamiquement leur architecture par rapport aux buts prédéfinis.

Dans la suite, nous présentons diverses approches traitant le dynamisme et l'adaptation d'applications : l'approche à services, les architectures logicielles, les modèles à composants, les systèmes adaptatifs. Nous décrivons des travaux représentatifs pour chacune de ces approches.

4.1 APPROCHE A SERVICES DYNAMIQUE

Dans le chapitre précédent, nous avons présenté l'approche à services et le principe d'interaction entre ses différents acteurs. Comme nous l'avons vu, ce principe d'interaction peut avoir lieu à n'importe quel moment dans le cycle de vie de l'application : au développement, au déploiement ou à l'exécution. De ce fait, les fournisseurs de services peuvent s'enregistrer auprès de l'annuaire et les consommateurs peuvent découvrir les fournisseurs pendant l'exécution des applications.

Retarder le principe d'interaction à l'exécution permet ainsi la construction d'applications dynamiques. Afin de supporter le dynamisme, deux nouvelles primitives ont été ajoutées au principe d'interaction de base (voir la Figure 22) :

- **la notification** qui informe les consommateurs de l'arrivée ou du départ d'un fournisseur de service, et
- **le retrait de services** qui signale qu'un fournisseur de service n'est plus en mesure d'offrir son service.

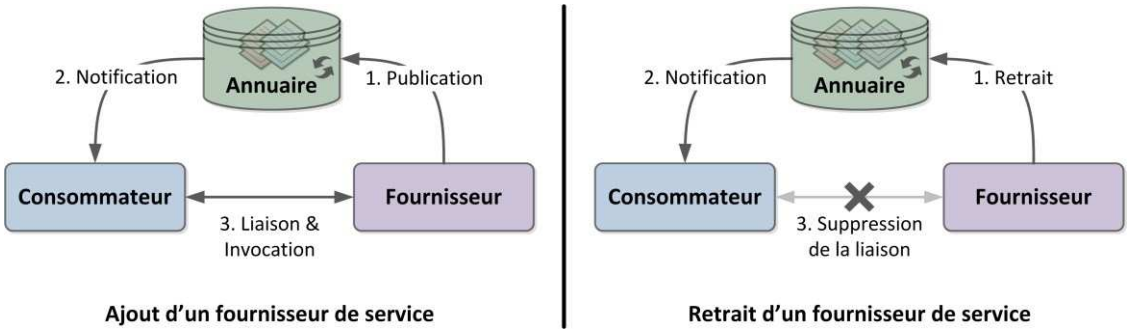


Figure 22. Approche à services dynamique

Grâce à ces deux primitives, les consommateurs de services peuvent gérer la disponibilité dynamique de services et réagir à leur apparition ou leur disparition. Un consommateur d'un service pourrait, par exemple, sélectionner un autre fournisseur d'un service compatible – parmi plusieurs disponibles – afin de s'adapter à son contexte d'exécution.

L'architecture à services étendue (SOA étendu) proposée par Papazoglou [7] peut être étendue afin de prendre en compte le dynamisme. Un telle architecture, appelée **SOA dynamique étendu**, reprend les concepts du SOA étendu, mais se focalise sur la gestion du dynamisme. Ainsi, les primitives de notification et de retrait de services doivent être ajoutées aux mécanismes de base du SOA. Les mécanismes de composition doivent être capables de gérer l'arrivée et le départ des services participant à une composition afin de gérer la structure de la composition et d'assurer son fonctionnement. Les mécanismes de gestion et de supervision doivent être capables de vérifier en permanence la cohérence et la conformité de l'application en fonction des buts. De plus, un SOA

étendue dynamique peut fournir des mécanismes additionnels afin de gérer, de manière transversale, des propriétés non-fonctionnelles des applications telles que la sécurité ou la qualité de service. Implémenter tous les mécanismes du SOA dynamique étendu pour la gestion du dynamisme est une tâche extrêmement complexe. De ce fait, la plupart des SOA supportant le dynamisme ne sont pas étendus : ils ne supportent pas la composition ni la gestion d'applications.

Dans le chapitre précédent, nous avons décrit deux SOA : OSGi et les services Web (cf. Chapitre 2, section 1.2.4). Afin de compléter, nous présentons ci-dessous la façon dont chacune de ces technologies traite le dynamisme.

4.1.1 OSGI

La plate-forme à services OSGi fournit les mécanismes suivants permettant la création d'applications à services dynamiques :

- **des mécanismes de déploiement :** les *bundles* sont déployés et administrés à l'exécution. Il est possible d'installer, de démarrer, d'arrêter, de mettre à jour ou de supprimer des *bundles* à l'exécution sans arrêter la plate-forme.
- **des mécanismes de notification de services :** les primitives de retrait et notification de la disponibilité dynamique de services sont implantées par la plate-forme. Les fournisseurs de services (contenus dans les *bundles*) peuvent apparaître et disparaître à l'exécution. Un consommateur peut être notifié des arrivées et des départs de services, ainsi que des modifications de leurs propriétés publiées, et agir en conséquence.

Toutefois, comment nous l'avons souligné précédemment, OSGi ne fournit pas de mécanismes de composition d'applications. Toute composition ainsi que le dynamisme qui l'influence doivent être gérés par le développeur. En effet, le dynamisme des services peut valider ou invalider une composition. Le développeur de l'application doit gérer l'apparition et la disparition de services, y compris le relâchement des services disparus.

Ainsi, bien que la plate-forme OSGi fournisse les mécanismes de base pour créer des applications dynamiques, la manque de support pour la définition de compositions rend complexe la gestion des applications au-dessus de cette plate-forme.

4.1.2 LES SERVICES WEB

Les services Web peuvent être mis à disposition à travers le registre UDDI. Ce registre peut être utilisé pour trouver, au développement ainsi qu'à l'exécution, des fournisseurs et des services. Cependant, le registre UDDI est aujourd'hui peu utilisé : la majorité des organisations qui fournissent des services Web le font sans passer par des registres en indiquant directement leur localisation. Ainsi, la consommation de services entre consommateurs et fournisseurs est directe.

En plus de l'enregistrement et de la recherche de fournisseurs et de services, le registre UDDI permet aux fournisseurs de se retirer de l'annuaire ainsi que de retirer leurs services fournis. Il supporte aussi l'enregistrement de consommateurs de services pour recevoir des notifications concernant des changements dans le registre : ajout, retrait et modification au niveau des services ou des fournisseurs. La gestion du dynamisme (apparition et disparition de services participant à une composition) doit être gérée par les développeurs des applications.

Bien que les mécanismes de sélection et de notification sont spécifiés et implantés, ceux-ci sont rarement utilisés dans le milieu industriel : les entreprises utilisent rarement les annuaires UDDI, préférant assigner aux clients des services Web identifiés et souvent contractualisés.

4.2 ARCHITECTURES LOGICIELLES DYNAMIQUES

Nous avons introduit dans le chapitre précédent le concept d'architecture logicielle. Nous avons vu que ce concept est fondamental pour la conception et le développement d'applications ainsi que pour la gestion de leur exécution.

Cependant, afin de considérer et de supporter le dynamisme des contextes d'exécution, les architectures logicielles doivent pouvoir préciser la variabilité des applications permettant de gérer leur adaptation dynamique : des composants et des connecteurs peuvent être ajoutés résultant ainsi en une architecture différente à l'architecture initiale.

Les architectures logicielles dynamiques permettent de décrire la partie fixe d'une application (i.e., la partie statique) ainsi que la partie variable (i.e., la partie dynamique). Une architecture logicielle dynamique permet donc d'adapter la structure d'une application lors de son exécution. Baresi et al. proposent la définition suivante d'architecture logicielle dynamique :

> « *Dynamic Software Architectures represent systems that do not simply consist of a fixed, static structure, but can react to certain requirements or events by run-time reconfiguration of its components and connections.* » [67]

Le dynamisme d'une architecture logicielle peut comporter l'adaptation dynamique (des modifications sur la structure d'une application) ainsi que l'évolution dynamique des applications (des modifications sur la définition de l'application par l'ajout ou la modification de types par exemple).

Divers ADLs ont été proposés pour la description d'architectures logicielles dynamiques. De manière générale, ces approches permettent d'intégrer des politiques spécifiques de reconfiguration qui décrivent quand et comment l'architecture d'une application doit être reconfigurée : les reconfigurations sont programmées. Dans la suite, nous présentons deux langages représentatifs pour la description d'architectures dynamiques : Darwin et Dynamic Wright.

4.2.1 DARWIN

Darwin [45] est un ADL déclaratif qui permet de décrire le dynamisme d'une application. Ce langage fournit une sémantique basée sur le π-calcul.

Les applications décrites en utilisant Darwin peuvent être hiérarchiques : elles peuvent contenir des composants primitifs et des composants composites. Les composants primitifs sont définis dans un langage de programmation tel que Java ou C++. Leurs interfaces sont décrites par l'ADL Darwin afin de permettre la spécification de l'architecture de l'application. Les descriptions des composants composites contiennent des instances de composants (primitifs ou composites) et des connecteurs entre ces instances (déclarés par l'opérateur *bind*). A partir d'une telle description, des instanciations sont effectuées afin de créer une architecture exécutable.

Darwin fournit deux mécanismes pour spécifier le dynamisme des applications lors de l'instanciation de leurs composants :

- **l'instanciation paresseuse :** un composant, sélectionné au préalable, est instancié lorsqu'un autre composant le demande. Dans l'exemple de la Figure 23a, le service (*Server*) sera instancié lors de sa première invocation par le client.
- **l'instanciation dynamique :** un composant est arbitrairement choisi et instancié en fonction de ses interfaces fournies (Figure 23b). Un service particulier pour la création des composants dynamiques est nécessaire.

L'instanciation paresseuse permet de décrire les configurations potentielles d'une application, en déclarant les instances qui seront créées suite à son invocation lors de l'exécution de l'application. L'instanciation paresseuse ne permet d'instancier qu'un seul composant par clause d'interconnexion, contrairement à l'instanciation dynamique qui permet de multiples instanciations par clause.

Toutefois, il est impossible de distinguer une instance parmi un ensemble d'instances créées dynamiquement. De ce fait, la suppression d'une instance ou d'une liaison particulière, ainsi que la liaison avec une instance particulière d'un composant créée dynamiquement ne sont pas supportées.

```
component ClientServerSystem {
  inst
    client : Client;
    server : dyn Server;
  bind
    client.c -- server.s;
}
```

a. Instanciation paresseuse

```
component ClientServerSystem {
  inst
    client : Client;
  bind
    client.create -- dyn Server;
    client.c -- Server.s;
}
```

b. Instanciation dynamique

Figure 23. Exemple dans Darwin

4.2.2 DYNAMIC WRIGHT

Wright est un langage de description qui permet de représenter la structure d'une application en termes de composants, connecteurs et configurations, chacun décrit par une algèbre de processus proche de CSP[35]. Dynamic Wright [43] est une extension de Wright qui intègre la description de reconfigurations dynamiques dans la description des applications. Ce langage est utilisé pour la vérification/simulation d'une application ou d'une famille d'applications par rapport aux différentes reconfigurations spécifiées. Par exemple, les contraintes architecturales sont vérifiées automatiquement.

Dynamic Wright décrit les configurations et les reconfigurations sous la forme de processus. Une configuration (*configuror*) décrit comment les instances de composants et les instances de connecteurs interagissent (voir la Figure 24). Il décrit ainsi la configuration initiale de l'application, mais aussi des configurations possibles en termes de conditions (des événements déclenchés par des processus constituant l'application) et de reconfigurations à effectuer sur l'architecture.

```
CONFIGUROR ClientServeur
  new.C : Client →
  new.P : ServeurPrimaire →
  new.S : ServeurSecondaire →
  new.Conn : Connecteur →
  attach.C.p to Conn.client →
  attach.P.p to Conn.serveur → Arrêt
  WHERE
  Arrêt = (P.control.down → S.control.up →
      Conn.control.changeOk →
      detach.P.p from Conn.serveur →
      attach.S.p to Conn.serveur → Rétablissement) → ν
  Rétablissement = (P.control.up → S.control.down →
      Conn.control.changeOk →
      detach.S.p from Conn.serveur →
      attach.P.p to Conn.serveur → Arrêt) → ν
```

Figure 24. Exemple dans Dynamic Wright

[35] *Communicating Sequential Processes*

Dans la Figure 24, la configuration initiale décrit l'instanciation des composants *Client*, *ServeurPrimaire* et *ServeurSecondaire* et du connecteur *Connecteur,* et la création de liens entre eux. Deux reconfigurations sont aussi décrites, chacune attachée à une condition (*Arrêt* et *Rétablissement*) qui déclenche la reconfiguration. Les opérateurs *new*, *del*, *attach* et *detach* sont utilisés pour décrire les reconfigurations de l'architecture.

Dynamic Wright supporte ainsi l'ajout et la destruction d'instances de composants et la création et la destruction de liaisons. Cependant, Dynamic Wright ne possède pas d'environnement d'utilisation ou d'exécution : les architectures décrites ne peuvent être que vérifiées et simulées.

4.3 MODELES A COMPOSANTS DYNAMIQUES

Le dynamisme a été aussi considéré dans le développement de modèles et de plates-formes à composants. Ces modèles supportent différents types de dynamisme ou l'automatisent de différentes façons. Cette section présente divers modèles à composants, en se focalisant sur les caractéristiques qu'ils fournissent pour supporter le dynamisme.

4.3.1 ARCHJAVA

ArchJava[36] [10] est une extension du langage de programmation Java qui introduit les éléments d'architecture (composant, connecteur et configuration) directement dans le langage. ArchJava ajoute ainsi de nouveaux éléments de syntaxe au langage Java pour gérer les composants et les connecteurs.

Un composant ArchJava (primitif ou composite) est décrit par une interface qui contient les services fournis et les services requis. Les composants composites contiennent la description d'un assemblage de composants et peuvent contenir aussi du code logiciel. Un composant composite représente une configuration (i.e., une application) : il contient la description des instances de composants de l'application ainsi que leurs interconnexions.

ArchJava garantit l'intégrité de l'architecture durant la phase de développement. A la compilation, ArchJava assure qu'aucune communication (connexion) ne vient contourner la structuration de l'application imposée par la description de l'architecture.

ArchJava offre la possibilité de créer des instances de composants et des connexions dynamiquement dans un composite. Cependant, afin de garantir l'acceptabilité des communications entre composants, toute nouvelle connexion dynamique doit avoir été déclarée préalablement. Les types de connexions acceptables entre deux composants sont définies au niveau d'un composite par le mot clé *pattern*. Par exemple, l'expression « **connect pattern** *Router.workers, Worker.serve;* » décrit une connexion possible entre le composant *Router* et le composant *Worker* à travers leurs ports *worker* et *serve*. Les connexions seront instanciées dynamiquement sur l'appel de la fonction *connect()* dans le code des composants.

La prise en charge du dynamisme reste toutefois limitée : le code lié à la gestion du dynamisme d'une architecture est mêlé au code métier des composants ; la destruction d'instances de composants et de connexions n'est pas supportée.

4.3.2 SOFA 2.0

SOFA 2.0[37] [12] est un modèle à composants dont la conception a été influencée par SOFA/DCUP [68], un modèle supportant la mise à jour dynamique d'applications. Ce modèle fournit un métamodèle pour la spécification de composants et des connecteurs entre composants.

[36] **http://www.archjava.org/**
[37] **http://sofa.ow2.org/**

Un composant (primitif ou composite) est représenté par un ensemble d'interfaces, fournies et requises, qui déterminent le type de composant. Un composant est géré par des contrôleurs modulables et extensibles. Deux contrôleurs de base sont fournis dans SOFA 2.0 : le contrôleur du cycle de vie des composants (*start*, *stop*, *udpate*) et le contrôleur de liaisons entre composants.

Les fonctionnalités de reconfiguration dynamique sont encapsulées dans les contrôleurs des composants. Afin de contrôler la modification d'une architecture, seulement les reconfigurations dynamiques conformes à un patron de reconfiguration prédéfini sont autorisées.

SOFA 2.0 permet la création et la suppression d'instances de composants et de connecteurs à l'exécution ainsi que le remplacement et la mise à jour dynamiques d'instances. SOFA 2.0 supporte la gestion des versions de composants, permettant du partitionnement passif, i.e., la coexistence d'anciennes et de nouvelles versions.

Une application SOFA 2.0 est exécutée dans un environnement d'exécution distribué, appelé SOFAnode, qui est composé d'un ensemble de conteneurs de déploiement (i.e. la machine virtuelle de Java plus SOFA 2.0 *runtime*), hébergés sur différents ordinateurs, fournissant de mécanismes pour l'exécution des composants SOFA 2.0. Un SOFAnode contient un dépôt qui concentre des descriptions de composants et leurs implémentations. Ce dépôt est utilisé, tout au long du cycle de vie de l'application, comme la source principale de composants.

4.3.3 FRACTAL

Fractal[38] [14], introduit dans le chapitre précédent (cf. section 1.1.2.2), fournit des capacités de partage, d'introspection et de reconfiguration. Julia[39] [15], une des plates-formes les plus utilisées de Fractal, permet la spécification de composants Java et d'architectures de composants, ainsi que leur manipulation à l'exécution. Chaque composant se compose d'un contrôleur, qui gère toutes les interactions du composant avec l'extérieur. Ce contrôleur propose un ensemble d'interfaces de contrôle qui fournissent des capacités de réflexion : introspection, création et destruction de liaisons, ajout et suppression de sous-composants, etc. Ces interfaces permettent de réaliser des reconfigurations dynamiques ad-hoc et programmées mais limitées : elles sont contraintes par la description de l'architecture qui définit une unique configuration.

Toutefois, Fractal étant un modèle ouvert et extensible, d'autres approches ont proposé des extensions afin d'augmenter le niveau de dynamisme. Par exemple, WildCAT[40] [69] permet de détecter les modifications du contexte d'exécution des applications. Ces modifications génèrent des événements qui peuvent être capturés par exemple par des gestionnaires de politiques de reconfiguration afin d'adapter les applications à leur contexte d'exécution.

SAFRAN [70], une autre extension du modèle Fractal, utilise des concepts et des techniques de l'AOP[41] afin de développer le code d'adaptation de manière séparée au code métier des applications. SAFRAN propose d'encapsuler des mécanismes de reconfiguration dans des aspects, nommés aspects d'adaptation. Un aspect d'adaptation définit des politiques d'adaptation (des règles ECA[42]), exprimées dans un langage de reconfiguration spécifique appelé FScript.

SAFRAN permet d'ajouter ou de supprimer des aspects d'adaptation aux composants d'une application pendant l'exécution. Il fournit : la capture d'événements internes et externes ; des capacités d'introspection permettant d'analyser l'architecture en exécution et les propriétés des composants ; des mécanismes pour créer, supprimer, lier, délier des instances de composants ; des mécanismes pour vérifier la cohérence des reconfigurations et annuler celles échouées (*rollback* de transactions).

[38] **http://fractal.ow2.org/**
[39] **http://fractal.ow2.org/julia/**
[40] **http://wildcat.ow2.org/**
[41] *Aspect Oriented Programming*
[42] *Event Condition Action*

4.4 SYSTEMES ADAPTATIFS

Il existe deux approches pour réaliser des systèmes adaptatifs qui diffèrent par la façon dont les patrons d'interaction du système sont gérés. D'une part, les approches *top-down* reposent sur un modèle prescriptif, généralement centralisé, qui décrit l'architecture du système à réaliser. L'administration est guidée par ce modèle : les décisions sont prises sur ce modèle et des modifications sont appliquées sur le système. Un exemple d'approche *top-down* est celui des systèmes auto-adaptatifs, basés sur un modèle architectural et un ensemble de buts de haut niveau guidant le processus d'adaptation.

D'autre part, dans les approches *bottom-up*, généralement décentralisées, les interactions sont gérées localement par les composants du système. L'auto-adaptation émerge des décisions d'adaptation locales prises par chacun des composants. Un exemple d'approches *bottom-up* sont les systèmes auto-organisés (*self-organizing*) basés sur des fonctions algorithmiques guidant le processus d'adaptation. Ces systèmes sont composés d'instances autonomes qui s'exécutent et s'auto-organisent par rapport à des critères différents face aux changements du contexte d'exécution.

Dans la suite de cette section, nous présentons divers travaux représentatifs implémentant ces approches.

4.4.1 K-COMPONENT

K-Component [11] est un modèle à composant, développé au-dessus de CORBA, qui permet de spécifier des applications auto-adaptatives. Le modèle sépare l'implémentation des composants de la gestion du dynamisme. Les activités d'adaptation : observation, raisonnement et exécution de reconfigurations, sont effectuées directement sur l'architecture réifiée de l'application et reproduites sur l'application en exécution.

Une application, composée d'un ensemble d'instances de composant et de connecteurs, est supervisée par un gestionnaire de configuration. Ce gestionnaire maintient en permanence le graphe des instances et des connecteurs qui composent l'application. Les interactions entre le gestionnaire de configuration et l'application sont effectuées par des événements émis par l'application afin de remonter des informations au gestionnaire, et émis par le gestionnaire de configuration afin d'effectuer des reconfigurations sur l'application.

Les reconfigurations sont décrites dans des contrats d'adaptation. Ces contrats définissent des règles conditionnelles permettant la reconfiguration de l'architecture d'une application en fonction des événements émis par l'application ou l'environnement, et la définition de contraintes architecturales. La reconfiguration de l'architecture est réalisée en utilisant les opérations de reconfiguration fournies par le gestionnaire de configuration. Lors d'une adaptation, une instance de composant peut être remplacée en la supprimant de l'architecture, et un ajoutant une autre fournissant la même interface. Cependant, des nouveaux types ne peuvent pas être y ajoutés.

Le gestionnaire de configuration fournit un environnement d'exécution de contrats d'adaptation qui supporte le chargement et déchargement dynamique de contrats ; et fournit des capteurs et des actionneurs qui permettent d'agir sur l'application en exécution.

K-Component est ainsi un modèle à composants intéressant pour créer des applications auto-adaptatives dynamiquement. Le fait de pouvoir reconfigurer directement l'architecture de l'application permet de définir des reconfigurations avec des politiques de haut niveau.

4.4.2 RAINBOW

Le projet Rainbow [71] définit un langage et une plate-forme pour construire des applications auto-adaptatives. Le langage permet d'exprimer des politiques d'administration et d'adaptation. La plate-forme fournit un ensemble de mécanismes réutilisables qui permettent de collecter (via des

sondes) les informations correspondantes à l'architecture d'une application en cours d'exécution, ainsi que de réaliser (via des effecteurs) des adaptations sur cette architecture. Ces mécanismes facilitent la conception et l'implémentation de gestionnaires d'adaptation, permettant aux concepteurs de se concentrer sur d'autres aspects de l'application plutôt que sur les mécanismes d'adaptation.

La plate-forme d'exécution de gestionnaires d'adaptation fournie par Rainbow est indépendante de la plate-forme d'exécution des applications administrées. Une application auto-adaptative Rainbow est donc composée de : (i) l'application administrée instrumentée via des sondes et des effecteurs qui s'exécutent sur le même support d'exécution que l'application et collectent des informations ; (ii) une infrastructure de traduction qui transforme l'ensemble des informations collectées afin de produire une représentation abstraite de l'architecture de l'application en exécution indépendante de sa plate-forme d'exécution ; et (iii) un gestionnaire d'adaptation découpé en quatre parties :

- le gestionnaire du modèle qui maintient le modèle de l'architecture en exécution,
- l'évaluateur de l'architecture qui analyse la conformité de l'architecture à un ensemble de contraintes fixées par le concepteur du gestionnaire,
- le gestionnaire d'adaptation qui détermine la stratégie à adopter pour faire revenir l'application à un état acceptable, et
- l'exécuteur de stratégies qui démarre les stratégies sur l'architecture de l'application en passant par l'infrastructure de transformation.

Ce découpage est conforme au découpage MAPE-K (*Monitor*, *Analyze*, *Plan*, *Execute*, *Knowledge*) proposé par IBM [72] pour la construction de gestionnaires autonomiques. L'implémentation d'un gestionnaire d'adaptation requiert donc l'implémentation de mécanismes de traduction. Toutefois, grâce à l'utilisation de mécanismes de traduction, les gestionnaires implémentés peuvent s'adapter à des applications différentes.

Une des principales limitations de Rainbow est que le raisonnement et les adaptations sont basés uniquement sur l'architecture de l'application administrée : d'autres variations du contexte d'exécution ne sont pas considérées. En outre, les gestionnaires d'adaptation construits avec Rainbow sont centralisés. De ce fait, Rainbow possède les défauts des solutions centralisées : point unique de défaillance et problème de passage à l'échelle lorsque le nombre d'informations capturées devient important. Dans [73], une approche de coordination du fonctionnement de plusieurs gestionnaires d'une application a été proposée, cependant, cette approche permet simplement de décomposer la fonction d'administration en plusieurs entités et non de gérer l'application de manière décentralisée.

4.4.3 JADE

Jade [74] est un environnement qui permet la gestion autonome d'infrastructures logicielles. Jade se focalise essentiellement sur la gestion de systèmes complexes, notamment d'applications patrimoniales et d'applications J2EE s'exécutant sur grappes de nœuds.

Jade fournit ainsi des mécanismes qui permettent d'instrumenter une application administrée. L'application est vue comme une composition structurelle de composants administrables. L'application est représentée dans le modèle à composants Fractal proposant des mécanismes d'introspection et de réflexion nécessaires à l'adaptation. La composition ainsi que la configuration de l'application peuvent évoluer dans le temps. Jade maintient, dans une base de connaissances, la représentation de l'architecture actuelle de l'application administrée : le raisonnement et l'adaptation sont basés sur l'architecture d'exécution de l'application.

L'administration d'applications est divisée en aspects comme l'autoréparation ou l'auto-optimisation. Chaque aspect est traité par un gestionnaire autonomique qui spécifie ses propres politiques (algorithmes) de gestion. Jade fournit des mécanismes communs permettant la construction de tels gestionnaires autonomiques. De plus, Jade fournit des gestionnaires autonomiques génériques qui prennent en charge les aspects d'autoréparation et d'auto-optimisation. Le gestionnaire d'autoréparation répare l'architecture d'une application lors de la détection d'une panne. Le

gestionnaire d'auto-optimisation permet d'allouer automatiquement des ressources en fonction du temps de réponse des composants. Bien que Jade permette d'administrer une application au travers de plusieurs gestionnaires autonomiques, les conflits entre les différents gestionnaires ne sont pas gérés.

4.4.4 ADAPT-MEDIUM

Adapt-Medium [75] est une approche basée sur l'architecture qui traite l'adaptation dynamique d'applications distribuées. Cette approche est basée sur le concept de « *adapt-medium* », qui définit un ensemble de gestionnaires d'adaptation locaux qui collaborent pour réaliser des adaptations à une application pendant son exécution. Les adaptations dynamiques qui peuvent être effectuées sont définies – lors du développement – comme un ensemble d'alternatives ou de variantes architecturales (similaire à l'approche des lignes de produits).

Toutes les variantes liées à un composant d'une application sont inclues dans son gestionnaire d'adaptation (*adapt-manager*) qui est déployé avec le composant, dans le même hôte. Lorsqu'une adaptation (une variante de l'architecture) est nécessaire, par exemple suite à un changement dans le contexte d'exécution, les gestionnaires d'adaptation des composants impliqués dans l'adaptation composent un *adapt-medium* afin de gérer de manière collaborative l'adaptation. Les adaptations sont effectuées en remplaçant des composants par des composants alternatifs. L'approche supporte des adaptations programmées de manière décentralisée.

4.5 COMPARAISON DES APPROCHES

Cette section compare les différentes approches présentées dans la section précédente. Bien que toutes ces approches visent à supporter le dynamisme et l'adaptation d'applications logicielles, elles le font à partir de points de vue différents. Afin d'évaluer les capacités des différentes approches et de les comparer, nous avons identifié certains critères que nous jugeons pertinents pour la construction et l'exécution d'applications dynamiquement adaptatives :

- **spécification formelle :** l'approche est basée sur un formalisme (logique, graphes, algèbre de processus) permettant la spécification, à un haut niveau d'abstraction, de l'architecture d'une application. Les spécifications formelles permettent de vérifier les propriétés et le comportement des applications, supportant ainsi le développement d'applications dynamiques, conformes et robustes.
- **séparation de préoccupations :** l'approche sépare les différents aspects d'une application, notamment, l'aspect d'adaptation de la logique métier. La séparation des préoccupations peut être considérée au niveau spécification : la spécification de la logique d'adaptation d'une application est séparée de celle de la logique métier de l'application ; et au niveau des mécanismes supportant l'exécution de l'application : les mécanismes traitant les adaptations sont séparés du code fonctionnel de l'application.
- **niveau de l'adaptation :** l'approche supporte différents types de modifications lors de l'exécution d'une application : reconfiguration (la structure peut être modifiée par des opérations de reconfiguration impactant l'application à l'exécution), évolution de la spécification (la définition peut être modifiée en ajoutant des types de composants impactant un ensemble d'applications à l'exécution).
- **types de déclencheurs :** l'approche supporte différents types de déclencheurs d'adaptations : entités externes (changements réactifs), l'application elle-même (changements proactifs).
- **types de modifications :** l'approche supporte différents types d'opérations pour la reconfiguration dynamique d'applications : ajouts d'éléments (implémentations et instances de composants), suppression d'éléments, mise à jour et remplacement d'éléments, création dynamique de liaisons entre éléments, et suppression de liaisons entre éléments.

- **gestion de l'adaptation :** l'approche fournit une gestion de l'adaptation centralisée (la spécification et le contrôle de l'adaptation sont centralisés) ou distribuée (la spécification de l'adaptation est distribuée sur les éléments de l'application).
- **introspection :** l'approche fournit des mécanismes permettant à une application d'observer son état interne (ses propriétés) et sa configuration (ses composants et ses relations). Cette propriété permet à une application de raisonner sur elle-même afin de s'auto-administrer.
- **cohérence et la conformité :** l'approche fournit des mécanismes pour garantir la cohérence des applications avant et après le processus d'adaptation dynamique (transfert d'état, arrêt sécurisé, transactions) ainsi que pour garantir la conformité.

Le tableau suivant montre les approches présentées auparavant en les positionnant par rapport à ces critères.

		Spécification formelle	*Séparation de préoccupations*	*Niveau d'adaptation*		*Types de déclencheurs*		*Types de modifications*	*Gestion « C » Centralisée ou « D » Décentralisée*	*Introspection*	*Cohérence*			*Conformité*
				Reconfiguration	*Evolution*	*Externes*	*Internes*				*Transfert d'état*	*Arrêt sécurisé*	*Transactionnel*	
ADL	***Darwin***	✓	x	✓	x	x	✓	½	C	x	x	✓	✓	✓
	Dynamic Wright	✓	x	✓	x	x	✓	✓	C	x	x	x	x	✓
Modèles	***ArchJava***	x	x	✓	x	x	✓	½	C	✓	x	x	x	✓
	Sofa 2.0	✓	✓	✓	x	✓	✓	✓	D	✓	x	x	x	✓
	Fractal (Julia)	x	✓	✓	x	x	✓	✓	D	✓	x	x	x	x
Systèmes	***K-Component***	x	✓	✓	x	x	✓	½	D	✓	✓	✓	x	?
	Rainbow	x	✓	✓	x	x	✓	½	C	✓	x	x	x	?
	Jade	x	✓	✓	x	x	✓	½	D	✓	x	x	x	x
	Adapt-Medium	x	✓	✓	x	x	✓	½	D	x	x	x	x	?

Tableau 6. Comparaison des approches

D'après ce tableau, nous pouvons conclure qu'il existe un écart entre les approches traitant la spécification de l'adaptation à un haut niveau d'abstraction et les approches fournissant des mécanismes pour effectuer des adaptations dynamiques. En effet, les langages de description d'architectures dynamiques, comme Darwin et Dynamic Wright, se concentrent sur les spécifications plutôt que sur les mécanismes supportant le dynamisme.

Certaines approches, comme SOFA 2.0 et Julia, se concentrant sur l'architecture à l'exécution des applications, et fournissent des mécanismes d'adaptation dynamique basés sur l'architecture. D'autres approches, comme Rainbow, se focalisent sur la spécification de l'adaptation ainsi que sur les mécanismes supportant les adaptations : l'adaptation d'une application est effectuée par rapport aux objectifs spécifiés. De plus, le support d'adaptation fourni pour les approches reste limité : le raisonnement et l'adaptation sont généralement basés sur l'architecture des applications, l'évolution de la définition des applications n'est pas supportée, les types de modifications (reconfigurations) supportés sont restreints, la cohérence et la conformité des applications suite aux adaptations ne sont pas garanties.

5 SYNTHESE

Afin de construire des applications dynamiques et adaptatives, divers aspects doivent être considérés : la spécification du dynamisme autorisé (quand adapter), la spécification de l'adaptation (comment adapter), l'implémentation de mécanismes supportant la gestion du dynamisme et de l'adaptation (introspection, opérations de reconfiguration, vérification de la cohérence et la conformité).

La mise en place des applications adaptatives reste ainsi très complexe. En effet, la spécification et l'implémentation de la logique d'adaptation de ces applications sont des tâches difficiles qui ont été attaquées par différentes approches. Cependant, nous constatons que les solutions proposées par les travaux existants sont restreintes : elles se focalisent soit sur la spécification du dynamisme et de l'adaptation, soit sur les mécanismes d'adaptation de l'architecture des applications à l'exécution.

Certains travaux cherchent à réduire cet écart entre la spécification et la réalisation de l'adaptation et proposent des mécanismes de gestion du dynamisme et d'adaptation par rapport aux buts de dynamisme et d'adaptation de l'application. Toutefois, le support fourni par ces approches reste limité : la cohérence des applications n'est pas gérée, la conformité n'est pas garantie, l'évolution de la définition n'est pas supportée.

Dans cette thèse nous proposons des formalismes pour la définition d'applications dynamiquement adaptatives. Notre approche permet l'adaptation d'une application à son contexte d'exécution, par composition dynamique et adaptative, en conformité et cohérence avec sa définition. De plus, des aspects spécifiques à l'adaptation d'une application peuvent être définis de manière séparée à la définition de l'application. Notre plate-forme d'exécution gère le dynamisme des applications. Elle est extensible via des gestionnaires, permettant ainsi d'administrer une application avec ses différents aspects associés : chaque gestionnaire traite un aspect spécifique de l'application. Ainsi, le gestionnaire d'adaptation fait partie des gestionnaires nécessaires pour contrôler l'exécution d'une application selon des aspects d'adaptation spécifiques.

CHAPITRE 4
COMPOSITION DE SERVICES

Les approches à composants, à services et à composants à services, présentées dans le Chapitre 2, offrent la possibilité de construire des applications en assemblant des entités logicielles préexistantes. Ce processus d'assemblage est appelé une **composition de services** et le résultat peut être un nouveau service appelé service composite.

Dans ce chapitre, nous nous intéressons à la composition de services. Nous décrivons les différentes approches de composition de services. Nous présentons divers travaux dédiés à la composition de services en soulignant leurs avantages et leurs limitations afin d'identifier les besoins et les défis de la composition d'applications à services dynamiques et adaptatives.

1 PROCESSUS DE COMPOSITION

La composition de services permet de développer des applications en réutilisant des services existants pour créer d'autres services plus complexes.

L'approche à services spécifie le patron d'interaction entre fournisseurs et consommateurs de services (publication, recherche, liaison, invocation). Cependant, elle n'indique pas comment réaliser une application par composition de services. Plusieurs approches, langages et outils ont ainsi été proposés, tant dans le monde académique qu'industriel, pour réaliser des applications par composition de services.

Dans [76], le processus de composition de services est réalisé à travers plusieurs phases qui permettent le passage de la spécification abstraite d'une composition vers une composition concrète de services :

- **la phase de conception** permet de définir des services composites d'une manière abstraite. La fonctionnalité fournie par la composition ainsi que les fonctionnalités apportées par les différents services participants doivent être identifiées.
- **la phase de planification** identifie l'ensemble des services requis conformes à la description abstraite de la composition définie dans la phase précédente.
- **la phase de construction** sélectionne les services, selon une stratégie de sélection donnée, parmi ceux identifiés dans la phase de planification. Les services sont préparés pour l'exécution : des configurations et des adaptations éventuelles peuvent être réalisées. Les liaisons entre les services sélectionnés sont établies. La définition concrète de la composition est réalisée dans cette phase.
- **la phase d'exécution** de la composition réalise l'invocation des services préparés au préalable.

Le processus de composition de services consiste ainsi tout d'abord à définir un service composite abstrait par les descriptions des services requis, ensuite à découvrir et à sélectionner les services parmi ceux qui sont disponibles, et enfin à réaliser les liaisons entre les services sélectionnés.

La découverte, la sélection et la liaison de services (i.e., les phases de planification et de construction) peuvent être réalisées avant ou pendant la phase d'exécution caractérisant ainsi la composition. Lorsque tous les services participant à la composition sont sélectionnés et liés avant la phase d'exécution, la composition est appelée **composition statique**. Quand la sélection et la liaison de services peuvent être réalisées à l'exécution, la composition est nommée **composition dynamique**. Enfin, lorsque les sélections et les liaisons déjà réalisées peuvent être modifiés à l'exécution, la composition est appelée **composition adaptative dynamique**.

Malgré sa décomposition en plusieurs phases, le processus de composition de services reste une activité complexe et de longue durée. En effet, chacune des phases est divisée en tâches complexes. Par exemple, les services identifiés pour la composition peuvent présenter des problèmes d'incompatibilité (des types de données d'entrée et de sortie des services par exemple). Des mécanismes de médiation doivent ainsi être crées et utilisés dans la phase de construction afin d'adapter les services les uns aux autres et de résoudre les problèmes d'incompatibilité. Cela est une tâche difficile qui, dans la plupart des approches, est effectuée manuellement par les développeurs des applications.

La complexité de la composition de services est associée donc non seulement à la complexité de la logique métier des applications, mais aussi à la complexité technique de la réalisation de l'approche à services. Les développeurs doivent avoir une double expertise : l'expertise métier et l'expertise technique nécessaire pour réaliser de compositions de services en utilisant des technologies à services particulières. Afin de permettre aux développeurs de se concentrer sur la logique métier des applications plutôt que sur les détails techniques, il est nécessaire de disposer de langages, de modèles et de mécanismes pour la réalisation de compositions de services.

2 APPROCHES DE COMPOSITION

Il existe diverses approches pour la réalisation de compositions de services. Elles sont souvent classifiées par rapport à la façon dont le contrôle de la composition est géré : extrinsèque ou intrinsèque aux services. Ces deux façons de gérer le contrôle définissent deux styles de composition : la composition par procédés, aussi appelée composition comportementale, et la composition structurelle.

2.1 COMPOSITION PAR PROCEDES

Dans cette approche, la composition de services est définie en spécifiant la logique de coordination de services par un procédé. Un procédé est généralement représenté par un graphe orienté d'activités et le flot de contrôle qui établit l'ordre d'invocation des activités. Chaque activité représente une fonctionnalité réalisée concrètement par un service. La composition est décrite dans un langage spécifique interprété par un moteur d'exécution particulier qui réalise les invocations des services, le routage des donnés d'un service à un autre, et la gestion des erreurs.

Dans une composition par procédé, le flot de contrôle est explicite et le contrôle des invocations de services est externe aux services composés. Deux catégories de composition de services par procédés sont distinguées :

- **l'orchestration de services** qui décrit, du point de vue du service composite, les interactions entre les différentes activités : messages échangés, ordre d'invocation (séquence, parallèle, choix) ; ainsi que des opérations internes réalisées entre ces interactions (transformations de données, invocations de modules internes). La réalisation de chaque activité correspond à l'invocation d'un service concret. Le contrôle de la composition est centralisé (voir la Figure 25).
- **la chorégraphie de services** qui décrit la collaboration d'un ensemble de partenaires pour atteindre un but commun. La chorégraphie décrit la façon dont les services participants doivent collaborer afin de remplir le but commun : les échanges de messages, les règles auxquelles les interactions entre les différents participants sont soumises, la façon dont les différents participants se coordonnent. Le contrôle de la composition est distribué entre les participants (voir la Figure 26).

L'orchestration et la chorégraphie de services peuvent être combinées pour permettre l'intégration de systèmes d'entreprises différentes. Chaque partenaire modélise en interne, par orchestration, la réalisation des fonctionnalités rendues disponibles. La collaboration entre les différents partenaires est modélisée par chorégraphie.

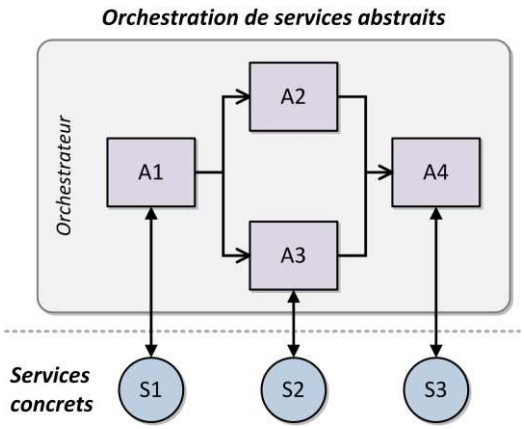


Figure 25. Orchestration de services

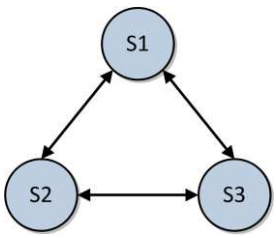


Figure 26. Chorégraphie de services

A la conception, une composition par procédés peut être décrite de manière abstraite par des procédés abstraits.

La sélection et la liaison des services concrets à utiliser peuvent être effectuées de manière statique – lors de la conception de la composition (les services à utiliser et les liaisons sont exprimés dans la spécification de la composition) ou du déploiement (un fichier de configuration est utilisé pour spécifier les services à utiliser) – ou dynamique pendant l'exécution.

La sélection et la liaison dynamiques peuvent être mises en place par des approches différentes [20]. Une première approche consiste à utiliser une variable pour stocker la référence du service à invoquer (*dynamic binding by reference*). Cette approche est générique car elle ne fait pas d'hypothèses sur la façon dont la référence est assignée à une variable. L'avantage de cette approche est sa simplicité d'utilisation et de mise en place. Cependant, son principal inconvénient est le mélange du modèle de composition et des opérations d'affectation des variables de références de services.

Une deuxième approche consiste à utiliser des requêtes, associées à des références de services et basées sur les propriétés de services, à être exécutées dans un dépôt afin de chercher et déterminer le service à invoquer (*dynamic binding by lookup*). Une autre approche consiste à déterminer dynamiquement non seulement le service, mais aussi l'opération à invoquer sur le service (*dynamic operation selection*). Cette approche offre une solution flexible qui permet la définition d'activités abstraites qui ne spécifient pas d'opération à invoquer. L'opération est sélectionnée à l'exécution, en même temps que le service.

Actuellement, la composition par procédés est utilisée seulement par la technologie des services Web. Divers langages ont été conçus pour l'orchestration et la chorégraphie de services Web comme WS-BPEL (décrit par la suite) et WS-CDL respectivement.

2.1.1 WS-BPEL

WS-BPEL, acronyme de *Web Services Business Process Execution Language*, est une spécification du consortium OASIS pour l'orchestration de services Web. Elle est devenue le standard pour l'orchestration de services Web remplaçant les spécifications précédentes XLANG[43] de Microsoft, et WSFL[44] d'IBM.

La spécification WS-BPEL définit un langage de procédés, basé sur XML, qui permet de décrire la logique de contrôle nécessaire pour orchestrer des services Web. Les procédés peuvent être modélisés de deux façons :

- **abstraite :** seuls les messages échangés entre les différents participants sont spécifiés (les participants ne sont pas explicitement spécifiés). Un procédé abstrait peut être utilisé pour produire plusieurs implémentations du service composite.
- **exécutable :** les services concrets requis sont identifiés et spécifiés dans la définition du procédé, les activités du procédé sont ordonnées, les messages échangés sont identifiés ainsi que le traitement des fautes et des exceptions.

Un procédé est composé d'activités enchaînées par des échanges de données. Ces activités peuvent être de type basique ou composite : les activités basiques (*invoke*, *receive*, *reply*) permettent d'interagir avec les services ; tandis que les activités composites (*flow*, *sequence*, *switch*, *while*) permettent d'exprimer le flot de contrôle du procédé.

Les liaisons entre le procédé et les services Web concrets peuvent être effectuées de façon statique, à la conception ou au déploiement ; ou dynamiquement pendant l'exécution du procédé. Si une liaison est effectuée à la conception, l'adresse (*endpoint*) du service concret est spécifiée dans la

[43] **http://www.ebpml.org/xlang.htm**
[44] *Web Services Flow Language*, **http://xml.coverpages.org/wsfl.html**

définition du procédé. Au déploiement, un fichier de description fournit l'information pour effectuer la liaison. A l'exécution, le langage fournit un mécanisme pour l'assignation de références (*dynamic binding by reference*). L'utilisation de ce mécanisme entraîne la modification des procédés car il est nécessaire de définir des variables pour stocker les références, ainsi que d'utiliser des activités pour récupérer les références de services.

Les procédés sont interprétés et exécutables par un moteur d'orchestration spécifique qui permet de réaliser les communications avec les services concrets et l'invocation des fonctionnalités de ces services. Actuellement, il existe de nombreux moteurs d'exécution de procédés comme OW2 Orchestra[45], Oracle BPEL Process Manager[46] ou IBM Business Process Manager[47].

WS-BPEL présente plusieurs limitations : les services composés sont des services Web décrits par des interfaces WSDL, l'orchestration d'autres types de services n'est pas possible. Le niveau d'abstraction de la spécification de la composition est assez bas [77] : la construction de la composition requiert, en plus des connaissances métier, des connaissances techniques de la part des développeurs, rendant assez complexe la spécification de la composition.

Plusieurs travaux ont été réalisés pour augmenter le niveau d'abstraction de la composition de services, par exemple, en s'appuyant sur l'ingénierie dirigée par les modèles. L'approche FOCAS[48] [78], développé dans notre équipe, fait partie des approches basées sur l'IDM pour la réalisation de compositions de services par orchestration. FOCAS permet de spécifier des orchestrations abstraites indépendantes des technologies utilisées pour l'implémentation des services. Un procédé est spécifiée par des préoccupations différentes (contrôle, donnés, services, aspects non-fonctionnels) : chaque préoccupation est modélisée par un métamodèle indépendant. L'orchestration de services est réalisée ainsi par la composition de différents modèles. FOCAS permet ainsi d'orchestrer différents types de services, comme des services Web, OSGi ou iPOJO. De plus, FOCAS fournit un environnement extensible (sous Eclipse) qui fournit des outils pour le développement et l'exécution d'orchestrations de services : éditeur de modèles, outils de composition de modèles, générateur de code, moteur d'exécution.

2.2 COMPOSITION STRUCTURELLE

Une composition structurelle consiste à décrire la structure d'une application en indiquant ses composants et les connexions entre eux. Chaque composant déclare explicitement les services qu'il fournit et ceux qu'il requiert. La composition assemble donc des composants dont les services requis par un composant correspondent aux services fournis par un autre composant.

La logique de contrôle, spécifiant comment et à quel moment les opérations des services composés doivent être invoquées, est implicite et repartie entre les différents composants : le contrôle est exprimé à l'intérieur des composants. Par exemple, la Figure 27 montre un assemblage de composants à services dont la logique de contrôle se trouve à l'intérieur du composant *A*.

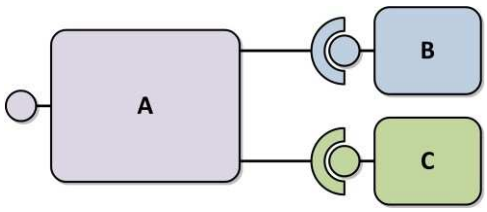


Figure 27. Composition structurelle de services

[45] **http://orchestra.ow2.org/**
[46] **http://www.oracle.com/technetwork/middleware/bpel/**
[47] **http://www-01.ibm.com/software/integration/business-process-manager/**
[48] *Framework for Orchestration, Composition and Aggregation of Services*

Dans le Chapitre 1, nous avons introduit l'approche à composants à services, dont la composition de services est définie de façon structurelle. Toutefois, l'approche permet également de composer des services réalisés de façon comportementale : une composition par procédés peut être utilisée comme une implémentation de service. Par exemple, SCA permet d'implémenter un composant par une composition comportementale telle que WS-BPEL. Le mécanisme de composition de services est défini par les modèles à composants à services.

La définition structurelle d'une composition de services en termes de spécifications de services donne plus de flexibilité à la composition et permet de contrôler l'assemblage des services d'un point de vue global. De plus, d'autres propriétés peuvent être associées à la définition d'un service composite, par exemple pour contraindre la composition (contraintes contextuelles), pour gérer le dynamisme des services, pour adapter la composition, etc.

Par la suite, nous allons présenter la spécification SCA [26] et l'approche iPOJO [27] qui fournissent des modèles à composants à services permettant de réaliser des compositions structurelles de services. L'objectif est de compléter leurs descriptions, présentées dans les chapitres précédents, avec les concepts associés à la composition et au dynamisme de services.

2.2.1 SCA

SCA [26] spécifie un modèle pour la création de composants (présenté dans la section 1.3.2.1 du Chapitre 2) et pour la construction d'applications. Une application SCA est un assemblage de composants, nommé **composite**, communiquant au travers de services (voir la Figure 28). Un composite SCA contient des composants (primitifs ou composites), des services, des références, les liaisons qui les connectent, et des propriétés qui peuvent être utilisées pour configurer les composants. Le modèle de composition SCA est hiérarchique et permet de lier des composants hétérogènes.

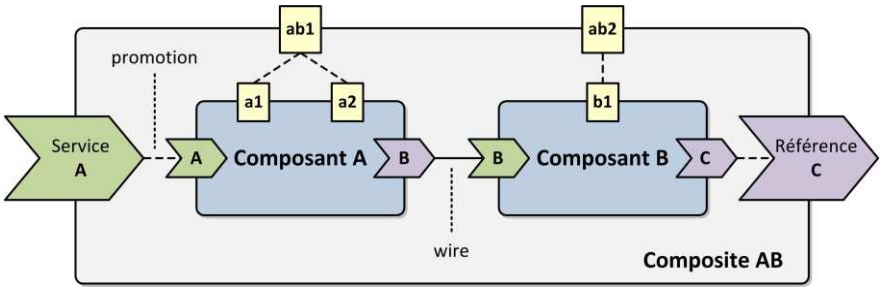
Un composite SCA a donc la même vue externe qu'un composant primitif SCA. De la même façon que les composants primitifs SCA, un composite SCA précise ses services fournis et ses références de services. Les services fournis par un composite se retrouvent parmi les services fournis par ses composants internes et sont exportés à l'extérieur du composite par un mécanisme appelé promotion (*promote*). Le même mécanisme est utilisé pour les références de services. La promotion des références des composants internes les rend visibles à l'extérieur du composite. Ces références seront donc résolues à l'extérieur du composite. Les propriétés d'un composite peuvent être aussi promues afin de configurer les composants internes à partir des valeurs indiquées pour le composite.

Dans l'exemple de la Figure 28, le Composite *AB* fournit le service *A*, promu du composant *A*. La référence *C* du composite *AB* est promue de la référence *C* du composant *B*. Les propriétés *ab1* et *ab2* du composite *AB* permettent de configurer les propriétés *a1* et *a2* du composant *A*, et *b1* du composant *B*, respectivement.

Les composants à l'intérieur d'un composite SCA sont liés par des ***wires***. Un *wire* connecte une référence de service d'un composant avec un service fourni par un autre composant. Une telle connexion est possible si les opérations définies par les deux interfaces des composants sont équivalentes (mêmes opérations, paramètres, valeurs de retour, exceptions) indifféremment du langage utilisé pour la description des interfaces.

Dans l'exemple de la Figure 28, le service *A* est lié (*wired*) au composant *A*. La référence *B* du composant *A* est liée au service *B* du composant *B*, qui à son tour a sa référence *C* liée à la référence *C* du composite *AB*. Ces connecteurs sont fixes et ne peuvent pas évoluer durant l'exécution de l'application. SCA spécifie une fonction appelée *Autowire*, dont l'objectif est de simplifier l'assemblage de composites : les références non promues et non liées explicitement à un service d'un composant sont automatiquement liées à des services de composants qui satisfont les références dans le même composite.

Les services contenus dans un composite peuvent être accessibles à distance. Le protocole de communication à utiliser (JMS[49], RMI[50], SOAP[51]) est spécifié sur les *bindings* des services et sur les *bindings* des références (voir la Figure 28).




```
<composite name="CompositeAB">
    <service name="ServiceA" promote="ComposantA">
        <interface.java interface="services.compositeAB.ServiceA"/>
        <binding.sca/>
    </service>
    <reference name="ServiceC" promote="ComposantB">
        <interface.java interface="services.c.ServiceC"/>
        <binding.ws port="http://www.c.org/ServiceC#
                wsdl.endpoint(ServiceC/ServiceCSOAP)"/>
    </reference>
    <property name="ab1" type="ab1:AB1ComplexType">
        <AB1ComplexPropertyValue xsi:type="ab1:AB1ComplexType">
            <ab1:a1>1</ab1:a1>
            <ab1:a2>2</ab1:a2>
        </A1ComplexPropertyValue>
    </property>
    <property name="ab2">3</property>
    <component name="ComposantA">
        <implementation.java class="services.compositeAB.ServiceAImpl"/>
        <property name="a1" source="$ab1/a1"/>
        <property name="a2" source="$ab1/a2"/>
        <service name="ServiceA"/>
        <reference name="serviceB" target="ComposantB"/>
    </component>
    <component name="ComposantB">
        <implementation.java class="services.compositeAB.ServiceBImpl"/>
        <property name="b1" source="$ab2"/>
        <reference name="serviceC"/>
    </component>
</composite>
```

Figure 28. Un composite SCA

Dans le but de contraindre les implémentations de composant pouvant être contenues dans un composite, SCA définit le concept de *constrainingType*. Un *constrainingType* contient des services, des références de services, des propriétés et des politiques abstraites de qualité de services, et est

[49] *Java Message Service*
[50] *Remote Method Invocation*
[51] *Simple Object Access Protocol*

indépendant de toute implémentation. Cette indépendance permet la définition abstraite de composites, en utilisant des *constrainingTypes*.

Un composite peut être utilisé dans un autre composite par inclusion. Quand un composite est inclus dans un autre composite, tout son contenu est disponible pour utilisation dans le composite inclusif : le contenu du composite inclus est complètement visible et peut être référencé par d'autres éléments dans le composite inclusif. SCA ne permet pas de contraindre la visibilité du contenu des composites inclus dans d'autres composites. De plus, la spécification SCA ne spécifie pas le partage de composants entre plusieurs composites qui ne sont pas imbriqués.

Il existe plusieurs implémentations, propriétaires et libres, de la spécification SCA. Parmi les implémentations commerciales, nous citons IBM WebSphere Application Server V7 Feature Pack for SCA[52] qui fournit des outils pour faciliter le développement, l'assemblage, le déploiement et l'exécution de composants SCA. Parmi les implémentations libres, le projet Eclipse SCA Tools[53] propose également des mécanismes et des outils pour la conception, l'assemblage, le déploiement, l'exécution et le monitoring de composants SCA. Il fournit entre autres des éditeurs et de validateurs de modèles (SCA Composite Designer et SCA XML Editor), des mécanismes de génération/introspection de code Java, des extensions pour l'exécution d'applications sur les plates-formes Apache Tuscany[54] et OW2 FraSCAti[55]. Certaines plates-formes comme Paremus Service Fabric[56] ou OW2 FraSCAti gèrent le dynamisme en s'appuyant sur des technologies à services dynamiques comme OSGi, ou en étendant le modèle fourni par la spécification SCA. Toutefois, la plupart des implémentations de SCA ne fournissent pas de support pour gérer le dynamisme et la reconfiguration dynamique d'applications.

SCA fournit un modèle de composition intéressant pour la création d'applications à composants à services. Cependant, les implémentations de composants de services ainsi que les liaisons entre composants sont déterminées et fixées au déploiement, empêchant toute sélection et substitution de composants à l'exécution. De plus, bien que les principes de l'approche à composants à services visent le support du dynamisme, SCA ne spécifie pas de concepts ni de mécanismes pour la gestion du dynamisme des composants, ce qui constitue une limitation majeure de cette spécification.

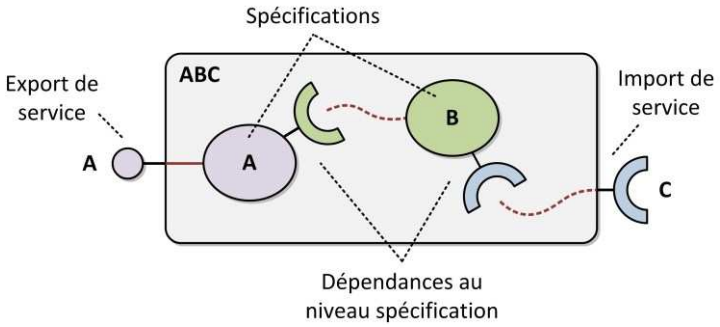
2.2.2 IPOJO

Le modèle iPOJO [27] distingue deux types de composants : les types de composants atomiques (présentés dans la section 1.3.2.2 du Chapitre 2) et les types de composants composites. Un type de composant **composite** iPOJO permet d'assembler structurellement d'autres types de composants. Il peut être spécifié en fonction des sous-services (les dépendances de service du composite), les services fournis et les instances qui seront créées directement. La Figure 29 présente le schéma d'un type de composite nommé *ABC* ainsi que sa description.

Tout comme un composant atomique, un composite IPOJO peut offrir des services au travers d'une ou plusieurs spécifications et peut dépendre d'autres spécifications de services. Les spécifications fournies et requises d'un composite peuvent être spécifiées à travers les mécanismes d'export et d'import de spécifications de services.

Les sous-services sont les services qui sont utilisés par le composite. Ils sont indiqués par leur spécification (voir la Figure 29). Deux types d'actions sont possibles afin d'utiliser un service : l'instanciation du service au sein du composite (*instantiate*), ou l'import du service depuis le contexte de service supérieur (*import*).

[52] **http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca/**
[53] **http://eclipse.org/soa/sca/**
[54] **http://tuscany.apache.org/**
[55] **http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti**
[56] **http://www.paremus.com/products/products_psf.html**




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

Figure 29. Description d'un composite iPOJO

Lors du déploiement d'un composite, iPOJO vérifie la cohérence de la composition. Une composition est cohérente si tous les services requis par les services et les instances internes spécifiés dans la composition seront disponibles lors de l'exécution. Cette vérification ne concerne que les dépendances de spécifications de services.

La résolution d'un composite (sélection/instanciation de services, réalisation de liaisons dynamiques) est réalisée par son conteneur. Pour chaque service à instancier, le conteneur recherche une implémentation du service. Si une implémentation compatible est disponible, une instance est créée à l'intérieur du composite. Lorsque la dépendance est agrégée, une instance par implémentation disponible est créée. La Figure 30 montre une instance du type de composite *ABC* présenté dans la Figure 29.

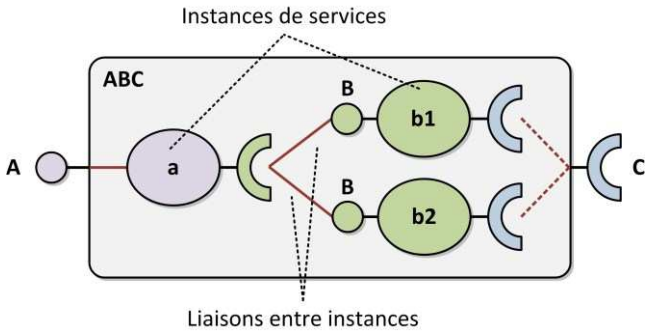


Figure 30. Instance d'un composite iPOJO

Une instance peut être remplacée si son implémentation disparaît. En effet, les dépendances, étant des dépendances de services, ne ciblent pas d'implémentation particulière. De ce fait, une instance peut être substituée par une autre provenant d'une implémentation disponible. De plus, l'état d'une instance peut être persistant et réinjecté dans une autre instance si la spécification de service définit un modèle d'état.

Un service importé du composite réagit de manière similaire aux dépendances de service des composants atomiques. Dès qu'un fournisseur est disponible à l'extérieur du composite (dans le contexte du service englobant), celui-ci est publié à l'intérieur du composite (i.e., dans son contexte de

service). Le fournisseur importé n'est pas isolé dans le composite et peut donc être utilisé par d'autres consommateurs externes au composite. Les services exposés par les instances contenues dans un composite ne sont visibles qu'à l'intérieur du composite : ils ne sont pas publiés globalement (dans le registre OSGi) et ne sont donc utilisables que par d'autres instances se trouvant dans le même composite. De cette manière, iPOJO permet l'isolation de services : le contexte d'un composite est un sorte d'annuaire de services privé au composite.

Une instance de composite est valide si toutes les dépendances des services et des instances contenus dans la composition sont satisfaites. La disparition d'une instance ou d'une implémentation participant au composite invalide la composition si aucun autre service ne peut substituer le service disparu. En fonction de l'arrivé et du départ dynamiques de services, l'état d'une instance de composite peut osciller entre invalide et valide. Une instance de composite valide publie et fournit ses services. Une instance invalide n'est pas utilisable.

iPOJO propose la notion de filtre contextuel afin de gérer le dynamisme provenant du contexte. Les filtres contextuels sont reconfigurés en fonction des sources de contexte disponibles. Un gestionnaire de contexte s'enregistre à différentes sources de contexte accessibles sous la forme de services. En fonction des informations disponibles, ce gestionnaire reconfigure les dépendances de service. Cette reconfiguration porte sur l'injection de valeurs dans les filtres contextuels. Ainsi, grâce à ce filtrage contextuel, il est possible de créer des compositions réagissant au contexte. Le contexte peut donc influencer les services importés ainsi que les implémentations de service choisies, rendant le modèle de composition très flexible.

Bien que le modèle de composants proposé par iPOJO soit très intéressant pour créer des applications à services dynamiques, il présente diverses limitations notamment pour le contrôle de la composition des applications.

Tout d'abord, une spécification iPOJO a une seule interface[57], ce qui peut entraîner divers problèmes lors de la composition d'une application. Pour montrer ces problèmes, prenons comme exemple un client *S* qui veut utiliser un service fournissant *A* et *B* (voir la Figure 31). Pour ce faire, *S* a deux dépendances (indépendantes) vers les deux spécifications *A* et *B*. Une implémentation *Y* implémente ces deux spécifications. Une autre implémentation *Z* implémente seulement la spécification *B*. A l'exécution, il existe trois instances : *y1* et *y2* (avec des configurations différentes) de l'implémentation *Y*, et *z1* de l'implémentation *Z*.

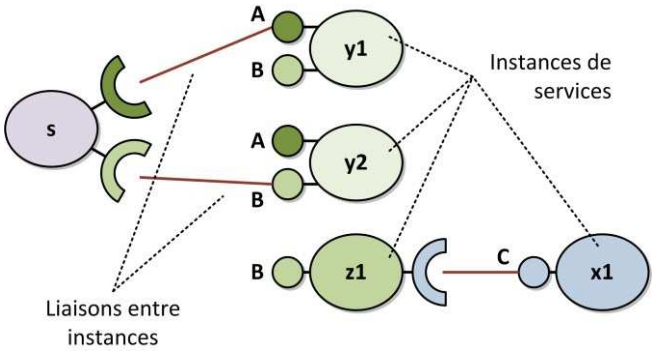


Figure 31. Exemple de résolution de dépendances dans iPOJO

Lors de la résolution des dépendances *A* et *B* de l'instance *s* du client, diverses situations peuvent se présenter :

[57] Par ailleurs, à l'exécution, les spécifications de services sont dérivées des implémentations : une spécification n'existe à l'exécution que s'il existe une implémentation qui la fournit.

- **la même instance d'une implémentation est sélectionnée :** l'instance *y1* résolvant la dépendance *A* et la dépendance *B* est liée au client *s*. Cela est la situation recherchée par le client *s* (utiliser un service fournissant les deux spécifications *A* et *B*).
- **deux instances différentes de la même implémentation sont sélectionnées :** l'instance *y1* résolvant la dépendance *A*, et l'instance *y2* résolvant la dépendance *B* sont liées au client *s* (situation illustrée dans la Figure 31).
- **deux instances d'implémentations différentes sont sélectionnées :** l'instance *y1* résolvant la dépendance *A*, et l'instance *z1* résolvant la dépendance *B* sont liées au client *s*.

Dans les deux dernières situations, bien que les dépendances de *s* soient satisfaites, le fait de choisir des instances différentes peut générer des conflits dans le fonctionnement de *s* (qui veut utiliser le même service – la même instance – satisfaisant les deux dépendances). iPOJO permet de contraindre la résolution de dépendances en utilisant de filtres LDAP qui portent sur les propriétés des fournisseurs de services (implémentations, instances). Néanmoins, cela implique la connaissance des propriétés spécifiques des fournisseurs de services. Pour résoudre le problème présenté dans l'exemple, les propriétés des implémentations ou des instances devraient être connus par le client afin d'exprimer les filtres appropriés et choisir la bonne instance satisfaisant les deux dépendances. Cependant, la connaissance des implémentations va à l'encontre du principe de l'approche à services. Une autre façon de résoudre le problème (de manière programmatique) est de chercher d'abord toutes les implémentations qui fournissent toutes les deux spécifications, et puis de créer les filtres adéquats, complexifiant ainsi le code du client.

Ensuite, afin qu'un composite iPOJO puisse fournir ses services, il doit être dans un état valide : toutes ses dépendances (internes et externes) doivent être satisfaites. Ainsi, un composite doit être complétement résolu, même si les services dont il dépend ne sont pas utilisés (invoqués). Cela représente aussi une limitation du modèle iPOJO. En effet, nous considérons qu'un composite peut fournir certains de ses services même s'il n'est pas complétement résolu. Pour ce faire, la résolution de dépendances des services du composite doit être faite à la demande (lors de l'invocation des services), entraînant la sélection d'implémentations et d'instances, l'instanciation d'implémentations, le déploiement d'implémentations et de spécifications : l'application est ainsi construite de manière incrémentale et à la demande. L'avantage de la résolution à la demande est que seul les services utilisés (invoqués) seront chargés/instanciés/liés, permettant ainsi l'optimisation des ressources.

Enfin, iPOJO ne supporte pas le déploiement dynamique de services afin de résoudre une composition.

iPOJO propose un outil minimal de développement sous la forme de plugin Eclipse[58]. Cet outil vise à faciliter le travail des développeurs en automatisant des tâches telles que la création automatique de fichiers de description de composants et la création de bundles.

3 SYNTHESE

Dans ce chapitre nous avons présenté la composition par procédés et la composition structurelle. Le choix du type de composition à utiliser est lié au contexte dans lequel la composition de services est réalisée. La composition par procédés est plus utilisée pour intégrer des services entre différentes organisations. La composition structurelle est plus utilisée quand les développeurs ont le contrôle du cycle de vie des services composés. La granularité des services peut être aussi considérée comme un critère de sélection du type de composition à utiliser : les services à gros grain sont composés avec une vision plus centrée sur le métier ce qui favorise une composition par procédés ; les services à grain fin correspondent à une vision plus programmatique et une composition structurelle est plus pertinente.

[58] **http://cwiki.apache.org/confluence/display/FELIX/iPOJO+Eclipse+Plug-in**

L'utilisation combinée des deux types de composition est possible, comme dans SCA, permettant ainsi la description structurelle de compositions où les implémentations des composants peuvent être réalisées par composition de procédés.

Nous nous intéressons particulièrement à la composition structurelle de composants à services en termes de spécifications de services. Les approches actuelles ne satisfont pas tous les besoins de la composition dynamique d'applications : définition simple, flexibilité, construction incrémentale, déterminisme, opportunisme, vérification de la conformité et de la cohérence, gestion du partage et de la visibilité, adaptation, évolution.

Cette thèse propose ainsi une solution pour la composition dynamique d'applications qui répond à ces besoins. Cette solution combine des concepts de l'approche à composants, de l'approche à services, de l'architecture logicielle et de l'ingénierie dirigée par les modèles dans le but de proposer un modèle à composants à services pour la conception, le développement et l'exécution d'applications à services dynamiques.

De nombreux acteurs peuvent intervenir dans le processus de réalisation d'une application à services, de sa conception jusqu'à son exécution. Les différents acteurs effectuent un ensemble de tâches qui peuvent être assez complexes : spécification de services, réalisation d'implémentations, conception de la composition, déploiement, etc. Nous soulignons ainsi le besoin d'environnements logiciels pour faciliter le processus de composition de services, de la conception à l'exécution, en fournissant un ensemble d'outils spécialisés : éditeurs, validateurs, compilateur, débogueur, interpréteur, support d'exécution.

DEUXIEME PARTIE :

CONTRIBUTIONS

CHAPITRE 5
PROPOSITION

Nous proposons une approche dirigée par les modèles pour la construction d'applications flexibles et dynamiques par composition de services. Dans ce premier chapitre de la deuxième partie, nous présentons tout d'abord la problématique liée à la construction d'applications à services. Après d'exposer les objectifs de cette thèse, nous présentons une vision globale de notre approche en mettant en évidence les aspects considérés pour sa mise en place, et en détaillant les trois parties que la constituent : conception, développement et exécution. Les détails sur notre approche sont présentés dans les chapitres suivants.

1 PROBLEMATIQUE

Dans les chapitres précédents nous nous sommes aperçus que l'approche à services offre des concepts particulièrement intéressants pour la construction d'applications flexibles et dynamiques. Toutefois, nous avons identifié des besoins et des défis majeurs posés par les applications modernes, telles que les applications ubiquitaires et mobiles. Concrètement, nous avons fait les constatations suivantes :

- **les applications à services modernes doivent s'exécuter, de plus en plus, dans des contextes ouverts, dynamiques, hétérogènes et distribués.** Dans de tels contextes aux caractéristiques techniques très différentes, la disponibilité des services et des dispositifs, les préférences et la localisation des utilisateurs peuvent varier de manière imprévisible au cours de l'exécution des applications.
- **la variabilité des contextes d'exécution rend complexes la conception, le développement, l'exécution et le contrôle des applications à services.** En effet, il est difficile, voire impossible, de connaître au moment de la conception et du développement d'une application, les dispositifs qui seront disponibles à l'exécution, les conditions précises dans lesquelles l'application sera utilisée et encore moins les réactions appropriées pour l'adapter à telles conditions.
- dû à la variabilité des contextes d'exécution, **l'architecture d'une application à services ne peut plus être complètement figée avant son exécution.** Elle doit être définie à haut niveau d'abstraction, par des propriétés et des contraintes que les services participant à la composition de l'application doivent satisfaire, afin de rendre cette définition indépendante des contextes d'exécution.
- **les applications à services doivent pouvoir être composées graduellement avant et pendant leur exécution.** Les services concrets participant à la composition d'une application doivent pouvoir être définis et assemblés lors du développement de l'application (composition statique), mais aussi pendant son exécution (composition dynamique) afin de considérer le contexte d'exécution courant.
- **la conformité et la cohérence du développement et de l'exécution d'une application vis-à-vis de son architecture de conception sont difficiles à vérifier et à garantir**. La discontinuité existante entre la conception, le développement et l'exécution d'une application rend difficile de vérifier et de garantir la conformité et la cohérence de son implémentation et de son exécution. La définition abstraite d'une application doit donc être utilisée comme cadre structurel et sémantique afin de guider l'implémentation et l'exécution de l'application et d'assurer ainsi la conformité et la cohérence.
- **les applications à services doivent pouvoir s'adapter dynamiquement aux contextes d'exécution.** Les applications doivent de plus en plus être disponibles de façon quasiment ininterrompue. Afin de faire face à la variabilité de leur contexte d'exécution, elles doivent s'adapter en cours d'exécution et en conformité et cohérence avec leur définition abstraite.
- **les applications à services doivent pouvoir évoluer dynamiquement** afin de corriger, améliorer, étendre ou réduire leurs fonctionnalités. La définition abstraite d'une application doit être connue/préservée à l'exécution non seulement pour permettre de guider la composition dynamique de l'application, mais aussi pour pouvoir être modifiée et permettre ainsi l'évolution dynamique de l'application.
- **diverses approches abordent la construction d'applications à services sous différentes perspectives.** Certaines approches proposent des modèles à composants pour le développement de services et/ou pour sa composition. Quelques-unes proposent des langages pour la description de l'architecture des applications. D'autres offrent des infrastructures pour supporter leur exécution. Cependant, en général, les approches supportant la composition ne supportent pas le dynamisme : la composition est statique empêchant toute sélection et substitution de composants à l'exécution. Inversement, les approches proposant des mécanismes de gestion du dynamisme ne proposent pas de mécanismes de composition et de contrôle d'applications ou ils restent limités.

2 OBJECTIFS

En considérant les problèmes précédemment mentionnés, notre principal objectif est de **faciliter la construction d'applications flexibles et dynamiques à base de services** en couvrant leur cycle de vie, de leur conception à leur exécution. De manière plus détaillée, nos objectifs sont les suivants :

- proposer un formalisme permettant de **définir une application à services en intention**, à partir de ses propriétés, ses contraintes et ses préférences de composition. Ce formalisme doit permettre de spécifier un cadre abstrait qui permette de guider la composition de l'application.
- proposer des formalismes et des mécanismes permettant de **réaliser la composition d'une application graduellement avant et pendant l'exécution de l'application**. Les mécanismes de composition, présents tout au long du cycle de vie des applications, doivent vérifier et garantir la conformité et la cohérence de la composition des applications.
- proposer des formalismes et des mécanismes de **déploiement et d'exécution** qui permettent et facilitent la mise à disposition, l'installation, l'activation, la désactivation et la désinstallation d'applications.
- proposer un formalisme permettant de **représenter, de façon homogène, l'état d'exécution actuel d'une application à services**. Ce formalisme doit permettre de contrôler l'état d'exécution de l'application en conformité et cohérence avec sa définition.
- **fournir des environnements et des outils liés à chaque phase du cycle de vie des applications**. En particulier, nous considérons des environnements et des outils pour la conception, le développement, le déploiement de services et l'exécution d'applications à services. De tels environnements doivent être extensibles afin d'ajouter, par exemple, des mécanismes supportant la définition et/ou la gestion de propriétés orthogonales à la fonctionnalité métier des applications.

Dans la section suivante, nous présentons notre approche mise en place pour répondre à ces objectifs.

3 NOTRE APPROCHE

Pour mettre en place notre approche, nous prenons en compte les suivants besoins :

1. Définir une application à services par une liste exhaustive de tous ses composants est contraignant, inadéquat, voire même impossible pour un certain nombre de raisons, parmi lesquelles :
    - certains services requis par l'application peuvent ne pas exister à un moment donné dans un ou plusieurs espaces de sélection disponibles (dépôts locaux/distants de services),
    - certains services requis par l'application peuvent exister mais être indisponibles à un moment donné,
    - l'application peut requérir/préférer d'utiliser en priorité les services disponibles dans la plate-forme d'exécution (mode opportuniste).

2. La définition d'une application à services doit considérer la nature dynamique des services qui participent ou qui peuvent participer à sa composition :
    - certains services requis par l'application peuvent être disponibles à tout moment lors de l'exécution de l'application,
    - certains services utilisés par l'application peuvent devenir indisponibles ou inadéquats à tout moment lors de l'exécution de l'application et devraient être substitués afin de garantir l'exécution de l'application (adaptation dynamique).

Ainsi, en considérant ces besoins, nous souhaitons offrir la capacité de définir une application à services de manière abstraite, et de réaliser sa composition concrète de manière graduelle, avant et/ou pendant la phase d'exécution, en conformité avec sa définition. Pour ce faire, nous considérons les problèmes suivants, d'un point de vue *top-down* (de la conception à l'exécution) :

- Comment définir une application/service de manière abstraite ? La définition doit être suffisamment riche afin d'avoir l'information nécessaire à son développement et son exécution, mais suffisamment abstraite afin de permettre son exécution dans une vaste gamme de contextes d'exécution différents.
- Comment concrétiser un service abstrait ? Comment assurer la conformité et la cohérence de la concrétisation d'un service ?
- Comment développer et exécuter une application définie de manière abstraite ? Comment assurer la conformité et la cohérence du développement et de l'exécution d'une application par rapport à sa définition abstraite ?

D'un point de vue *bottom-up*, nous aimerions éliminer les contraintes et les défauts du protocole d'interaction de l'approche à services auxquels l'exécution d'applications à services est confrontée. En prenant pour base de comparaison la plate-forme à services OSGi, nous observons les problèmes suivants :

- Le niveau d'abstraction des services est très bas : un service est définit comme un objet qui implémente une interface (Java). Il faut définir des concepts abstraits qui permettent de manipuler facilement services et applications, et qui puissent être utilisés également lors de la conception et du développement.
- La sélection de services est basée sur des filtres portant sur des propriétés qui ne sont pas déclarées : un client peut définir des contraintes de sélection portant sur des propriétés que les fournisseurs n'ont pas. La validité des contraintes de sélection devrait être assurée afin d'éviter l'échec de la sélection.
- Le concept de dépendance est faible : une dépendance est définie vers une seule interface. Cela peut provoquer des ambiguïtés dans la sélection de fournisseurs de services et donner lieu à des sélections incohérentes. La cohérence de la sélection des services requis par un client doit être garantie.
- La composition (le choix d'un fournisseur) est à la charge des clients indépendamment de l'application à laquelle ils appartiennent : il n'y a pas de vue globale de l'architecture d'une application. La composition de services doit dépendre de l'application et de son contexte (de ses contraintes et préférences), elle doit être gérée et garantie d'un point de vue global.

Des points mentionnés ci-dessus, nous retenons que la construction d'applications à services, que ce soit de façon *top-down* ou *bottom-up*, a besoin de formalismes et de mécanismes qui supportent l'abstraction, la concrétisation et la catégorisation de services. Ces formalismes et mécanismes devraient être appliqués de manière homogène tout au long du cycle de vie des applications, afin d'intégrer aussi bien l'approche *top-down* que *bottom-up*. Notre approche propose ainsi **la même démarche d'abstraction, de concrétisation, de catégorisation et de composition de services avant et durant la phase d'exécution des applications**.

Notre approche s'appuie sur les principes de l'approche à composants à services, présentée dans le Chapitre 2 (cf. section 1.3), qui propose d'utiliser des composants pour implémenter des services. Pour la conception et le développement d'applications, notre approche suit une vision orientée composants, permettant l'encapsulation, la composition et la réutilisation de composants à services ainsi que la description globale de la structure des applications. Pour l'exécution d'applications, notre approche suit une vision orientée services afin de permettre la liaison tardive, le dynamisme des services, et une vision orientée composants afin de contrôler leur composition dynamique. Notre approche concilie donc l'approche à composants et l'approche à services en bénéficiant ainsi de leurs avantages.

Notre approche définit des métamodèles à composants pour la conception, le développement et l'exécution de services primitifs et composites ; **une application à services est un service composite**. Pour couvrir le cycle de vie des applications à services, nos métamodèles – chacun correspondant à une phase différente du cycle de vie – possèdent des relations entre eux qui déterminent la façon de passer d'une phase à une autre ainsi que les informations (les concepts) partagées/transférées entre les phases. Dans la phase d'exécution par exemple, des informations produites dans les phases de conception et de développement de services et d'applications sont connues/préservées afin de pouvoir contrôler l'exécution des applications.

Notre approche est structurée en trois phases, comme illustrée dans la Figure 32 :

1. **La conception d'une application à services.** Une application est définie en intention par des services abstraits liés entre eux, et par des propriétés, des contraintes et des préférences indiquant les règles de composition de l'application. Une définition en intention spécifie ainsi un cadre abstrait qui permet de guider/contrôler la composition d'une application lors de son développement et de son exécution.
2. **Le développement et la configuration d'une application à services.** Une application est développée et configurée (composée) manuellement et/ou automatiquement, de façon partielle ou complète, en fonction de sa définition en intention et des services disponibles dans un ou plusieurs dépôts de composants à services. Cette phase permet ainsi de définir une application en extension. La définition en intention et en extension d'une application constituent le **modèle de l'application**.
3. **L'exécution d'une application à services.** Une application est déployée et exécutée sur l'environnement d'exécution en conformité avec son modèle d'application. L'exécution de l'application inclue sa composition (la résolution de ses services requis), son adaptation (la reconfiguration de son architecture suite à la variabilité de son contexte d'exécution) et son évolution (la modification de son modèle d'application afin de corriger, améliorer, étendre ou réduire ses fonctionnalités).

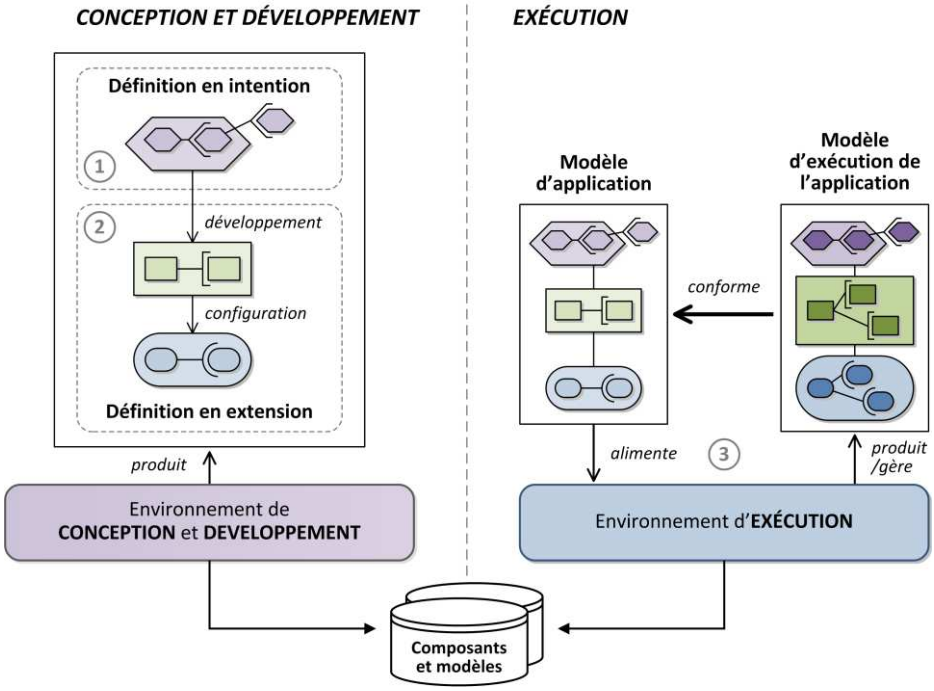


Figure 32. Vision globale de notre approche

Les détails de chacune de ces phases sont présentés dans les sections suivantes.

3.1 PHASE DE CONCEPTION

Dans cette phase, une application à services est définie indépendamment des services concrets participant à sa composition afin de lui apporter la flexibilité nécessaire pour faire face à des environnements d'exécution dynamiques et peu prévisibles.

L'abstraction est un principe usuel de l'ingénierie dirigée par les modèles qui permet de décrire une application selon un ou plusieurs points de vue. Chaque point de vue correspond à un modèle représentant un aspect de l'application à un certain niveau d'abstraction. L'approche MDA (*Model Driven Architecture*) par exemple préconise de modéliser une application indépendamment des technologies et des plates-formes d'exécution utilisées, offrant ainsi l'avantage de la pérennité du modèle face aux évolutions technologiques, à l'hétérogénéité et au dynamisme des environnements d'exécution. Dans le cas du MDA, le principe d'abstraction permet de séparer les aspects indépendants des aspects spécifiques de la technologie utilisée pour l'implémentation d'une application.

Le principe d'abstraction vise à réduire la complexité de la conception d'applications en fournissant des représentations de haut niveau pérennes, réutilisables et compréhensibles par les acteurs intervenant dans le processus de construction des applications. De ce fait, nous proposons de **définir une application à services de manière abstraite**.

La définition abstraite d'une application à services impose d'augmenter le niveau d'abstraction des services. Pour ce faire, nous proposons un mécanisme générique qui permet la définition d'éléments à différents niveaux d'abstraction ainsi que leur catégorisation. Ce mécanisme vise à couvrir les besoins d'abstraction et de concrétisation des éléments à travers le concept de **groupe d'équivalence**. Les détails sur les groupes d'équivalence sont présentés dans la section 2.2 du Chapitre 6. Sur la base du mécanisme de groupes d'équivalence, nous définissons le concept de composant à services à différents niveaux d'abstraction : **spécification**, **implémentation** et **instance**, ainsi que des catégories (i.e., des groupes) entre ces concepts.

Dans notre approche, un composant à services fournit des services (des interfaces de services) et des messages et requiert des services et des messages. Nous généralisons les concepts d'interface et de message par le concept de ressource : interfaces et messages sont des types de ressources (toutefois, notre approche étant extensible, d'autres types de ressources peuvent être déclarés, des événements par exemple). Par abus de langage, nous dirons par la suite « service » pour « composant à services ».

La notion la plus abstraite du concept de service est celle de spécification de service, représentée graphiquement dans la Figure 33 :

> Une **spécification** définit un service par ses ressources fournies (interfaces, messages), ses ressources requises, ses propriétés, ses contraintes et ses préférences.

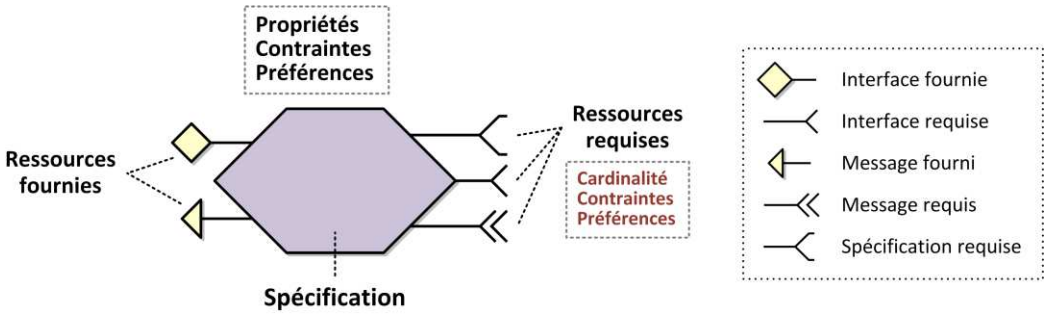


Figure 33. Spécification de service

Une spécification définit un contrat de structure et de comportement qui doit être rempli par les fournisseurs de services qui l'implémentent : différentes implémentations d'une spécification sont possibles. Une spécification peut être utilisée par des consommateurs de services afin de déclarer des

dépendances vers l'ensemble des ressources fournies par la spécification. Les spécifications sont ainsi des objets de premier niveau utilisables et composables qui permettent la définition abstraite d'applications.

Nous définissons le concept de définition abstraite d'une application, nommé **définition en intention**, par le concept de spécification composite de service illustré dans la Figure 34 :

> Une **spécification composite** définit une application ou un sous-système. En tant que spécification, elle définit les ressources fournies, les ressources requises, les propriétés, les contraintes et les préférences de l'application ou du sous-système. En tant que composite, elle définit les spécifications contenues (primitives ou composites), les interactions entre elles (connecteurs), et les règles (propriétés, contraintes et préférences contextuelles) qui permettront de guider/contrôler la composition concrète de l'application ou du sous-système.

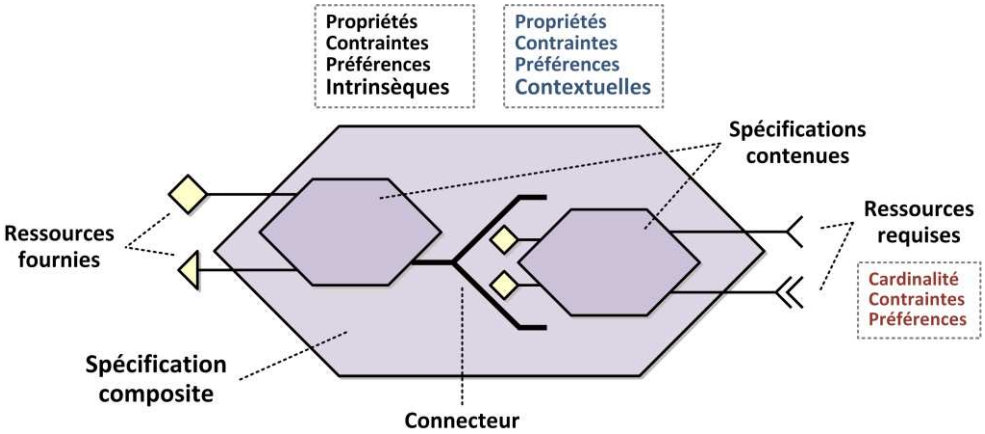


Figure 34. Spécification composite de service

Le concept de spécification composite étend ainsi le concept de spécification en ajoutant les caractéristiques, structurelles et sémantiques, propres à une composition de services. La définition en intention d'une application définit les propriétés qui permettront de guider la composition de l'application (des contraintes et des préférences de sélection de services) et de la contrôler (des propriétés de partage et de visibilité des services propres à l'application).

Nous avons défini les concepts de spécification primitive et composite de service par des langages de modélisation, permettant ainsi de vérifier et d'assurer leur cohérence et leur validité. Ces langages sont spécifiés par les métamodèles présentés dans les sections 3.1 et 3.2 du chapitre suivant. En outre, les concepts de spécification primitive et composite étant définis formellement, les règles de composition définies dans une spécification composite (portant sur les propriétés des spécifications), peuvent être vérifiées statiquement afin de garantir leur validité, et celle de la spécification composite.

Pour assister cette phase, nous proposons un environnement spécialisé (intégré à Eclipse) qui permet de définir en intention des applications à services par la création de spécifications primitives et composites.

En conclusion, le concept de définition en intention permet de définir des applications à services avec un haut niveau de flexibilité. Une définition en intention spécifie ainsi un cadre abstrait pour la composition concrète d'une application ou d'un ensemble (famille) d'applications. La flexibilité d'une application peut être diminuée lors des phases de développement et de configuration par la concrétisation de ses services abstraits et/ou l'ajout des nouvelles règles de composition, ou préservée jusqu'à la phase d'exécution.

3.2 PHASE DE DEVELOPPEMENT

Dans cette phase une application définie en intention est composée (développée et configurée). Cette composition, en termes d'implémentations et d'instances de services, est guidée/contrôlée par la définition en intention de l'application.

La composition d'une application est basée sur la résolution des services abstraits indiqués dans sa définition en intention (i.e., la concrétisation des spécifications de services). Résoudre un service abstrait consiste ainsi à sélectionner ou à créer un ou plusieurs services plus concrets qui satisfont un ensemble de contraintes et de préférences données. Une spécification est donc résolue par une ou plusieurs implémentations qui implémentent cette spécification. Nous définissons le concept d'implémentation de service comme suit (voir schéma dans la Figure 35) :

> Une **implémentation** est un programme exécutable qui implémente une spécification en utilisant une technologie d'implantation à services particulière (Services Web, OSGi ou iPOJO, par exemple). Une implémentation possède au moins les caractéristiques définies par sa spécification : ressources fournies, ressources requises, propriétés, contraintes et préférences, et peut posséder des caractéristiques additionnelles.
> La relation *implements* qui relie une implémentation à une spécification permet de vérifier et d'assurer la conformité et la cohérence de l'implémentation vis-à-vis de la spécification.

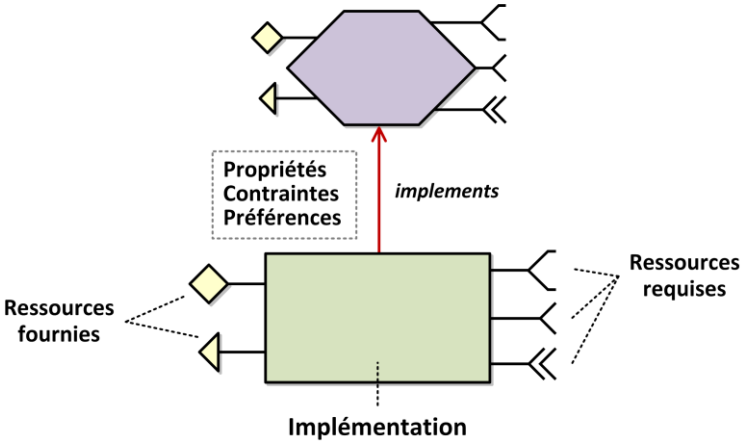


Figure 35. Implémentation de service

Une implémentation est résolue par une ou plusieurs instances qui instancient cette implémentation. Nous définissons le concept d'instance de service comme suit (voir schéma dans la Figure 36) :

> Une **instance** d'une implémentation est une instance de la classe associée à cette implémentation. Une instance possède les caractéristiques définies par son implémentation : ressources fournies, ressources requises, propriétés, contraintes et préférences.
> La relation *instantiates* qui relie une instance à son implémentation permet de vérifier et de garantir la conformité et la cohérence de l'instance par rapport à l'implémentation.

Le concept d'instance est un concept d'exécution. En effet, les instances sont les services concrets (i.e., des objets de classes) utilisés pour composer des applications sur une plate-forme d'exécution. Toutefois, dans la phase de développement, une instance correspond à la déclaration d'une configuration particulière d'une implémentation : une instance est une **déclaration d'instance**.

Une déclaration d'instance définit les valeurs initiales des propriétés configurables définies par son implémentation (i.e., les valeurs avec lesquelles l'instance sera créée lors de la phase d'exécution).

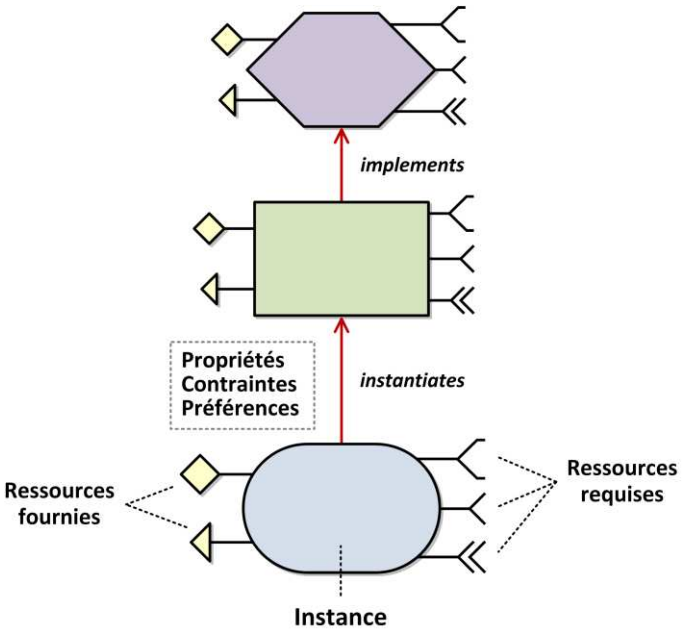


Figure 36. Instance de service

Ainsi, dans cette phase, la composition d'une application permet d'obtenir une architecture en termes d'implémentations, appelée **implémentation composite**, et une architecture en termes de déclarations d'instances, appelée **instance composite**, conformes à la définition en intention. Les concepts d'implémentation composite et d'instance composite de service sont définis comme suit :

> Une **implémentation composite** représente une application ou un sous-système en termes d'implémentations de services. En tant qu'implémentation, elle implémente une spécification (primitive ou composite) et possède au moins les caractéristiques définies par celle-ci. En tant que composite, elle contient un ensemble d'implémentations (primitives ou composites), des connecteurs entre elles, et des règles (propriétés, contraintes et préférences contextuelles) pour guider/contrôler la création de ses instances composites.
>
> Une **instance composite** représente une application ou un sous-système en termes d'instances de services. En tant qu'instance, elle instancie une implémentation (composite), et possède les caractéristiques définies par son implémentation. En tant que composite, elle contient un ensemble de déclarations de services (primitives ou composites) et de connecteurs entre elles.

Ces concepts, schématisés dans la Figure 37, sont définis par les métamodèles présentés dans les sections 3.1 et 3.2 du chapitre suivant.

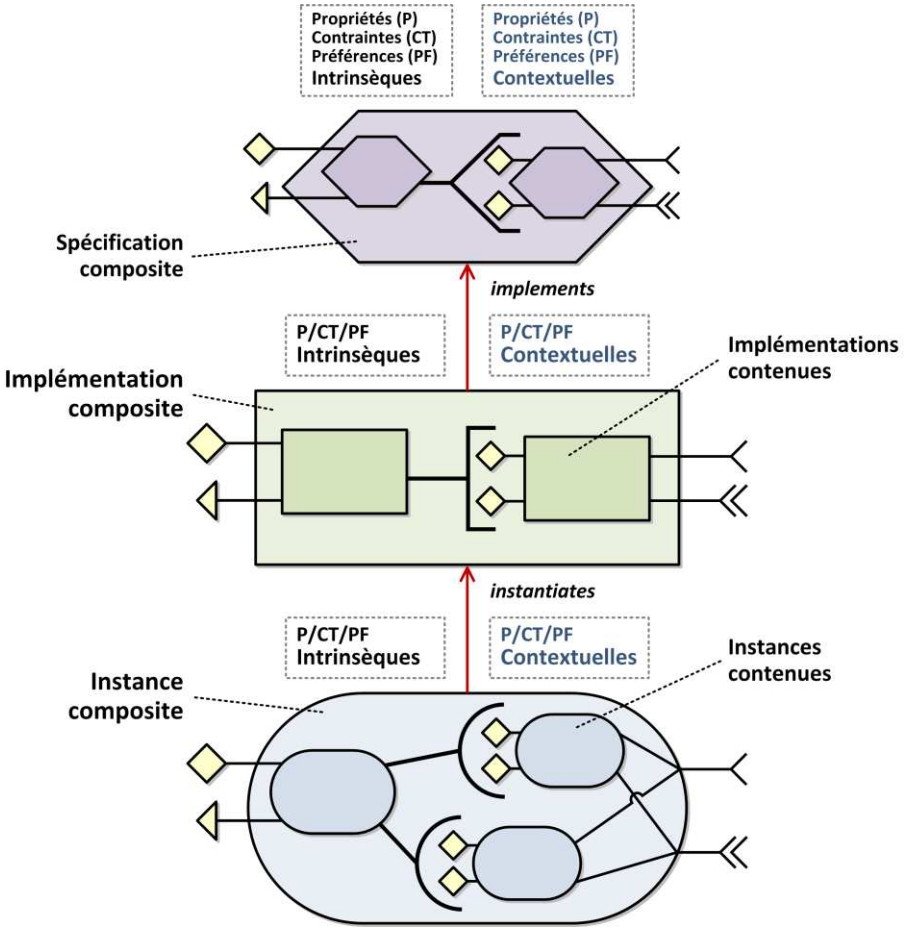


Figure 37. Implémentation composite et Instance composite de services

Notre approche permet de réaliser la composition d'une application de manière incrémentale à travers la résolution de ses services. Afin de résoudre un service, il faut définir (1) comment le résoudre (i.e., jusqu'à quel niveau de concrétisation), (2) quand le résoudre (i.e., quels sont les déclencheurs de la résolution), et (3) où le sélectionner (i.e., quels sont les dépôts considérés pour la sélection). Dans la phase de développement :

(1) La résolution d'un service (appelée résolution statique) est réalisée par défaut de façon **partielle**, i.e., à un niveau de concrétisation. Toutefois, elle peut être définie comme :
- **complète** afin de résoudre le service récursivement jusqu'à sa concrétisation totale (le résultat de la résolution complète d'une spécification est une ou plusieurs déclarations d'instance), et/ou
- **étendue** afin de résoudre aussi les dépendances des services.

(2) La résolution est réalisée automatiquement de façon **immédiate** pour les services définis comme statiques.

(3) Divers dépôts de composants, appelés **espaces de sélection**, peuvent être considérés pour la résolution de services. La résolution peut ainsi être :
- **fermée** afin de considérer seulement les composants présents dans l'espace de travail local (*workspace*), et/ou
- **coopérative** afin de tenir compte des composants mis à disposition par des organisations tierces dans divers dépôts distants.

Une implémentation composite, réalisée dans la phase de développement, représente **l'architecture statique de l'application en termes d'implémentations**, et une instance composite représente **l'architecture statique de l'application en termes d'instances**. Ces composites, représentent ainsi la **définition en extension** de l'application. Ils peuvent être incomplets : ils peuvent ne pas contenir tous les composants requis et garder ainsi un certain niveau d'abstraction et de flexibilité afin de supporter la composition de l'application pendant son exécution. En effet, la variabilité du contexte d'exécution impose que certaines parties de l'application soient déterminées durant son exécution (par exemple si leur sélection dépend de propriétés liées à l'exécution). De ce fait, en plus de la composition statique, notre approche permet et surtout promeut la **composition dynamique** des applications au cours de leur exécution. La composition dynamique peut diminuer les probabilités d'échouer d'une application, par exemple, si un service sélectionné et lié statiquement au développement n'est pas disponible à l'exécution.

Pour permettre la composition dynamique des applications notre approche permet de garder un haut niveau d'abstraction jusqu'à la phase d'exécution. Pour ce faire, nous définissons le concept de modèle d'application (schématisé dans la Figure 38) :

> Un **modèle d'application** est une entité exécutable constituée d'une spécification composite, représentant la définition en intention d'une application, d'une implémentation composite et d'une instance composite, représentant la définition en extension de l'application.

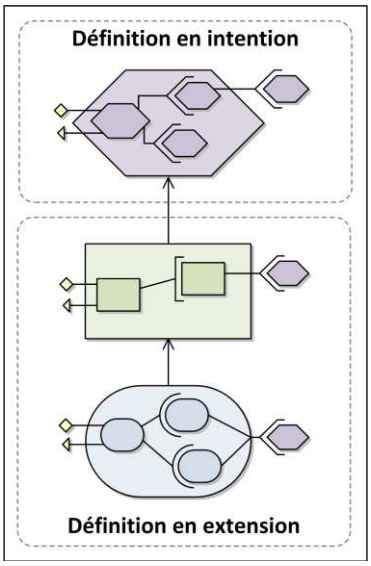


Figure 38. Modèle d'application

Cette phase est assistée par un environnement spécialisé (intégré à Eclipse) et par des outils pour le développement et la configuration d'applications. Ils permettent de construire, à partir d'une définition en intention, les architectures statiques d'une application en spécifiant manuellement les composants concrets contenus dans l'application, et/ou en sélectionnant automatiquement certains composants de l'application en considérant les composants contenus dans l'espace de travail et/ou dans des dépôts de composants distants. L'environnement de développement gère le packaging des composants et le placement des packages dans différents dépôts de composants (locaux ou distants).

En conclusion, à travers les concepts d'implémentation et d'instance composite, notre approche permet de définir des applications ayant différents niveaux de flexibilité allant des applications complètement statiques jusqu'à des applications complètement dynamiques.

3.3 PHASE D'EXECUTION

L'exécution d'une application est basée sur la sélection, le déploiement, l'activation et la connexion des services selon les informations fournies dans son modèle d'application.

Le processus de résolution de services dans la phase d'exécution (résolution dynamique) suit la même démarche que le processus de résolution réalisé dans la phase de développement (résolution statique) : résoudre un service abstrait consiste à sélectionner ou à créer un ou plusieurs services plus concrets qui satisfont un ensemble de contraintes et préférences données. Comment nous l'avons mentionné dans la section précédente, la résolution définit (1) comment résoudre, (2) quand résoudre, et (3) où sélectionner. Dans la phase d'exécution :

(1) La résolution d'un service est réalisée de façon **complète** : le résultat de la résolution d'un service est une instance sélectionnée ou créée.

(2) La résolution d'un service est réalisée suite à l'occurrence d'un événement particulier. Nous définissons les types de résolution suivants qui permettent d'établir quand la résolution d'un service doit être réalisée :
- **la résolution paresseuse** : la résolution d'un service est réalisée quand ce service est demandé.
- **la résolution immédiate** : la résolution d'un service est effectuée lors du déploiement de l'application sur la plate-forme d'exécution.
- **la résolution adaptative** : la résolution d'un service est réalisée suite à une variation du contexte d'exécution, par exemple, suite à la disponibilité d'un service qui valide la conformité de l'exécution de l'application, ou suite à l'indisponibilité ou à la modification des propriétés d'un service qui invalide la conformité de l'application.

(3) Lors de la résolution d'un service, en plus des services disponibles dans des dépôts locaux et distants, les services disponibles dans la plate-forme d'exécution peuvent aussi être considérés. Ainsi, outre d'être fermée ou coopérative, la résolution peut être **opportuniste** afin de tenir compte des services disponibles dans la plate-forme d'exécution. La résolution dynamique fermée ou coopérative entraîne le déploiement et l'activation immédiats des services sélectionnés. Dans tous les cas, la résolution dynamique entraîne la connexion immédiate des services.

Nous adoptons par défaut une résolution dynamique paresseuse et opportuniste. En privilégiant la résolution paresseuse et opportuniste, l'utilisation de services et de ressources est optimisée. En outre, en considérant la résolution adaptative des services, le risque d'échec d'une application (lié à la variabilité du contexte d'exécution) est diminué.

Dans le Tableau 7, nous résumons les différents modes de résolution gérés par notre approche. Les modes de résolution font partie des règles de composition définis dans la spécification composite, et/ou dans l'implémentation composite d'une application.

	Comment résoudre	***Quand résoudre***	***Où sélectionner***	***Quand activer***
Statique	*Simple* *Complète* *Étendue* *Complète étendue*	*Immédiate*	*Fermée* *Coopérative*	*Paresseuse* *Immédiate*
Dynamique	*Complète*	*Paresseuse* *Immédiate* *Adaptative*	*Fermée* *Coopérative* *Opportuniste*	*Immédiate*

Tableau 7. Modes de résolution pour la composition d'une application

Pour gérer l'exécution d'une application en conformité avec son modèle d'application, nous disposons d'un modèle d'exécution, appelé **modèle d'état**, qui représente l'état d'exécution actuel d'une application. Ce modèle représente les architectures d'exécution de l'application en termes de spécifications (spécification composite), d'implémentations (implémentation composite) et d'instances de services (instance composite). Connecté causalement à l'application en exécution, le modèle d'état permet de surveiller l'application et de réaliser sa composition et son adaptation en conformité avec son modèle d'application.

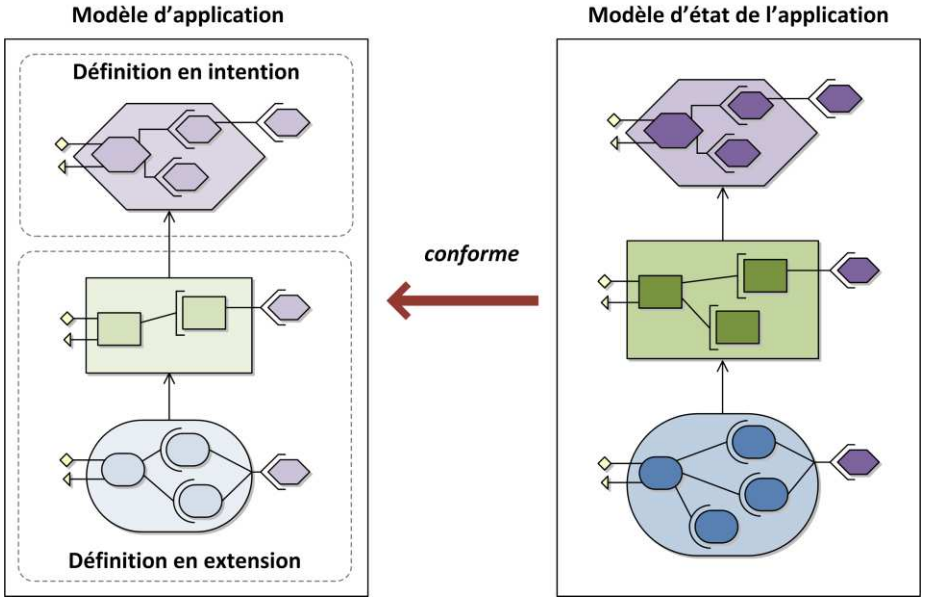


Figure 39. Modèle d'état

En outre, notre approche permet l'évolution dynamique d'une application, par la modification de son modèle d'application, afin de corriger, améliorer, étendre ou réduire ses fonctionnalités. L'évolution de l'application implique la vérification de la conformité du modèle d'état, pouvant entraîner l'adaptation de l'application.

Pour assister cette phase, nous avons réalisé une machine d'exécution d'applications à services. Elle est composée de deux composants principaux : une plate-forme générique et extensible d'exécution d'applications à services : APAM (*APplication Abstract Machine*), et un interpréteur de modèles d'application : COMPASS-RT.

COMPASS-RT, interprète des modèles d'application afin de les exécuter et de gérer leurs exécutions. Il gère la composition dynamique d'une application tout en assurant sa conformité par rapport à son modèle d'application. Suite à des modifications du contexte d'exécution de l'application, COMPASS-RT évalue la conformité et la cohérence de l'état d'exécution actuel de l'application (son modèle d'état) vis-à-vis du modèle d'application. Face à un état d'exécution non-conforme, COMPASS-RT recalcule la composition afin de l'adapter par des reconfigurations architecturales. COMPASS-RT permet également l'évolution dynamique d'une application à travers la modification de son modèle d'application.

COMPASS-RT repose sur la plate-forme APAM, qui fournit des mécanismes de base pour l'exécution et la gestion d'applications contrôlant des propriétés telles que la visibilité et le partage de composants intra- et inter-applications. COMPASS-RT étend les mécanismes de base fournis par APAM et peut collaborer (voire être étendu) avec d'autres environnements d'exécution qui gèrent des préoccupations orthogonales à l'application telles que la distribution, la sécurité ou la performance. L'exécution d'une application et de ses différentes préoccupations est ainsi contrôlée par un ensemble d'environnements spécifiques et indépendants.

4 SYNTHESE

Notre approche, structurée en trois phases, considère les besoins de flexibilité, de dynamisme et d'adaptation des applications. Nous estimons que notre approche est une contribution au domaine des applications dynamiques en apportant les propriétés suivantes :

- **Une application peut être définie en intention, en extension ou les deux.** Certaines parties de la définition d'une application peuvent rester abstraites jusqu'à son exécution, donnant plus de flexibilité à l'application face à des environnements dynamiques et peu prévisibles. Notre approche permet ainsi la définition d'applications dynamiques, statiques et mixtes (semi-statiques et semi-dynamiques).
- **La définition d'une application est validée statiquement.** Les propriétés, les contraintes et les préférences contextuelles d'une application portent sur les propriétés de ses services abstraits. Le concept de service abstrait étant formellement défini, la validité de la définition d'une application peut être vérifiée statiquement.
- **La composition d'une application peut être réalisée avant ou pendant la phase d'exécution.** Une application peut être composée graduellement avant son exécution : composition statique ; et/ou pendant son exécution : composition dynamique. De plus, la composition (statique ou dynamique) d'une application peut être réalisée selon des modes de résolution différents (voir le Tableau 7).
- **La conformité de la composition d'une application est garantie.** La composition statique d'une application est guidée par sa définition en intention afin de garantir la conformité de son développement et de sa configuration. La composition dynamique d'une application est guidée par le modèle de l'application (i.e., par les définitions en intention et en extension) afin de garantir la conformité de son exécution.
- **La protection d'une application est garantie.** Une application peut spécifier les règles de partage et de visibilité de ses services (implémentations et instances). Les composites représentant l'architecture d'une application en termes d'implémentations (implémentation composite), et instances (instance composite), sont des mécanismes d'encapsulation qui permettent la protection des services qu'ils contiennent.

Grâce à ces propriétés, notre approche rend plus simple et plus sûre la construction d'applications à services, de leur conception à leur exécution.

Nous avons mis en place notre approche en suivant une approche dirigée par les modèles. Cela nous a amené à établir différents métamodèles définissant de façon précise la sémantique associée aux notions de services et d'architectures à services. Le but du chapitre suivant est de présenter plus en détail les trois phases constituant notre approche, les mécanismes, les métamodèles et les environnements associés à chaque phase.

CHAPITRE 6
MODELE DE COMPOSANTS A SERVICES

Ce chapitre détaille la mise en place de notre approche introduite dans le chapitre précédent. Dans une première partie, nous présentons les principes et les mécanismes sur lesquels notre approche est fondée, à savoir le mécanisme de groupes d'équivalence et le principe de résolution. Nous décrivons, dans une deuxième partie, le métamodèle de composants à services proposé qui définit les concepts nécessaires pour la construction d'applications, en couvrant leur cycle de vie, de leur conception jusqu'à leur exécution. Nous présentons ce métamodèle en deux temps : d'abord, nous détaillons les concepts pour la construction de services primitifs, et puis nous décrivons les concepts pour la construction de services composites. Dans une troisième partie, nous détaillons les aspects des phases du cycle de vie des applications construites en utilisant notre approche.

1 SCENARIO D'APPLICATION

Tout d'abord, nous présentons un exemple d'application pervasive sur lequel nous nous appuierons tout au long de ce chapitre afin d'illustrer les différentes phases de notre proposition. Il s'agit d'une application de gestion multimédia pour des passerelles résidentielles qui permet aux utilisateurs de naviguer, de sélectionner et de reproduire des fichiers multimédia (image, son, vidéo) en utilisant les différents équipements électroniques audio et vidéo présents dans la maison (téléviseurs, haut-parleurs, écrans, téléphones portables, tablettes, etc.).

L'application est composée d'un service de gestion de médias qui interagit, d'une part, avec un certain nombre de bibliothèques qui contiennent des fichiers audio et vidéo (par exemple, des serveurs audio et/ou vidéo à la demande ou des serveurs domestiques), et d'autre part avec un service de lecture de médias qui contrôle la reproduction d'un fichier donné sur un des dispositifs disponibles dans la maison (voir la Figure 40).

Les bibliothèques de médias peuvent être disponibles dynamiquement dans la passerelle résidentielle. Quand une nouvelle bibliothèque est disponible, elle doit rejoindre automatiquement l'application de gestion multimédia (elle doit être connectée au service de gestion de médias) permettant ainsi d'accéder à ses fichiers contenus.

L'application doit ainsi utiliser les services et les dispositifs disponibles dans la maison, mais aussi ceux qui sont les mieux adaptés à un moment donné (par exemple, à l'emplacement ou aux préférences de l'utilisateur). De plus, si un service ou dispositif utilisé par l'application devient indisponible, l'application doit être adaptée dynamiquement afin de continuer à fournir ses services, par exemple en utilisant des services ou des dispositifs alternatifs.

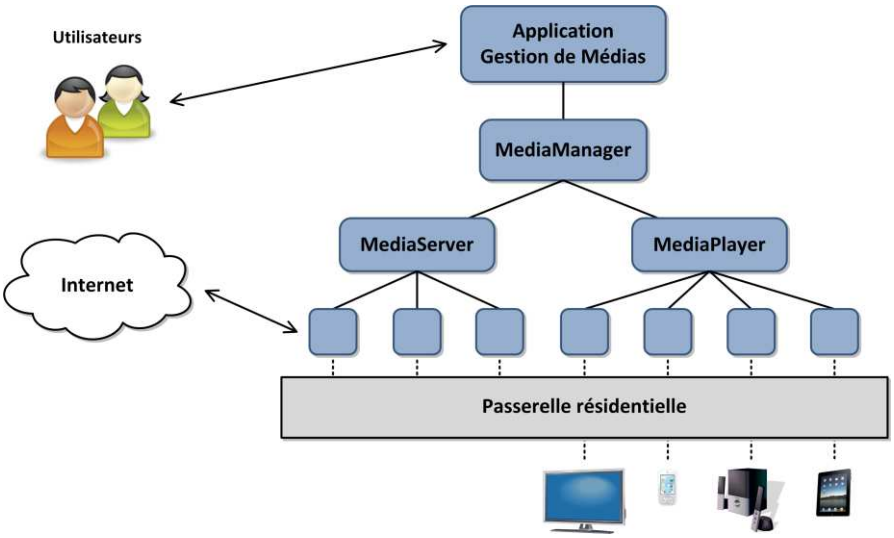


Figure 40. Exemple d'application de gestion multimédia pour des passerelles résidentielles

L'architecture concrète de cette application ne peut pas être complétement définie au moment de sa conception à cause de la disponibilité dynamique et imprévisible des services et des dispositifs. Notre approche propose de définir une telle application en intention (par un ensemble de propriétés, des contraintes et des préférences à satisfaire par les services participant à sa composition) et partiellement en extension (par un ensemble de services connus interconnectés) et de réaliser/contrôler sa composition pendant son exécution.

2 PRINCIPES ET MECANISMES DE BASE

Notre approche s'appuie sur le concept et le mécanisme de groupes d'équivalence. Un groupe d'équivalence est défini comme un ensemble d'éléments qui sont indiscernables pour un certain point de vue (une partie commune définie par des propriétés et des relations).

Les groupes d'équivalence sont fréquents. En UML[59], tout sous-type hérite de toutes les propriétés définies par son type : un type et ses sous-types forment un groupe d'équivalence. En programmation orientée objet, toutes les instances partagent les attributs communs définis par sa classe, et donnent une valeur à chaque attribut variable déclaré par celle-ci : toute classe et ses instances forment un groupe d'équivalence. Dans d'autres approches du génie logiciel, comme dans l'approche à composants ou dans l'approche à services, une définition (interface ou spécification respectivement) indique les propriétés fonctionnelles et non-fonctionnelles communes pour un ensemble d'implémentations. Chaque implémentation augmente ces informations par ses spécificités : toute définition et ses implémentations forment un groupe d'équivalence. Une autre utilisation de groupes d'équivalence se trouve dans le raffinement d'applications, comme dans le MDA[60], où la définition d'une application est raffinée de manière incrémentale de sa spécification à son implémentation. Le raffinement peut être géré en ajoutant à chaque étape des propriétés qui spécifient les choix effectués par les ingénieurs afin de restreindre l'espace des solutions : la définition d'une application et ses différents raffinements forment un groupe d'équivalence.

Dans les groupes d'équivalence mentionnés, nous pouvons constater les besoins de :

- **typage d'éléments** afin de leur imposer qu'ils soient conformes à la partie commune définie ; par exemple, une implémentation doit être conforme à sa spécification,
- **distinction des propriétés communes des propriétés spécifiques** à chaque élément afin de les réutiliser : la partie commune est un élément à part entière,
- **spécialisation/raffinement d'éléments** : un élément peut être substitué par un élément plus concret,
- **sélection d'éléments basée sur les propriétés communes** plutôt que par leur identité,
- **substitution d'éléments** par des éléments équivalents, et
- **manipulation de plusieurs éléments équivalents simultanément** : divers éléments équivalents peuvent cohabiter dans une même application.

Toutefois, les mécanismes existants ne couvrent pas, entre autres, l'ensemble des besoins énoncés ci-dessus. Pour cela, nous avons défini le concept de groupe d'équivalence associé à un mécanisme générique permettant de traiter ces besoins. Avant de détailler notre mécanisme de groupes, nous allons présenter d'abord des mécanismes existants de classification d'éléments et de propagation d'attributs qui permettent le typage et la spécialisation d'éléments.

2.1 MECANISMES DE CLASSIFICATION ET DE PROPAGATION

La définition de la partie commune d'un groupe d'éléments peut être considérée comme le type des éléments, car elle spécifie les propriétés communes de ceux-ci. Un élément est alors une instance, et peut à son tour être la définition de la partie commune d'un autre groupe d'éléments : il peut être instance et type à la fois. Cette double facette d'un objet est peu habituelle pour le monde de la programmation où il n'existe généralement que deux niveaux : type et instance ; par exemple, un objet Java ne peut pas être une classe Java. Toutefois, plusieurs langages, comme Ruby[61] ou Phython[62], permettent de définir des classes (des métaclasses) dont les instances sont des classes.

[59] *Unified Modeling Language*
[60] *Model Driven Architecture*
[61] **http://www.ruby-lang.org/**
[62] **http://www.python.org/**

Certaines approches de modélisation proposent des mécanismes de classification et de propagation d'attributs afin de résoudre le problème de multiples niveaux d'abstraction et de conformité.

La **matérialisation** [79], [80] est un patron qui définit une relation, appelée relation de matérialisation, entre une classe de catégories et une classe d'éléments plus concrets. Une relation de matérialisation est représentée par une ligne directe avec une étoile du côté de la classe plus concrète. La Figure 41 montre un exemple qui modélise des éléments d'une pièce de théâtre, et définit une relation de matérialisation entre les classes *Pièce_de_théâtre* et *Mise_en_scène* indiquant que toute mise en scène est une réalisation concrète (une matérialisation) d'une pièce de théâtre donnée de laquelle elle « hérite » d'un certain nombre de propriétés de plusieurs façons.

Une matérialisation possède deux facettes disjointes : instance et type. Chaque facette instance est une instance de la classe abstraite, tandis que la facette type associée est une sous-classe de la classe concrète. Dans l'exemple de la Figure 41, *Hamlet* est une facette instance de *Pièce_de_théâtre*, et *Mise_en_scène_Hamlet* est la facette type associée qui définit toutes les instances de *Mise_en_scène* de la pièce de théâtre *Hamlet*.

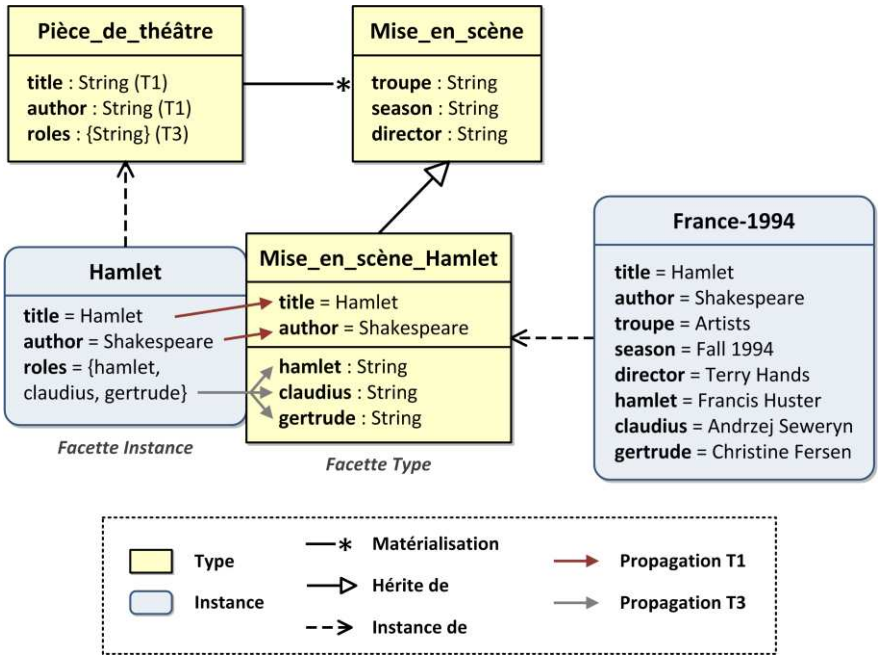


Figure 41. Matérialisation

Une facette type hérite des attributs de la classe concrète par un mécanisme d'héritage classique, et « hérite » aussi des attributs de la classe abstraite par des mécanismes de propagation d'attributs. Dans la Figure 41, nous illustrons les mécanismes de propagation T1 et T3. Le mécanisme T1 propage la valeur d'un attribut (mono- ou multivalué) d'une facette instance à la facette type associée comme un attribut de classe : les valeurs des attributs *title* et *author* (annotés T1) sont propagées de la facette instance *Hamlet* à la facette type *Mise_en_scène_Hamlet,* et seront donc héritées par toutes les instances de cette dernière. Le mécanisme T3 concerne des attributs multivalués (ensemble de *strings*) de la classe abstraite. Chaque élément dans l'ensemble des valeurs d'un attribut (annoté T3) d'une facette instance génère une nouvelle instance d'attribut en la facette type : l'attribut *roles* de *Pièce_de_théâtre* est propagé avec T3 à *Mise_en*_scène ; sa valeur {*hamlet*, *claudius*, *gertrude*} de la facette instance *Hamlet* génère trois instances d'attributs, *hamlet*, *claudius* et *gertrude*, dans la facette type associée *Mise_en_scène_Hamlet*. Ainsi, la mise en scène *France-1994* est une matérialisation de la pièce de théâtre *Hamlet* et possède donc les valeurs propagées de l'instance *Hamlet*.

Les matérialisations peuvent participer dans des compositions où la classe concrète d'une matérialisation est à son tour la classe abstraite d'une autre classe : la classe *Mise_en_scène* peut être la classe abstraite d'une classe *Réprésentation* par exemple. De plus, une classe abstraite peut avoir plusieurs relations de matérialisation vers des classes concrètes différentes.

L'approche **d'instanciation profonde** [81] cherche à répondre aux problèmes liés à la modélisation à multiples niveaux d'abstraction : classification ambiguë, réplication de concepts. Pour cela, cette approche considère que tout élément est potentiellement un type qui peut être instancié. L'instanciation des attributs des éléments est retardée en utilisant un nombre entier appelé *potency*. Le nombre *potency*, associé à chaque attribut d'un type d'élément, définit le nombre de niveaux d'instanciation nécessaires avant de donner une valeur à chaque attribut. Une instance hérite d'une définition d'attribut si *potency* est positif, et instancie l'attribut si *potency* est égal à zéro (voir la Figure 42). Ce mécanisme d'instanciation permet ainsi de définir les facettes d'instance et de type des éléments.

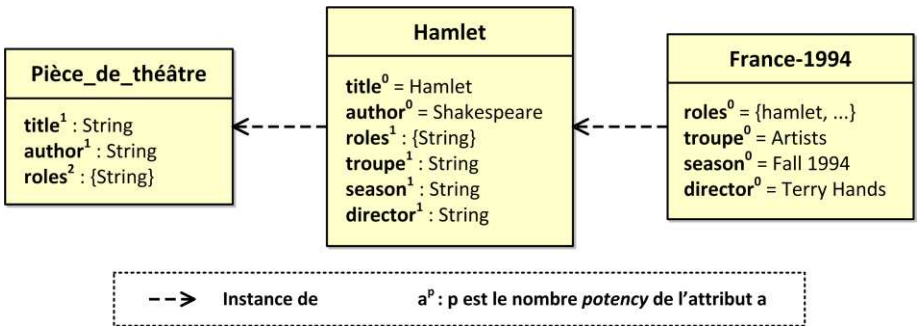


Figure 42. Instanciation profonde

Le concept de ***powertype*** [82], [83], défini par le langage de modélisation UML, permet de créer des sous-types dynamiquement. Un *powertype* est un type dont les instances sont des sous-types d'un autre type, nommé type partitionné. Un *powertype* et un type partitionné sont liés indirectement par des entités qui sont des instances du premier (facette instance) et des sous-types du second (facette type) : les instances d'un *powertype* sont en même temps des objets et des types (nommés *clabjets*). Dans l'exemple de la Figure 43, *Hamlet* est une instance de *Pièce_de_théâtre* (le *powertype*) est un sous-type de *Mise_en_scène* (le type partitionné), héritant ainsi des déclarations d'attributs de ce dernière.

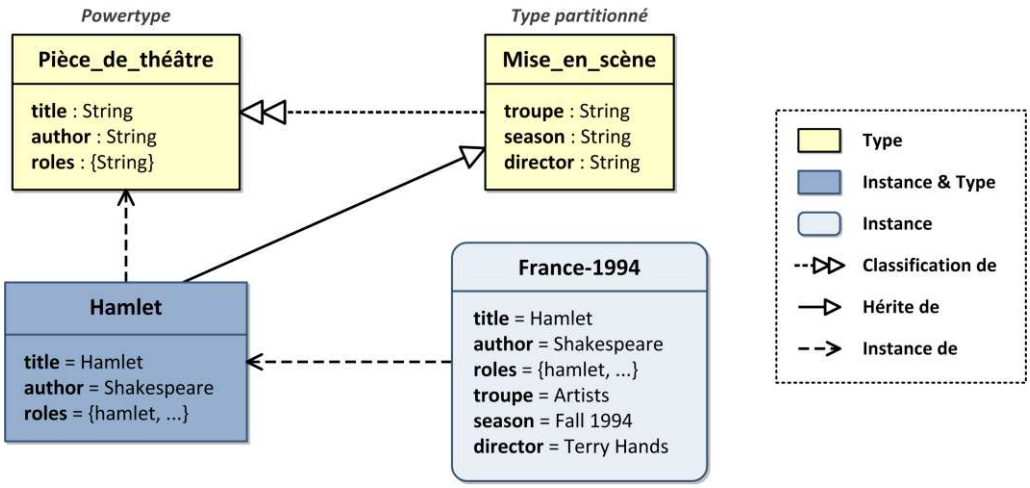


Figure 43. *Powertype*

Toutes ces approches redéfinissent la sémantique de l'instanciation. Elles présentent des avantages mais aussi des inconvénients par rapport à nos besoins (récapitulés dans le Tableau 8)[63].

	Avantages	***Limitations***
Matérialisation	*Divers types de propagation d'attributs/valeurs sont proposés. Le nombre de niveaux de classification est illimité.*	*Un objet résulte de l'assemblage des deux objets avec des identités distinctes. Chaque facette type est sous-type direct de la classe concrète d'une matérialisation. Ajouter des niveaux intermédiaires dans la hiérarchie de matérialisation requiert la modification des types de propagation utilisés.*
Instanciation profonde	*Le nombre de niveaux de classification est illimité.*	*Ajouter des niveaux intermédiaires dans la hiérarchie de classification requiert la modification des potency.*
Powertypes	*Le nombre de niveaux de classification est illimité.*	*Chaque instance d'un powertype est un sous-type direct d'un type partitionné. Considérer plus de deux niveaux d'abstraction du type partitionné requiert donc la modélisation de types et de relations additionnels.*

Tableau 8. Approches de classification et de propagation d'attributs

La limitation principale de ces approches, du moins pour l'usage que nous en faisons, se présente lorsqu'on souhaite utiliser des sous-types d'un type participant à la classification. Considérons par exemple un sous-type de *Mise_en_scène* nommé *Classique*. *Italy-1995* est une mise en scène (une instance) de *Hamlet*. Toutefois, elle est une mise en scène de type classique et est donc une instance du sous-type *Classique*. Cette situation n'est pas possible d'exprimer avec des *powertypes* (voir la Figure 44) : une instance de *Pièce_de_théâtre* ne peut être qu'un sous-type direct de *Mise_en_scène*. Or, il peut être intéressant de spécialiser aussi les types de mise en scène (classique, moderne, ...).

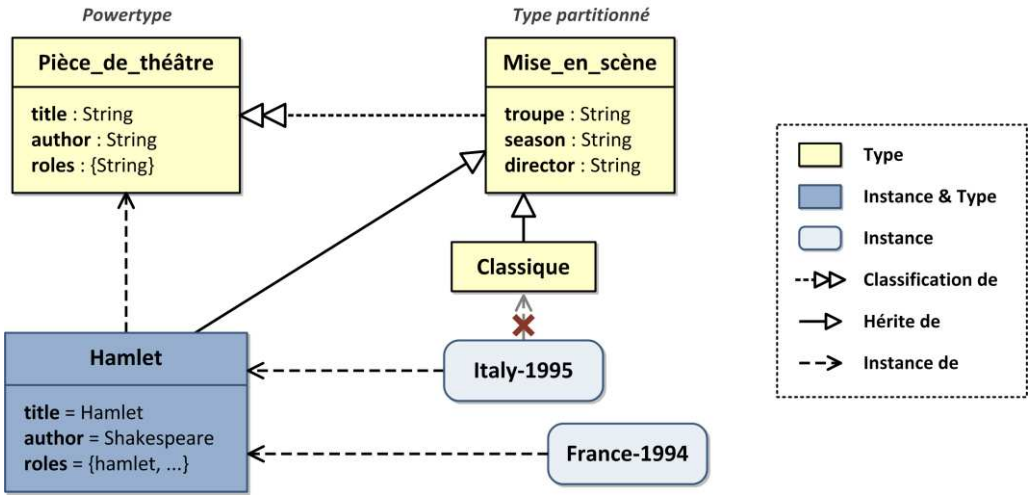


Figure 44. Sous-typage du type partitionné d'un *powertype*

[63] Dans [90], les approches de matérialisation, instanciation profonde, *powertypes* et objets multi-niveaux ont été comparées par rapport à différentes critères : compacité, extensibilité, flexibilité de requête, modélisation de relations à multiples niveaux.

Cela nous a amené à généraliser les mécanismes de matérialisation et de *powertypes*. Cette solution ajoute un mécanisme de propagation d'attributs propre aux relations de groupe (voir la Figure 45). Ce choix a été motivé par plusieurs considérations :

- la gestion automatique de la propagation des propriétés et des valeurs des éléments d'une classification avec un seul mécanisme,
- le sous-typage des types participant à une classification : les instances de tels sous-types appartient à la classification, et
- la possibilité d'avoir des éléments de types différents dans une même classification.

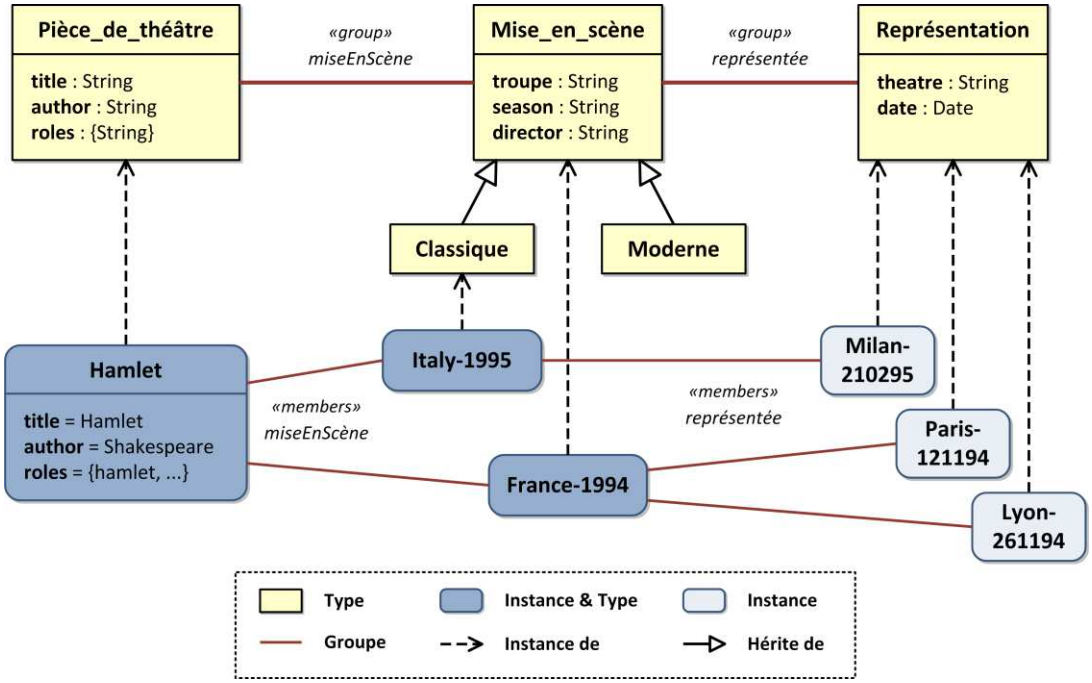


Figure 45. Mécanisme de groupes

Le mécanisme de groupes, détaillé ci-dessous, vise à répondre aux besoins énoncés précédemment : typage, distinction, concrétisation, sélection, substitution. Il permet ainsi la définition d'éléments à différents niveaux d'abstraction et leur catégorisation selon un point de vue commun. Il garantit la validité des catégories (par propagation/vérification d'attributs), et assure ainsi la matérialisation des éléments.

2.2 GROUPES D'EQUIVALENCE

Un **groupe d'équivalence** (ou simplement groupe) est défini comme un ensemble d'éléments qui sont indiscernables pour un certain point de vue. Un groupe est constitué d'un élément **représentant** et d'un ensemble d'éléments **membres** du groupe. Le représentant d'un groupe spécifie le point de vue du groupe : les propriétés (et relations) communes ainsi que les propriétés variables pour les membres du groupe. Les membres d'un groupe possèdent donc les mêmes propriétés (et relations) communes, et les mêmes propriétés variables avec des valeurs propres.

Un représentant est une **abstraction** des membres du groupe. Un membre est considéré comme un élément moins abstrait que son représentant : une **concrétisation**. Un représentant est considéré comme le type des membres du groupe. Un membre est alors considéré comme une instance du représentant, héritant de toutes ses propriétés. Un membre peut à son tour être le représentant d'un autre groupe, et être ainsi instance et type en même temps.

Le mécanisme de groupes permet ainsi la définition d'éléments à différents niveaux d'abstraction et leur catégorisation. Grâce au typage entre un représentant et les membres du groupe, la conformité et la cohérence des membres peuvent être vérifiées et assurées : l'ensemble des membres d'un groupe constituent un groupe d'équivalence valide. Cela permet d'assurer la concrétisation des éléments (i.e., le passage d'un niveau d'abstraction à un niveau plus concret). Le processus de concrétisation d'un élément, appelé résolution de groupe, est détaillé plus loin dans cette section.

2.2.1 TYPE DE GROUPE

Un groupe est spécifié à partir d'un **type de groupe**. Un type de groupe est défini entre des types : le type *RT* du représentant du groupe, un type de relation *GR*, dite relation de groupe, entre *RT* et un type *MT* de membres du groupe.

La Figure 46 illustre les concepts de type de groupe et de groupe sur l'exemple de modélisation des éléments d'une pièce de théâtre présenté précédemment. Deux types de groupes sont définis : le type de groupe *Pièce_de_théâtre* (représenté par le type *Pièce_de_théâtre* et dont les membres sont du type *Mise_en_scène*) et le type de groupe *Mise_en_scène* (représenté par le type *Mise_en_scène* et dont les membres sont du type *Représentation*). Ces types de groupes définissent ainsi qu'une pièce de théâtre a des mises en scène, et qu'une mise en scène donne lieu à plusieurs représentations. Ainsi, l'instance *Hamlet* de *Pièce_de_théâtre* est un représentant de groupe. L'instance *France-1994* de *Mise_en_scène*, étant un membre du groupe *Hamlet*, hérite des propriétés de *Hamlet*. Les instances *Paris-121194* et *Lyon-261194* sont des représentations de la mise en scène *France-1994* de *Hamlet.* Etant des membres du groupe *France-1994*, elles héritent des propriétés de *France-1994.*

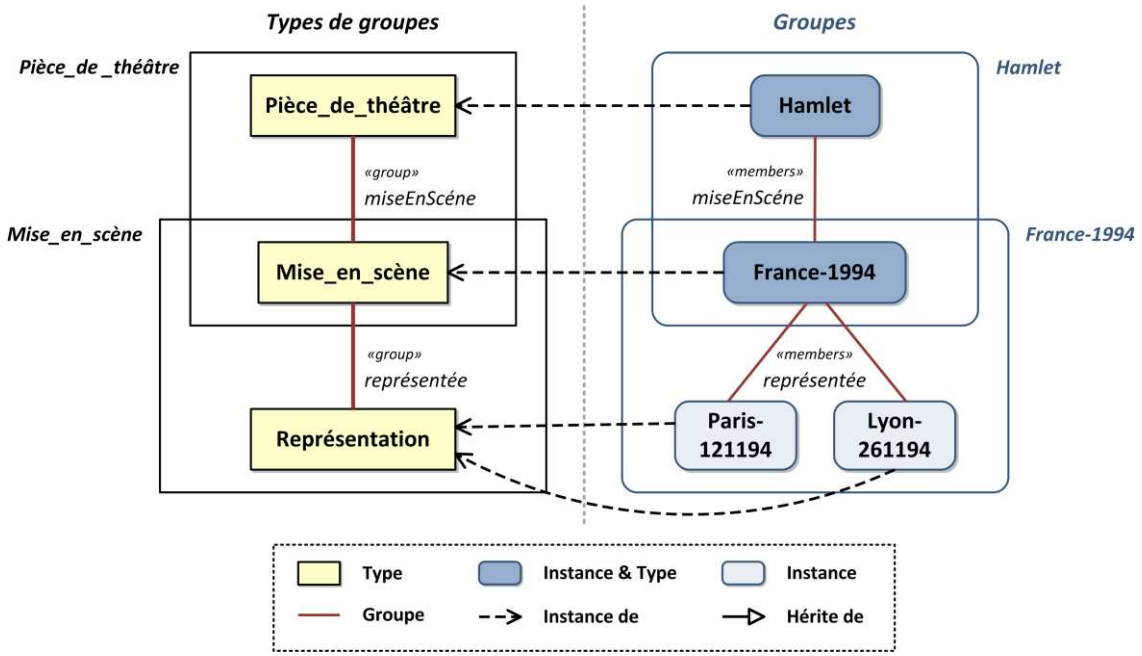


Figure 46. Types de groupes et groupes

Le concept de type de groupe, basé sur les notions de types d'éléments et de relations de groupe entre types d'éléments, est formellement défini comme suit :

TYPE DE GROUPE :
Soient « Types » un ensemble de types d'éléments et « Relations » un ensemble de relations entre types.

Soit RT un type d'élément, RT ∈ Types,
Le type RT définit un type de groupe, nommé RT, si est seulement s'il existe au moins une relation de groupe GR entre le type RT et un type d'élément MT, MT ∈ Types.
Le type de groupe RT implique que RT est le type de représentant du groupe, et que tout type MT ayant une relation de groupe avec le type RT, est un type de membres du type de groupe RT.

isGroupType(RT) $\Leftrightarrow \exists <RT, GR, MT> \in Relations,\ MT \in Types \wedge isGroupRelation(GR)$
$isGroupType(RT) \Rightarrow isGroupRepresentantType(RT) \ \wedge$
$\forall <RT, GR, MT> \in Relations,\ MT \in Types \wedge isGroupRelation(GR) \Rightarrow isGroupMemberType(RT,\ MT)$

Notations :
memberTypes(RT) = MT : l'ensemble de types de membres du type de groupe RT
$\forall mT \in MT, isGroupMemberType(RT,\ mT)$
isGroupRepresentantType(RT) : RT est le type de représentant du type de groupe RT
isGroupMemberType(RT, MT) : MT est un type de membres du type de groupe RT
isGroupRelation(GR) : GR es une relation de groupe (annotée « group »)

Un groupe (une instance d'un type de groupe) est défini comme un ensemble d'éléments (d'instances de types d'éléments) liés par une relation de groupe (notée « *members* ») à un élément représentant du groupe.

GROUPE :
Soient « Types » un ensemble de types d'éléments, « Elements » un ensemble d'éléments typés, et « Relations » un ensemble de liaisons entre éléments
Type(E) = T : E ∈ Elements est un élément de type T ∈ Types

*Un **groupe d'équivalence** est dénoté G = <R, M>, où :*
$R \in Elements$ *est le représentant du groupe G, tel que*
$Type(R) = RT \wedge isGroupRepresentantType(RT)$, *et*
$M = \{m_1, m_2, \ldots, m_n\}$ *est l'ensemble de membres du groupe G, tel que*
$\forall m \in M, (m \in Elements \wedge Type(m) \in memberTypes(RT) \wedge$
$\exists <R, GR, m> \in Relations,\ isGroupRelation(GR))$

2.2.2 RESOLUTION

La **résolution d'un groupe** consiste à sélectionner ou créer un ensemble de membres d'un groupe. Pour pouvoir créer des membres d'un groupe lors de sa résolution, le groupe doit être instanciable, i.e., il doit être associé à une fabrique (*factory method*) permettant la création de membres. Nous verrons que c'est le cas habituel du groupe dont le représentant est une implémentation et les membres sont des instances de cette implémentation.

La résolution d'un groupe peut être contrainte quant au nombre minimum obligatoire (*min*) et maximum permis (*max*) de membres à sélectionner/créer, par un intervalle $N=[min,max]$, tel que min, $max \in \mathbb{N}^*$ et $min \leq max$ (par défaut $min=max=1$). De plus, la résolution d'un groupe peut être contrainte quant aux propriétés que les membres sélectionnés/créés doivent satisfaire, par un ensemble C de contraintes. De telles contraintes portent sur les propriétés définies par le représentant du groupe, principalement sur les propriétés variables qui permettent de différencier l'ensemble des membres du groupe. La validité des contraintes de résolution d'un groupe peut ainsi être vérifiée et garantie : une contrainte est valide si elle fait référence à une propriété définie par le représentant du groupe.

De plus, la résolution d'un groupe peut être effectuée dans un des modes de résolution suivants : simple, complet, étendu ou complet étendu. Ces modes de résolution permettent de contrôler le niveau de résolution d'un groupe ainsi que la résolution de ses groupes associés.

Nous avons défini la fonction *resolve(G, N, C, complete, extended)=M*. Cette fonction résout le groupe *G*, en considérant l'intervalle *N*=[*min*,*max*] et l'ensemble *C* des contraintes de résolution, dans le mode de résolution spécifié (les variables *complete* et *extended* sont des variables booléennes, les deux variables mises à faux définissent le mode de résolution simple).

Lors de la **résolution simple** (nommée simplement résolution) d'un groupe *G* ayant *R* pour représentant, les relations « *members* » sont naviguées afin d'obtenir un ensemble *M* de *n* membres qui satisfont les contraintes de résolution *C*. Le nombre *n* de membres sélectionnés/crées doit appartenir à l'ensemble des nombres défini par l'intervalle *N*, $min \leq n \leq max$. La résolution d'un groupe *G* échoue s'il n'est pas possible de sélectionner et/ou de créer un nombre *min* de membres satisfaisant les contraintes de résolution *C*. Par exemple, la résolution du groupe *Hamlet* avec la contrainte « *(season=Fall 1994)* » retourne *France-1994.*

Lors de la **résolution complète** d'un groupe, la résolution est effectuée de manière récursive jusqu'à ce que les membres sélectionnés/créés ne soient plus eux-mêmes de représentants de groupe. Ainsi, un groupe *G* est complètement résolu si *G* est résolu, et si pour tout membre *MR* sélectionné/créé étant lui-même représentant de groupe, le groupe qu'il représente (un sous-groupe de *G*) est complètement résolu. Lors de la résolution complète d'un groupe *G*, l'ensemble de contraintes *C* sont propagés pour la résolution des sous-groupes de *G*. La résolution complète d'un groupe *G* échoue, si la résolution de *G* ou d'un sous-groupe de *G* échoue. Par exemple, la résolution complète du groupe *Hamlet* avec la contrainte « *((season=Fall 1994) AND (theatre=Marigny))* » retourne *Paris-121194*.

La **résolution étendue** d'un groupe *G* représenté par *R*, résout *G* ainsi que les groupes liés (« *links* ») à *R* et aux membres sélectionnés. Lors de la résolution étendue d'un groupe *G*, ni l'intervalle *N* ni l'ensemble de contraintes *C* ne sont propagés pour la résolution des groupes liés. Par contre, les propriétés (cardinalité et contraintes) définies sur les liaisons vers ces groupes sont considérées pour leur résolution. La résolution étendue d'un groupe *G* échoue si la résolution de *G* ou la résolution d'un des groupes liés échoue.

Enfin, la **résolution complète étendue** d'un groupe *G* représenté par *R*, résout complétement *G* ainsi que les groupes liés à *R* et à tous les membres sélectionnés. La résolution complète étendue d'un groupe *G* échoue, si la résolution complète de *G* ou d'un des groupes liés échoue.

La résolution étendue d'un groupe peut être effectuée en mode *backtrack* (retour sur trace) : lors de la résolution étendue d'un groupe *G*, si la résolution d'un groupe *G'* lié à *G* échoue, la sélection du membre sélectionné de *G'* est annulée et une autre résolution est effectuée (un autre membre de *G'* sera sélectionné). Effectuer la résolution en mode *backtrack* permet de trouver une solution si elle existe. Toutefois, elle peut être coûteuse en termes de temps d'exécution et d'utilisation de mémoire.

2.3 GROUPES DE SERVICES

Sur la base du mécanisme de groupes, nous définissons le concept de service et de service composite à trois niveaux d'abstraction : **spécification**, **implémentation** et **instance**. Une spécification est l'abstraction pour un ensemble d'implémentations de services. Une implémentation est à son tour l'abstraction pour un ensemble d'instances. En considérant ces trois types du concept de service, nous définissons deux types de groupes de services (voir la Figure 47) :

- **le type de groupe *Specification*** défini par le type *Specification* (type de représentant du groupe) et par sa relation de groupe *implements* avec le type *Implementation* (type de membre du groupe). Le type *CompositeSpecification*, étant sous-type du type *Specification*, est aussi un représentant du groupe (seulement pour des membres de type *CompositeImplementation*). Le type *CompositeImplementation*, étant sous-type du type *Implementation*, est aussi un type de membre du groupe *Specification*.

- **le type de groupe *Implementation*** défini par le type *Implementation* (type de représentant du groupe) et par sa relation de groupe *instantiates* avec le type *Instance* (type de membre du groupe). Le type *CompositeImplementation*, étant sous-type du type *Implementation*, est aussi un représentant du groupe (seulement pour des membres de type *CompositeInstance*). Le type *CompositeInstance*, étant sous-type du type *Instance*, est aussi un type de membre du groupe *Implementation*.

Nous spécifions ainsi que :

- une spécification est le type d'un ensemble d'implémentations,
- une implémentation est à la fois une instance de spécification et le type d'un ensemble d'instances : une implémentation est instance et type au même temps, et
- une instance est une instance d'implémentation.

Toutefois, les types composites sont restreints :

- une spécification composite peut être seulement le type d'un ensemble d'implémentations composites,
- une implémentation composite peut être seulement le type d'un ensemble d'instances composites, et
- une instance composite est seulement une instance d'implémentation composite.

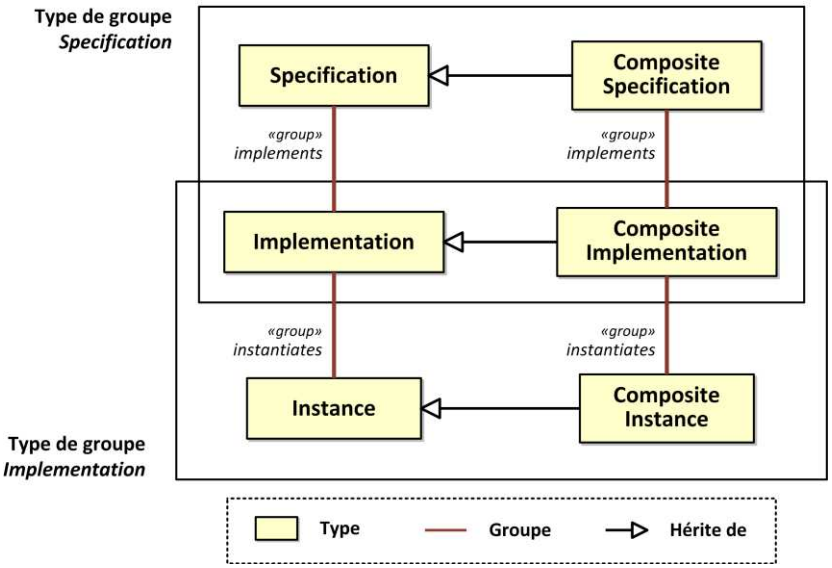


Figure 47. Types de groupes de services

Le mécanisme de groupes, tenant compte du sous-typage des types de membres de groupes, considère ainsi les instances d'une implémentation composite et les instances d'une instance composite comme des membres d'un groupe *Specification*.

Un groupe de services peut définir des relations de dépendance vers d'autres groupes de services. Une telle relation de dépendance (annotée « *links* ») est définie du représentant du groupe client vers le représentant du groupe fournisseur. Elle peut définir des contraintes (de cardinalité et de sélection) pour la résolution du groupe fournisseur.

Par exemple, considérons l'application de gestion multimédia présentée précédemment. Nous identifions trois services principaux : le gestionnaire, les bibliothèques, les lecteurs de médias ; nous définissons donc trois spécifications de services : *MediaManager*, *MediaServer* et *MediaPlayer*.

La Figure 48 illustre le groupe de services *MediaManager* à trois niveaux d'abstraction (spécification, implémentation et instance). Le représentant du groupe est la spécification *MediaManager* qui définit les propriétés suivantes pour les services qui l'implémentent (les membres du groupe) :

- **propriétés communes** : l'interface *MediaManagerService* décrivant les fonctionnalités fournies par le service ; et les relations *ms* et *mp* définissant des dépendances vers les spécifications (groupes) *MediaServer* et *MediaPlayer* respectivement.
- **propriétés variables** : les propriétés *provider* et *freeware* permettant à chaque service membre du groupe de spécifier, respectivement, le nom du fournisseur et le caractère gratuit ou payant du service.

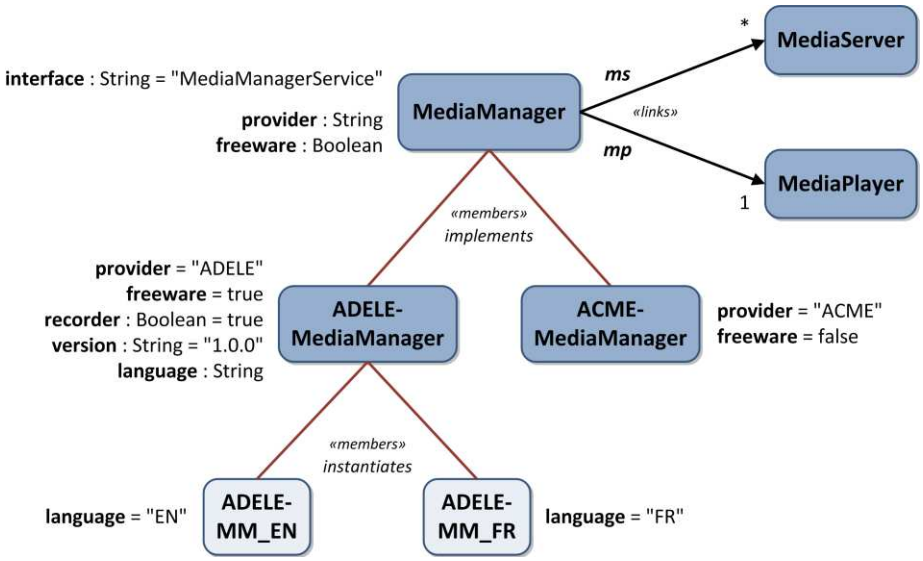
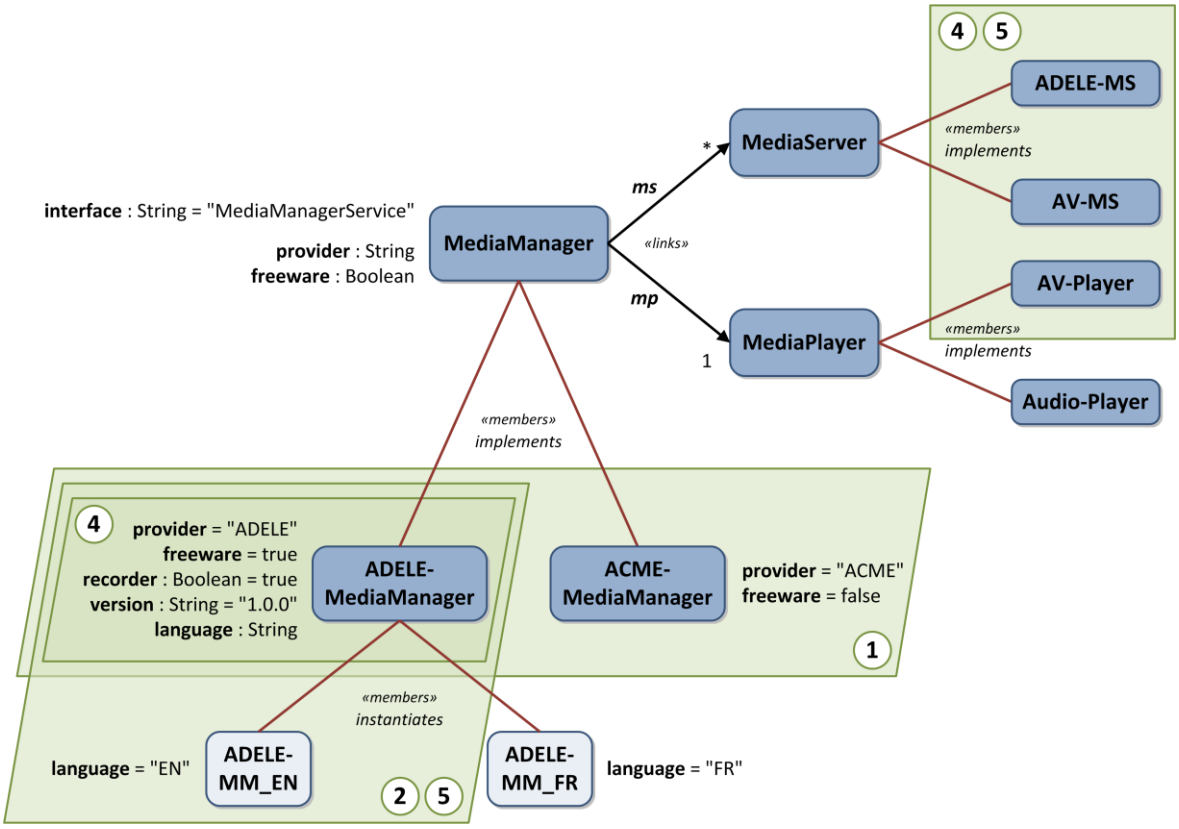


Figure 48. Le groupe de services *MediaManager*

Les implémentations *ADELE-MediaManager* et *ACME-MediaManager*, membres du groupe *MediaManager*, héritent des propriétés communes et variables définies par la spécification *MediaManager*. Les deux implémentations sont équivalentes si on ne considère que les propriétés communes, mais sont différenciables par les propriétés variables. L'implémentation *ADELE-MediaManager*, en tant que représentant de groupe, définit des propriétés communes (*recorder* et *version*) et variables (*language*) pour ses instances.

Les instances *ADELE-MM_EN* et *ADELE-MM_FR* de l'implémentation *ADELE-MediaManager* héritent de ses propriétés, et par transitivité de celles de la spécification *MediaManager*. Ces instances, équivalentes en considérant que leurs propriétés communes, sont membres du groupe *MediaManager*.

La Figure 49 illustre des exemples de résolution simple, complète, étendue et complète étendue du groupe *MediaManager*.



resolve(G, N, C, complete, extended) = M

1. resolve(MediaManager, [1,2], {}, false, false) = {ADELE-MediaManager, ACME-MediaManager}

2. resolve(MediaManager, [1,2], {(freeware=true)}, true, false) = {ADELE-MediaManager, ADELE-MM_EN}

3. resolve(MediaManager, [2,2], {(freeware=true)}, true, false) = {} // résolution échouée

4. resolve(MediaManager, [1,2], {(freeware=true)}, false, true)
= {ADELE-MediaManager, ADELE-MS, AV-MS, AV-Player}

5. resolve(MediaManager, [1,2], {(freeware=true)}, true, true)
= {ADELE-MediaManager, ADELE-MM_EN, ADELE-MS, AV-MS, AV-Player}

Figure 49. Exemples de résolution du groupe *MediaManager*

Ainsi, en utilisant le mécanisme de groupes, notre approche permet de définir des groupes de services à différents niveaux d'abstraction, et d'assurer leur concrétisation grâce au mécanisme de résolution. De plus, le mécanisme permet de réaliser un double typage pour les services : un typage technologique (spécification, implémentation, instance), ainsi qu'un typage métier (voir la Figure 50).

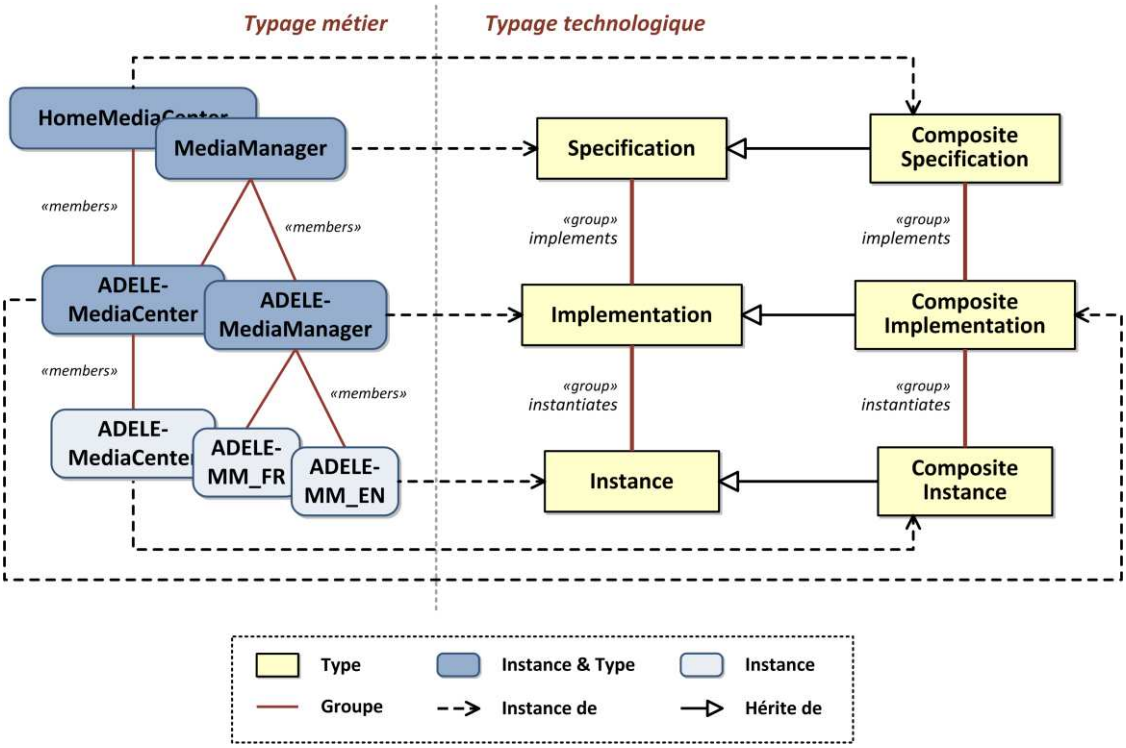


Figure 50. Types métier et types technologiques de groupes de services

En résumé, notre mécanisme de groupes est donc un mécanisme puissant qui :

- fournit un mécanisme solide d'abstraction et de typage,
- rend explicites les critères de classification des éléments,
- assure la conformité et la cohérence des éléments et garantit la validité des groupes,
- facilite la distinction, la sélection, la substitution des éléments,
- permet la sélection des éléments par des contraintes basées sur leurs propriétés plutôt que par leurs identités.

3 METAMODELE DE COMPOSANTS A SERVICES

Notre approche propose un métamodèle de composants pour la conception, le développement, la configuration et l'exécution de composants à services et d'applications à composants à services. La notion centrale de notre métamodèle est le concept de composant à services. Un **composant à services** est une entité, définie à différents niveaux d'abstraction, qui fournit des ressources (interfaces fonctionnelles, messages), requiert des ressources, possède des propriétés qui la caractérisent, ainsi que des contraintes et des préférences qui expriment les propriétés nécessaires et préférables pour son exécution. Une **application** est un composant à services, défini donc à différents niveaux d'abstraction, qui permet la composition d'autres composants à services.

Distinguer clairement les concepts de composant à services à différents niveaux d'abstraction nous semble essentiel pour pouvoir gérer les différentes phases du cycle de vie. Dans la suite, nous détaillons les différents éléments du métamodèle proposé. Nous décrivons d'abord les composants à services primitifs, et puis les composants et les mécanismes permettant leur composition.

3.1 COMPOSANTS A SERVICES PRIMITIFS

Dans notre approche, un composant à services primitif possède trois matérialisations, spécification, implémentation et instance, chacune correspondant à un niveau d'abstraction différent associé à une phase du cycle de vie des composants : conception, développement et exécution.

3.1.1 SPECIFICATION

Tout d'abord, une **spécification** décrit un composant à services par ses ressources (interfaces et messages) offertes et requises (voir la Figure 51).

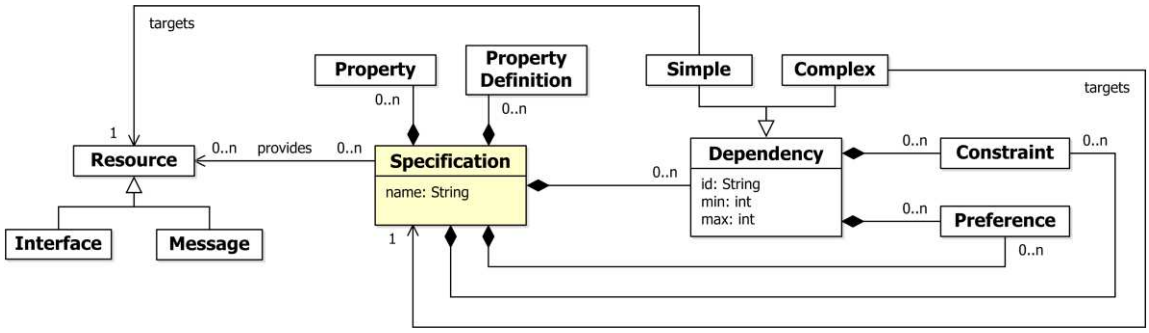


Figure 51. Spécification de composant à services

Contrairement aux approches à services, notre approche permet de définir des dépendances de ressources à partir de la spécification d'un composant à services. Ces dépendances de niveau spécification permettent la composition structurelle de spécifications (les spécifications sont composables) et augmentent la flexibilité de la composition (les implémentations des spécifications sont substituables). Ce niveau de dépendances n'annule pas les dépendances spécifiées au niveau des implémentations : une implémentation d'une spécification dépend des ressources spécifiées dans la spécification, et peut en addition dépendre d'autres ressources (dépendances propres à l'implémentation).

Les ressources requises peuvent être définies par des dépendances simples ou complexes (voir la Figure 51). Une dépendance simple est définie vers une seule ressource. Une dépendance complexe représente une dépendance vers un ensemble de ressources qui doivent être offertes par un même fournisseur. Les dépendances complexes permettent de gérer d'une façon plus contrôlée la résolution de dépendances, évitant les ambigüités de choix et les fonctionnements incohérents des clients.

Une dépendance, simple ou complexe, possède des attributs de cardinalité qui permettent d'exprimer l'optionalité et la multiplicité de la connexion ; et de filtrage qui permettent d'exprimer les propriétés nécessaires (contraintes) et préférables (préférences) que les fournisseurs des ressources requises doivent satisfaire. Les contraintes et les préférences de sélection sont exprimées dans notre langage de contraintes (présenté dans la section 4.1), qui vérifie la validité des expressions afin d'assurer la compatibilité entre clients et fournisseurs.

Ensuite, une spécification établit les propriétés (intrinsèques) et les définitions de propriétés qui caractérisent un composant à services. Les définitions de propriétés (aussi appelées propriétés configurables) sont en général affectées lors de la phase de développement du composant, toutefois elles peuvent être affectées à n'importe quel moment du cycle de vie du composant.

Enfin, une spécification définit les contraintes et les préférences (intrinsèques) qui caractérisent un composant à services. Celles-ci expriment respectivement les propriétés requises et préférables du composant, par exemple, une plate-forme particulière d'exécution ou une propriété non-fonctionnelle requise. De manière plus formelle, nous définissons le concept de spécification de la façon suivante :

*Une **spécification** est un 7-uplet S = <name, R, D, P, DP, CT, PF>, où :*

*name est le **nom symbolique** de la spécification S*

*$R = \{r_1, r_2, \ldots, r_n\}$ est l'ensemble des **ressources fournies** de S, tel que*
chaque ressource r_i est une interface ou un type de message

*$D = \{d_1, d_2, \ldots, d_n\}$ est l'ensemble des **dépendances** de S, tel que*
chaque dépendance d_i est un 5-uplet d = <id, target, cardinality, dCT, dPF>, où
id est l'identificateur de la dépendance d
target indique l'objet destination (une spécification ou une ressource) de d :
Type(target) = Specification | Resource
cardinality = <min, max> est la cardinalité de d, où
$min \in \mathbb{N}$ est le nombre minimum de connexions requises, et
$max \in \mathbb{N}^*$ est le nombre maximum de connexions autorisées, tel que
$min \leq max$, et max peut être défini comme un nombre indéterminé dénoté par « n »
dCT est l'ensemble des contraintes de d, et
dPF est l'ensemble des préférences de d

*$P = \{p_1, p_2, \ldots, p_n\}$ est l'ensemble des **propriétés** de S, tel que*
chaque propriété p_i est un 3-uplet p = <name, type, value>

*$DP = \{dp_1, dp_2, \ldots, dp_n\}$ est l'ensemble des **définitions de propriétés** de S, tel que*
chaque définition de propriété dp_i est un 3-uplet dp = <name, type, value>, où value est optionnel

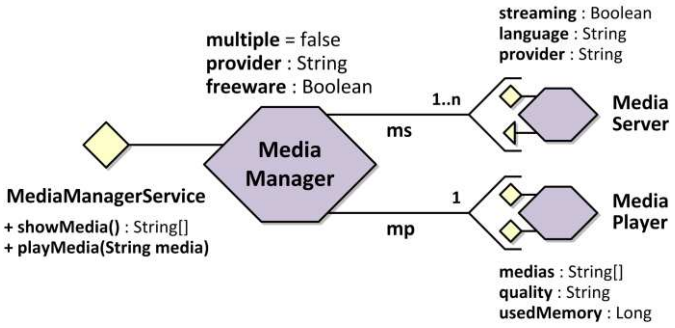
*CT est l'ensemble des **contraintes** de S, et*

*PF est l'ensemble des **préférences** de S*

Dans la pratique, une spécification est décrite dans un fichier de description XML en spécifiant ses ressources fournies et requises, ses propriétés et ses définitions de propriétés, ses contraintes et ses préférences. La Figure 52 présente un exemple de description XML (projeté sur Java) créée par notre environnement de développement (cf. Chapitre 7 section 1.2).

Pour illustrer le concept de spécification, nous reprenons les spécifications définies pour notre application de gestion multimédia : *MediaManager*, *MediaServer* et *MediaPlayer*. La spécification *MediaManager*, illustrée dans la Figure 52 :

- définit une ressource fournie à travers l'interface *MediaManagerService*,
- configure la propriété prédéfinie *multiple* (voir le Tableau 9),
- définit des définitions de propriétés avec les attributs *provider* et *freeware*, et
- spécifie des ressources requises par les dépendances *ms* et *mp* vers les spécifications *MediaServer* et *MediaPlayer* respectivement.




```
<specification name="MediaManager" interfaces="media.services.manager.MediaManagerService">
    <property name="multiple" value="false" />
    <definition name="provider" type="String" />
    <definition name="freeware" type="Boolean" />
    <dependency specification="MediaServer" id="ms" multiple="true" />
    <dependency specification="MediaPlayer" id="mp" multiple="false" />
</specification>
```

Figure 52. Spécification de composant à services – Exemple *MediaManager*

En résumant, les ressources fournies et requises, les propriétés et les définitions de propriétés, les contraintes et les préférences définies par une spécification sont les caractéristiques de tout composant à services qui l'implémente.

3.1.2 IMPLEMENTATION

Une spécification peut être implantée par plusieurs implémentations en utilisant différentes technologies à services (Services Web, UPnP, DPWS, OSGi, iPOJO) et/ou différents algorithmes. Une **implémentation** implémente une seule spécification (voir la Figure 53).

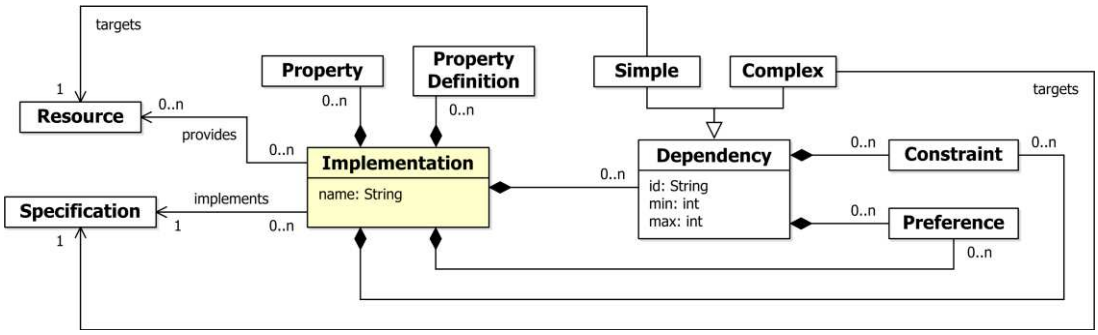


Figure 53. Implémentation de composant à services

La relation *implements* qui relie une implémentation et sa spécification est une relation de groupe : une spécification représente le type de ses implémentations. De ce fait, les caractéristiques définies par une spécification sont héritées par ses implémentations.

Une implémentation fournit et requiert ainsi, au moins, les ressources définies par sa spécification. Les dépendances définies par une spécification doivent être précisées par ses implémentations. Une dépendance peut être précisée en indiquant par exemple le nom de la variable ou de la méthode correspondante dans le code source, la cardinalité, les contraintes et les préférences additionnelles de sélection. Toutefois, les dépendances définies par une spécification contraignent les dépendances correspondantes de ses implémentations, par exemple, une dépendance obligatoire et

simple (*cardinality*=<1,1>) définie par une spécification ne peut pas être précisée comme optionnelle ou multiple par aucune de ses implémentations. Une implémentation possède également les propriétés, les contraintes et les préférences définies par sa spécification.

Selon la logique de groupes, les membres ont au moins les propriétés et les relations communes et variables définies par son représentant, mais peuvent en ajouter d'autres. Une implémentation peut ainsi ajouter des ressources fournies, des dépendances de ressources, des propriétés et des définitions de propriétés pour ses instances, ainsi que des contraintes et des préférences. Grâce au mécanisme de groupes, la validité d'une implémentation (i.e., sa conformité et sa cohérence vis-à-vis de son spécification) peut être assurée statiquement (lors de sa compilation/packaging).

Nous définissons formellement le concept d'implémentation comme suit :

Une ***implémentation*** *est un 8-uplet I = <name, S, R, D, P, DP, CT, PF>, où :*

name est le ***nom symbolique*** *de l'implémentation I*

S est la ***spécification*** *de I*

$R = SR \cup IR$ *est l'ensemble des* ***ressources fournies*** *de I, où*
SR est l'ensemble des ressources fournies de S, et
$IR = \{r_1, r_2, \ldots, r_n\}$ *est l'ensemble des ressources fournies définies par I*

$D = SID \cup ID$ *est l'ensemble des* ***dépendances*** *de I, où*
$SID = \{d_1, d_2, \ldots, d_n\}$ *est l'ensemble des dépendances de S précisées par I :*
Soit SD l'ensemble des dépendances de S
$\forall d' \in SD, \exists d = <id, target, cardinality, dCT, dPF> \in SID$:
id = id(d') est l'identificateur de la dépendance d
target = target(d') est la destination de d
cardinality = <min, max> est la cardinalité de d, où
$min \geq min(d') \wedge max \leq max(d') \wedge min \leq max$
dCT est l'ensemble des contraintes de d, $dCT(d') \subset dCT$
dPF est l'ensemble des préférences de d, $dPF(d') \subset dPF$
$ID = \{d_1, d_2, \ldots, d_n\}$ *est l'ensemble des dépendances définies par I, tel que*
chaque dépendance d_i *est un 5-uplet d = <id, target, cardinality, dCT, dPF>*

$P = SP \cup SIP \cup IP$ *est l'ensemble des* ***propriétés*** *de I, où*
SP est l'ensemble des propriétés de S
$SIP = \{p_1, p_2, \ldots, p_n\}$ *est l'ensemble des définitions de propriétés de S configurées par I :*
Soit SDP l'ensemble des définitions de propriétés de S
$\forall p \in SIP, \exists dp \in SDP : name(p) = name(dp) \wedge type(p) = type(dp)$
$IP = \{p_1, p_2, \ldots, p_n\}$ *est l'ensemble des propriétés définies par I, tel que*
chaque propriété p_i *est un 3-uplet p = <name, type, value>*

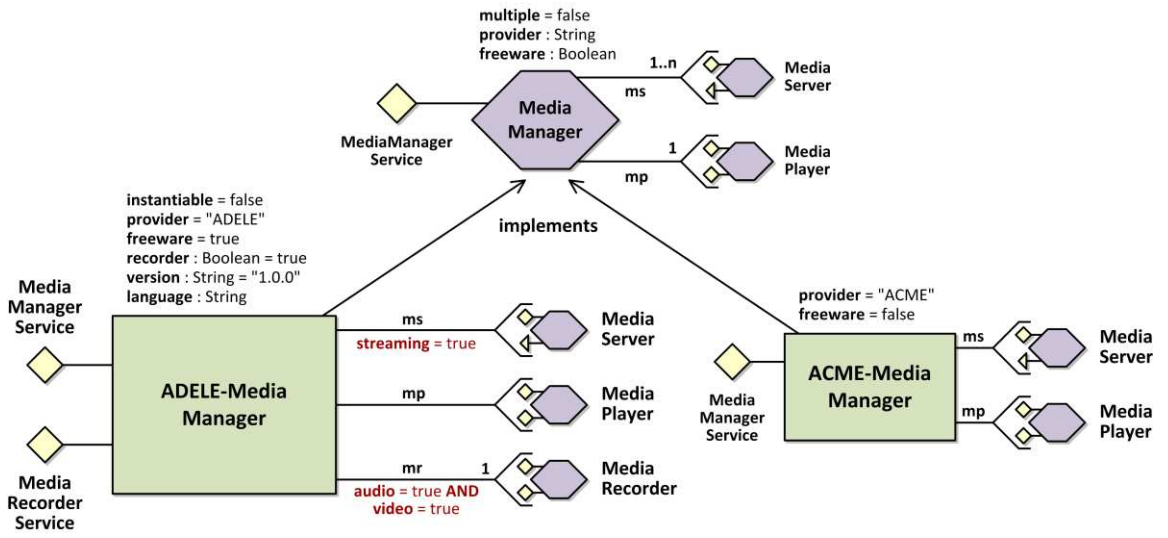
$DP = \{dp_1, dp_2, \ldots, dp_n\}$ *est l'ensemble* ***des définitions de propriétés*** *de I, où*
chaque propriété dp_i *est un 3-uplet dp = <name, type, value>, où value est optionnel*

$CT = SCT \cup ICT$ *est l'ensemble des* ***contraintes*** *de I, où*
SCT est l'ensemble des contraintes de S, et
ICT est l'ensemble des contraintes définies par I

$PF = SPF \cup IPF$ *est l'ensemble des* ***préférences*** *de I, où*
SPF est l'ensemble des préférences de S, et
IPF est l'ensemble des préférences définies par I

L'implémentation I est ***conforme*** *à la spécification* $S : I \models S$

La Figure 54 montre deux implémentations différentes de la spécification *MediaManager* présentée précédemment : *ACME-MediaManager* et *ADELE-MediaManager*. Elles possèdent donc les caractéristiques définies par la spécification *MediaManager* : l'interface fournie *MediaManagerService*, les spécifications requises *MediaServer* et *MediaPlayer*, et les propriétés *provider* et *freeware*.




```
<implementation name="ADELE-MediaManager" specification="MediaManager"
interfaces="media.services.MediaRecorderService" classname="adele.media.manager.ADELEMediaManager">
    <property name="instantiable" value="false" />
    <property name="provider" value="ADELE" />
    <property name="freeware" value="true" />
    <property name="recorder" type="Boolean" value="true" />
    <property name="version" type="String" value="1.0.0" />
    <definition name="language" type="String" />

    <dependency specification="MediaServer" id="ms">
        <interface field="mediaServers" name="mediaServerServices" />
        <constraints>
            <implementation filter="(streaming=true)" />
        </constraints>
    </dependency>
    <dependency specification="MediaPlayer" id="mp">
        <interface field="audioPlayer" name="audioPlayerService" />
        <interface field="videoPlayer" name="videoRecorderService" />
    </dependency>
    <dependency specification="MediaRecorder" id="mr" multiple="false">
        <interface field="audioRecorder" name="audioRecorderService" />
        <interface field="videoRecorder" name="videoRecorderService" />
        <constraints>
            <implementation filter="(&(audio=true)(video=true))" />
        </constraints>
    </dependency>
</implementation>
```

Figure 54. Implémentation de composant à services – Exemple *ADELE-MediaManager*

L'implémentation *ADELE-MediaManager* est un *MediaManager* « standard » mais en plus, il fournit des fonctionnalités pour l'enregistrement programmé de médias à travers l'interface *MediaRecorderService*, requiert un service d'enregistrement via la dépendance *mr* vers la spécification *MediaRecorder*, définit les propriétés *recorder* et *version*, et la définition de la propriété *language*. La Figure 54 présente aussi la description XML de cette implémentation.

En récapitulant, les ressources fournies et requises, les propriétés et les définitions de propriétés, les contraintes et les préférences d'une implémentation sont les caractéristiques de tout composant à services qui l'instancie.

3.1.3 INSTANCE

Une implémentation peut avoir diverses instances. Une **instance** appartient à une seule implémentation (voir la Figure 55).

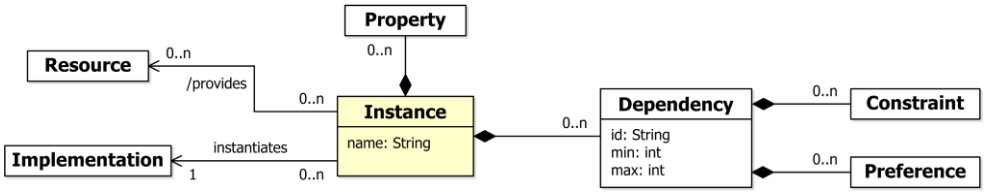


Figure 55. Instance de composant à services

De même que la relation entre une implémentation et sa spécification, la relation *instantiates* entre une instance et son implémentation est une relation de groupe : une implémentation représente le type de ses instances. Ainsi, les caractéristiques d'une implémentation, et par transitivité celles de sa spécification, sont héritées par ses instances (ressources fournisses et requises, propriétés, contraintes et préférences). Une instance ne peut pas définir de capacités, de dépendances ou de propriétés propres. Toutefois, elle peut ajouter des contraintes et des préférences de sélection sur les dépendances héritées de son implémentation.

Grâce au mécanisme de groupes, la validité d'une instance (i.e., la conformité et la cohérence de sa déclaration vis-à-vis de son implémentation) peut être assurée statiquement (lors de sa compilation, son packaging ou son déploiement). Nous définissons formellement le concept d'instance de la façon suivante :

*Une **instance** est un 7-uplet Inst = <name, I, R, D, P, CT, PF>, où :*

*name est le **nom symbolique** de l'instance Inst*

*I est l'**implémentation** de Inst*

*R est l'ensemble des **ressources fournies** de Inst, égal à l'ensemble des ressources fournies de I*

$D = InstSD \cup InstD$ *est l'ensemble des **dépendances** de Inst, tels que*
$InstSD = \{d_1, d_2, \ldots, d_n\}$ *est l'ensemble des dépendances de I précisées par Inst :*
Soit ID l'ensemble des dépendances de I,
$\forall d = \langle id, target, cardinality, dCT, dPF\rangle \in InstSD, \exists d' \in ID$, *tel que*
$id = id(d') \wedge target = target(d') \wedge cardinality = cardinality(d')$
$dCT(d') \subset dCT$
$dPF(d') \subset dPF$
$InstD = ID \setminus InstSD$ *est l'ensemble des dépendances de I non précisées par Inst*

$P = IP \cup InstP$ *est l'ensemble des **propriétés** de Inst, où*
IP est l'ensemble des propriétés de I
InstP est l'ensemble des définitions de propriétés de I ou de la spécification S de I configurées par Inst :
Soient IDP l'ensemble des définitions de propriétés de I, et
SDP l'ensemble des définitions de propriétés de S(I) et
$\forall p \in InstP, \exists dp : dp \in IDP \vee dp \in SDP \wedge name(p) = name(dp) \wedge \neg\exists p' \in IP : name(p) = name(p')$

*CT est l'ensemble des **contraintes** de Inst, égal à l'ensemble des contraintes de I*

*PF est l'ensemble des **préférences** de Inst, égal à l'ensemble des préférences de I*

L'instance Inst est conforme à l'implémentation I : $Inst \models I$, *et donc*
elle est aussi conforme à la spécification S de I : $Inst \models I \Rightarrow Inst \models S(I)$

3.1.4 PROPRIETES PREDEFINIES

Les trois types de composants à services – spécification, implémentation et instance – disposent d'un ensemble de propriétés prédéfinies et configurables qui caractérisent leur nature. En général, ces propriétés sont affectées lors de la création/définition des composants. Le Tableau 9 présente les propriétés prédéfinies propres à chaque type de composant ainsi que leur sémantique.

		Propriété	***Sémantique***
Spécification	***Implémentation***	***instantiable*** [*true* | *false*]	*Cette propriété indique si le composant (étant un représentant de groupe) est instanciable ou non. La valeur par défaut est false pour les spécifications, et true pour les implémentations.*
		multiple [*true* | *false*]	*Cette propriété indique si le composant (étant un représentant de groupe) peut être résolu par plusieurs membres ou non (i.e., si plus d'une résolution est autorisée). La valeur par défaut est true.*
		singleton [*true* | *false*]	*Cette propriété indique si le composant (étant un représentant de groupe) peut avoir plus qu'un seul membre ou non. La valeur par défaut est false.*
	Instance	***shared*** [*true* | *false*]	*Cette propriété indique si le composant peut être utilisé par plusieurs consommateurs simultanément ou non. La valeur par défaut est true.* *Note : les spécifications et les implémentations peuvent être utilisées par plusieurs consommateurs simultanément : elles sont toujours partageables.*

Tableau 9. Propriétés prédéfinies des composants à services primitifs

Ces propriétés permettent de contrôler leur résolution (pour les spécifications et les implémentations) et leur composition avant et/ou durant leur exécution.

3.2 COMPOSANTS A SERVICES COMPOSITES

Les composants primitifs peuvent être assemblés afin de composer une application ou un sous-système logiciel. Comme introduit dans le chapitre précédent, notre approche de construction d'applications repose sur les concepts de spécification composite, implémentation composite et instance composite, chacun correspondant à un niveau d'abstraction différent.

3.2.1 SPECIFICATION COMPOSITE

Une **spécification composite** spécifie une application ou un sous-système (un composite) par l'ensemble des propriétés qui le caractérisent : propriétés architecturales (ressources fournies et requises, spécifications contenues et connecteurs entre elles), contraintes et préférences. D'une manière similaire à une architecture de référence dans une approche de lignes de produits, une spécification composite décrit les propriétés structurelles et sémantiques, y compris des points de variation, pour un ensemble (ou famille) de composites à développer.

Une spécification composite est elle-même une spécification (voir le métamodèle dans la Figure 56). De ce fait, elle définit les ressources fournies et requises, et les propriétés, contraintes et préférences intrinsèques du composite.

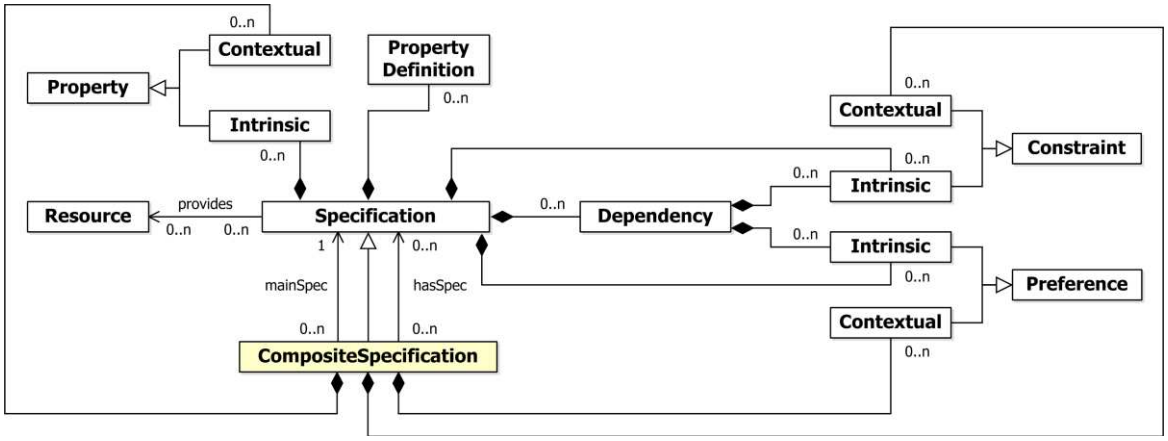


Figure 56. Spécification composite

En tant que composite, une spécification composite définit un ensemble de spécifications contenues (primitives ou composites) parmi lesquelles on retrouve (au moins) une spécification identifiée comme la spécification principale (*mainSpec*). Une spécification principale possède comme ressources fournies au moins les ressources fournies de la spécification composite. Les dépendances définies par la spécification principale d'une définition de composite seront résolues à l'intérieur du composite (lors de sa composition) sauf si elles sont définies comme dépendances de la spécification composite, auquel cas elles seront résolues à l'extérieur du composite.

Une spécification composite peut définir des ***propriétés contextuelles*** pour un composite. La valeur d'une telle propriété est déterminée en fonction des valeurs de certaines propriétés de divers composants appartenant au composite. Par exemple, l'état d'un composite peut être déterminé en considérant les états de ses composants. Le processus d'affectation de propriétés contextuelles se base sur des politiques définies par les architectes des composites : lors de la satisfaction d'une condition donnée, la propriété est affectée avec la valeur spécifiée.

De plus, une spécification composite peut définir des ***contraintes et des préférences contextuelles*** indiquant respectivement les propriétés requises et préférables que les composants (implémentations et instances) participant au composite doivent satisfaire. Les contraintes et préférences contextuelles permettent de guider/contrôler la composition du composite.

Une spécification composite est spécifiée comme une spécification primitive à laquelle on ajoute les caractéristiques propres à la conception d'un composite. Sa définition est ainsi la suivante :

*Une **spécification composite** est un 6-uplet CS = <S, **mS**, **Ss**, **CP**, **CCT**, **CPF**>, où :*

S = <name, R, D, P, DP, CT, PF> est la spécification primitive de CS

*mS est la **spécification principale** de CS, tel que*
R(S) ⊂ R(mS) les ressources fournies de mS sont au moins les ressources fournies de S

*Ss est l'ensemble des **spécifications contenues** dans CS*

*CP est l'ensemble des **propriétés contextuelles** de CS*

*CCT est l'ensemble des **contraintes contextuelles** de CS*

*CPF est l'ensemble des **préférences contextuelles** de CS*

Nous avons défini des propriétés contextuelles qui sont associées aux composants composites et qui permettent d'établir le mode de résolution de leurs groupes (fermé, coopératif, opportuniste), ainsi que la visibilité (et le partage) de leurs composants contenus.

Une spécification composite peut spécifier, par exemple, les composants pour lesquels la résolution peut considérer les composants appartenant à d'autres composants composites ; et les composants qui ne sont pas visibles ni partagés à d'autres composites. L'ensemble des propriétés contextuelles prédéfinies est présenté, plus loin dans cette section, dans le Tableau 10.

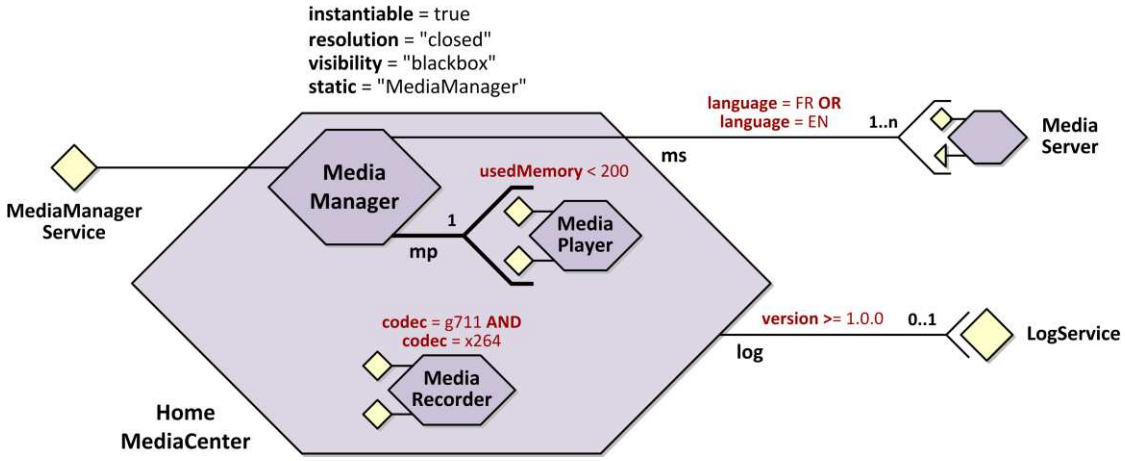
Pour construire l'application de gestion multimédia présentée dans la section 1 de ce chapitre, nous définissons à l'aide de notre langage de composition (cf. section 4.1.1) une spécification composite appelée *HomeMediaCenter*, illustrée dans la Figure 57.

En tant que spécification, *HomeMediaCenter* :

- établit l'interface *MediaManagerService* comme sa ressource fournie,
- configure la propriété prédéfinie *instantiable*, et
- spécifie deux dépendances avec des contraintes de cardinalité et de sélection : l'une (obligatoire et multiple) vers la spécification *MediaServer* et l'autre (optionnelle et simple) vers l'interface *LogService*.

En tant que spécification composite, *HomeMediaCenter* :

- établit la spécification *MediaManager* comme son spécification principale,
- spécifie deux spécifications contenues, *MediaPlayer* et *MediaRecorder*, avec des contraintes contextuelles de sélection, et
- configure les propriétés contextuelles prédéfinies pour la résolution (*resolution* et *static*) et la visibilité (*visibility*) de ses composants.




```
CompositeSpecification HomeMediaCenter {
    property resolution = closed;
    property visibility = blackbox;
    property instantiable = true;
    attribute provider type String;
    provides media.services.manager.MediaManagerService from static MediaManager;
    contains MediaPlayer instance (usedMemory < "200");
    contains MediaRecorder (codec = "g711" AND codec = "x264");
    requires multiple MediaServer (language = "FR" OR language = "EN") id ms;
    requires optional org.osgi.log.LogService (version >= "1.0.0") id log;
}
```

Figure 57. Spécification composite – Exemple *HomeMediaCenter*

Diverses implémentations composites peuvent être créées, avant ou pendant la phase d'exécution, à partir d'une spécification composite.

3.2.2 IMPLEMENTATION COMPOSITE

Une **implémentation composite**, appelée simplement composite, représente une application ou un sous-système au niveau développement/composition (voir le métamodèle dans la Figure 58).

Etant une implémentation, un composite implémente une seule spécification (primitive ou composite) et hérite de toutes ses caractéristiques. Il peut affecter les valeurs des propriétés définies par sa spécification, et peut définir des ressources fournies et requises propres, des propriétés et des définitions de propriétés, des contraintes et des préférences de sélection sur les dépendances héritées de sa spécification, des contraintes et des préférences propres.

Etant un composite, il contient des implémentations (primitives ou composites) et des connecteurs (*connector*) entre elles. Parmi ces implémentations, une implémentation au moins est identifiée comme l'implémentation principale du composite (*mainImpl*). L'implémentation principale fournit au moins les ressources fournies du composite (i.e., les ressources fournies de la spécification du composite et les ressources fournies propres au composite). Si le composite implémente une spécification composite (liés par la relation *conforms*), son implémentation principale doit implémenter la spécification principale (*mainSpec*) de la spécification composite. Un composite peut définir/ajouter des propriétés, des contraintes et des préférences contextuelles.

Les dépendances d'un composite, celles héritées de sa spécification et celles propres au composite, sont liées à des dépendances de ses implémentations contenues et seront donc résolues à l'extérieur du composite. La résolution d'une telle dépendance résulte ainsi dans une connexion entre une implémentation contenue dans le composite et une implémentation externe, et dans une connexion entre le composite et l'implémentation externe.

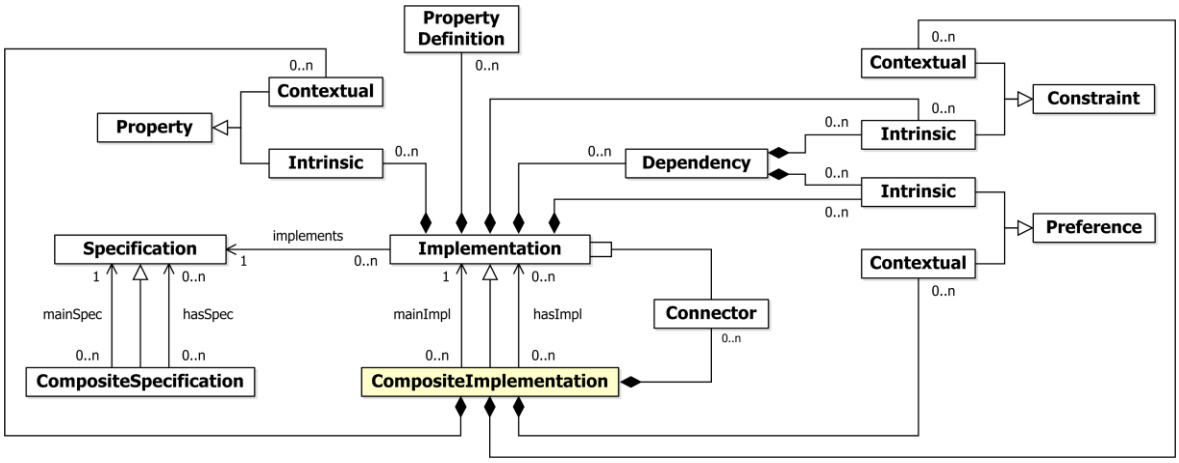


Figure 58. Implémentation composite

Un composite est spécifié comme une implémentation primitive à laquelle on ajoute les caractéristiques propres au développement d'un composite. Nous définissons formellement le concept de composite comme suit :

*Une **implémentation composite** est un 7-uplet CI = <I, **mI**, **Is**, **Cx**, **CP**, **CCT**, **CPF**>, où :*

I = <name, S, R, D, P, DP, CT, PF> est l'implémentation primitive de CI

*mI est l'**implémentation principale** de CI, tel que*
$R(I) \subset R(mI)$ les ressources fournies de mI sont au moins les ressources fournies de I

*Is est l'ensemble des **implémentations contenues** dans CI*

*$Cx = \{cx_1, cx_2, \ldots, cx_n\}$ est un ensemble de **connexions** entre implémentations contenues dans CI, tel que*
chaque connexion cx_i est un 2-uplet $cx = <I_1, I_2>$, où
$I_1 \in Is$ est l'origine de la dépendance, et $I_2 \in Is$ est la destination de la dépendance, tel que
$\exists d \in D(I_1)$, $target(d) = S(I_2) \mid target(d) \in R(I_2)$ il existe une dépendance de I_1 dont la destination (target) est la spécification de I_2, ou une ressource fournie par I_2

*$CP = SCP \cup ICP$ est l'ensemble des **propriétés contextuelles** de CI, où*
SCP est l'ensemble des propriétés contextuelles de la spécification S de I (si S est primitive, $SCP = \emptyset$), et
ICP est l'ensemble des propriétés contextuelles définies par CI

*$CCT = SCCT \cup ICCT$ est l'ensemble des **contraintes contextuelles** de CI, où*
SCCT est l'ensemble des contraintes contextuelles de la spécification S de I (si S est primitive, $SCCT = \emptyset$), et
ICCT est l'ensemble des contraintes contextuelles définies par CI

*$CPF = SCPF \cup ICPF$ est l'ensemble des **préférences contextuelles** de CI, où*
SCPF est l'ensemble des préférences contextuelles de la spécification S de I (si S est primitive, $SCPF = \emptyset$), et
ICPF est l'ensemble des préférences contextuelles définies par CI

*L'implémentation CI est **conforme** à la spécification S de I : $CI \models S$*

Dans notre approche, un composite peut être créé à partir d'une spécification primitive ou composite avant ou pendant la phase d'exécution. Sa validité est assurée par le mécanisme de groupes.

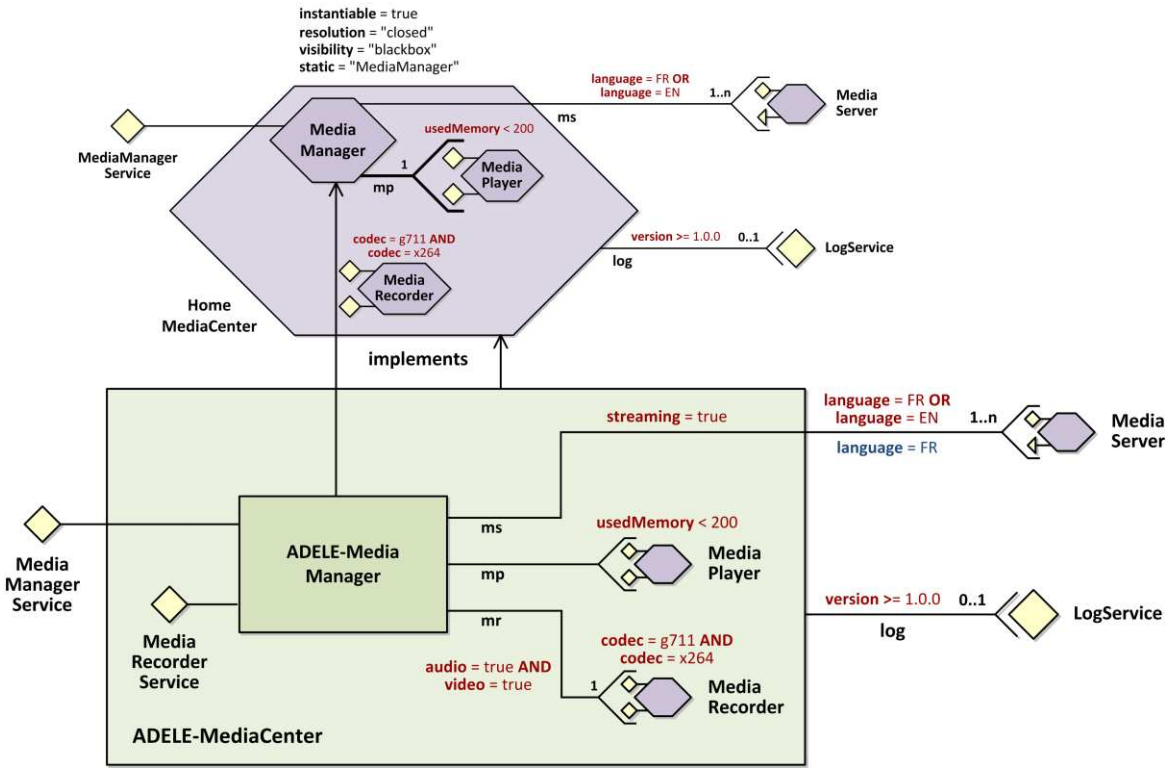
Pour illustrer le concept de composite, nous définissons l'implémentation composite *ADELE-MediaCenter* qui implémente la spécification composite *HomeMediaCenter* présentée auparavant. Cette implémentation, illustrée dans la Figure 59, hérite donc de toutes les caractéristiques définies par *HomeMediaCenter*. Sa composition est guidée (au moins) par l'ensemble des propriétés, contraintes et préférences contextuelles définies par *HomeMediaCenter*.

L'implémentation primitive *ADELE-MediaManager* (aussi présentée précédemment) est l'implémentation principale de *ADELE-MediaCenter*. De ce fait, *ADELE-MediaManager* implémente la spécification *MediaManager* définie comme la spécification principale de *HomeMediaCenter*.

Lors de la composition du composite, les dépendances de ses implémentations contenues sont résolues à l'intérieur du composite (résultant dans une nouvelle implémentation contenue et/ou connexion) sauf si elles sont définies comme des dépendances du composite.

Par exemple, considérons la dépendance vers la spécification *MediaServer* de l'implémentation *ADELE-MediaManager* contenue dans le composite *ADELE-MediaCenter* (voir la Figure 59). Cette dépendance étant aussi une dépendance du composite, sa résolution résultera donc dans une connexion entre *ADELE-MediaManager* et une implémentation externe sélectionnée, ainsi qu'une connexion entre *ADELE-MediaCenter* et l'implémentation externe.

Un composite peut être créé et composé partiellement ou complètement avant ou pendant la phase d'exécution. Dans la phase de développement, un composite représente (partiellement ou complètement) la structure statique de l'application en termes d'implémentations. Dans la phase d'exécution, un composite représente l'état d'exécution actuel d'un composite en termes d'implémentations. Notre mécanisme automatique de composition, exécuté durant la phase de développement ou durant la phase d'exécution, assure la conformité et la cohérence d'une composition vis-à-vis de sa spécification. Tout composite créé avant l'exécution est vérifié statiquement, lors de sa phase de compilation/packaging, afin d'assurer sa validité. Diverses instances peuvent être créées, avant ou pendant la phase d'exécution, à partir d'un composite.




```
<composite name="ADELE-MediaCenter" mainImplem="ADELE-MediaManager" specification="MediaManager">
    <property name="provider" value="ADELE" />
    <dependency specification="MediaServer" id="ms" multiple="true">
        <constraints>
            <implementation filter="(|(language=FR)(language=EN))" />
        </constraints>
        <preferences>
            <implementation filter="(language=FR)" />
        </preferences>
    </dependency>
    <dependency interface="org.osgi.service.log.LogService" id="log">
        <constraints>
            <implementation filter="(version&gt;=1.0.0)" />
        </constraints>
    </dependency>

    <contentMngt>
        <dependency specification="MediaPlayer">
            <constraints>
                <instance filter="(usedMemory&lt;200)" />
            </constraints>
        </dependency>
        <dependency specification="MediaRecorder">
            <constraints>
                <implementation filter="(&(codec=g711)(codec=x264))" />
            </constraints>
        </dependency>
        <borrow implementation="false" instance="false" />
        <local implementation="true" instance="true" />
    </contentMngt>
</composite>
```

Figure 59. Implémentation composite – Exemple *ADELE-MediaCenter*

3.2.3 INSTANCE COMPOSITE

Une **instance composite** représente une application ou un sous-système au niveau configuration (dans la phase de développement) et au niveau exécution (dans la phase d'exécution). Une instance composite est elle-même une instance qui instancie un composite (une implémentation composite), héritant ainsi des toutes ses caractéristiques et (par transitivité) de celles de sa spécification. En tant que composite, une instance composite contient un ensemble d'instances (primitives ou composites) et de liaisons entre elles. Parmi ces instances, une instance est identifiée comme l'instance principale de l'instance composite (*mainInst*), qui implémente l'implémentation principale du composite instancié.

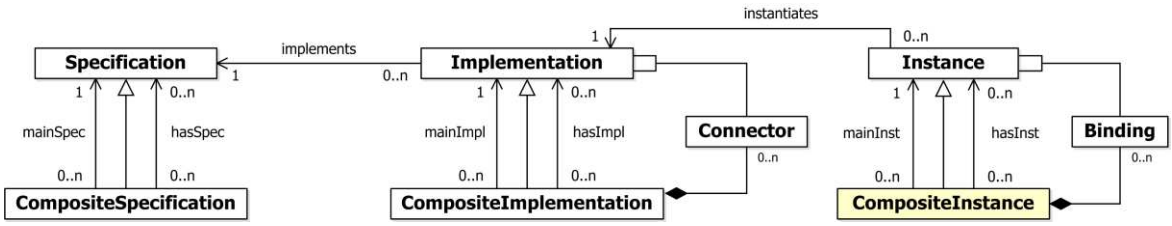


Figure 60. Instance composite

Au développement, une instance de composite représente une configuration particulière d'un composite : une déclaration d'instance composite. Une telle déclaration peut spécifier les valeurs des propriétés définies par le composite, et un ensemble de déclarations d'instances et de liaisons entre elles. A l'exécution, elle représente l'état d'exécution actuel d'un composite en termes d'instances.

Une instance composite est spécifie comme une instance primitive à laquelle on ajoute les caractéristiques propres à la configuration/exécution d'un composite. Elle est définie formellement de la façon suivante :

*Une **instance composite** est un 7-uplet CInst = <Inst, **mInst**, **Insts**, **Bnd**, **CP**, **CCT**, **CPF**>, où :*

Inst = <name, I, R, D, P, CT, PF> est l'instance primitive de CInst

*mInst est l'**instance principale** de CInst, tel que*
$R(Inst) \subset R(mInst)$ les ressources fournies par mInst sont au moins les ressources fournies de Inst, et
$I(mInst) = mImpl(I(Inst))$ l'implémentation de mInst est l'implémentation principale de l'implémentation de Inst

*Insts est l'ensemble des **instances contenues** dans CInst*

*$Bnd = \{b_1, b_2, \ldots, b_n\}$ est un ensemble de **bindings** entre instances contenues dans CInst, tel que*
chaque binding b_i est un 2-uplet $b = <Inst_1, Inst_2>$, où
$Inst_1 \in Insts$ est l'origine de la dépendance, et $Inst_2 \in Insts$ est la destination de la dépendance, tel que
$\exists d \in D(Inst_1)$, $target(d) = S(Inst_2) \mid target(d) \in R(Inst_2)$ il existe une dépendance de $Inst_1$ dont la destination (target) est la spécification de $Inst_2$, ou une ressource fournie par $Inst_2$

*$CP = ICP \cup InstCP$ est l'ensemble des **propriétés contextuelles** de CInst, où*
ICP est l'ensemble des propriétés contextuelles de l'implémentation de Inst, et
InstCP est l'ensemble des propriétés contextuelles définies par CInst

*$CCT = ICCT \cup InstCCT$ est l'ensemble des **contraintes contextuelles** de CInst, où*
ICCT est l'ensemble des contraintes contextuelles de l'implémentation de Inst, et
InstCCT est l'ensemble des contraintes contextuelles définies par CInst

*$CPF = ICPF \cup InstCPF$ est l'ensemble des **préférences contextuelles** de CInst, où*
ICPF est l'ensemble des préférences contextuelles de l'implémentation de Inst, et
InstCPF est l'ensemble des préférences contextuelles définies par CInst

*L'instance CInst est **conforme** à l'implémentation I de Inst : $CInst \models I$, et donc*
elle est aussi conforme à la spécification S de I : $CInst \models I \Rightarrow CInst \models S$

Notre mécanismes automatique de composition, exécuté avant ou durant la phase d'exécution, assurent la conformité d'une instance composite vis-à-vis de son composite et de sa spécification. Toutefois, toute déclaration d'instance créée avant l'exécution est vérifiée statiquement, lors de sa phase de compilation/packaging, afin d'assurer sa validité.

3.2.4 PROPRIETES PREDEFINIES

Les types de composants composites possèdent un ensemble de propriétés contextuelles prédéfinies et configurables qui permettent de spécifier la façon de gérer leur contenu. Le Tableau 10 présente ces propriétés ainsi que leur sémantique[64].

Ces propriétés peuvent être configurées à partir d'une spécification composite (comme dans la description de la Figure 57). Une implémentation composite hérite des propriétés contextuelles définies par sa spécification composite, et peut en ajouter d'autres. Une instance composite hérite des propriétés de son implémentation composite et transitivement de celles de sa spécification composite, et peut en ajouter d'autres (seulement pour le contrôle d'instances).

Chacun des composants composites, étant un sous-type d'un composant primitif, possède aussi les propriétés prédéfinies du type de composant primitif correspondant (définies dans le Tableau 9). Toutes ces propriétés sont généralement configurées lors de la création des composants.

		Propriété	***Sémantique***
Implémentation composite	***Spécification composite***	***static*** [*true* | *false* | *Expression*]	*Cette propriété indique si la résolution du composant composite (ou des certains composants contenues dans le composant composite, et spécifiés par une expression) doit être effectuée avant l'exécution.* *La valeur par défaut de cette propriété est false.*
		resolution [*open* | *closed* | *Expression*]	*Cette propriété indique si la résolution du composant composite (ou des certains composants contenues dans le composant composite, et spécifiés par une expression) considère des composants appartenant à d'autres composants composites (open). Dans les cas contraire (closed), la résolution du composant composite considère et/ou crée des composants propres.* *La valeur de cette propriété est par défaut open (comportement opportuniste).*
	Instance composite	***visibility*** [*blackbox* | *whitebox* | *Expression*]	*Cette propriété définit si tous les composants contenus dans les membres du composant composite (ou seulement des composants particuliers spécifiés par une expression) sont visibles à l'extérieur.* *La valeur par défaut de cette propriété est whitebox.*

Tableau 10. Propriétés contextuelles prédéfinies des composants à services composites

La Figure 61 présente le métamodèle qui factorise tous les concepts présentés dans le concept central de **composant à services**.

[64] Le Tableau 11 présente les propriétés contextuelles prédéfinies gérées par la machine APAM, la plate-forme sur laquelle nous avons projeté notre métamodèle (cf. Chapitre 7 section 2.1).

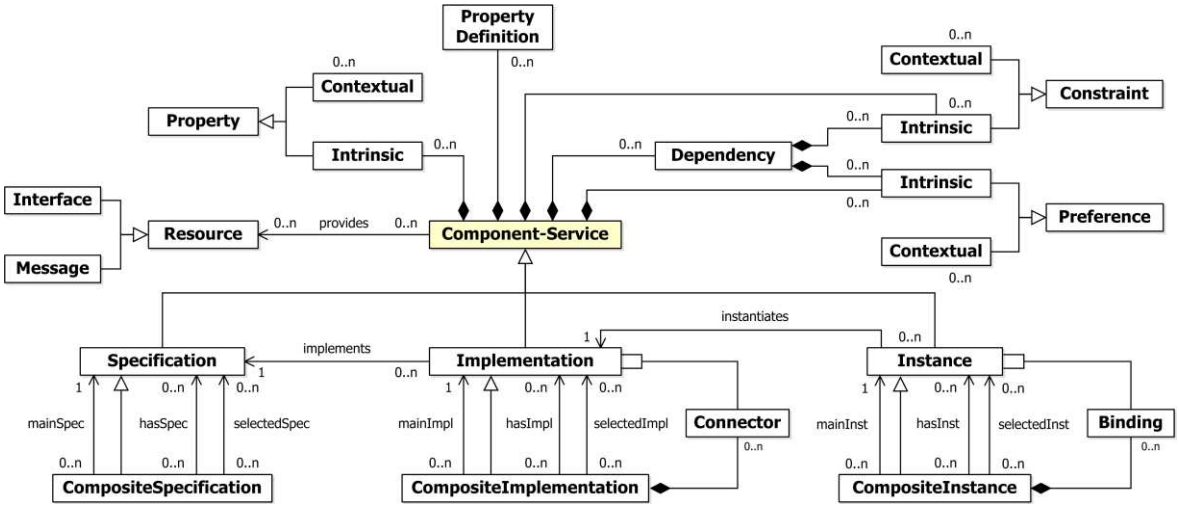


Figure 61. Métamodèle de composant à services

Notre métamodèle peut être projeté sur plusieurs langages de programmation. Particulièrement, nous l'avons projeté sur le langage de programmation Java en spécialisant les concepts du modèle pour ce langage. Par exemple, le nom de la classe Java a été ajouté comme attribut du concept d'implémentation. Ce modèle de développement permet ainsi d'implémenter un composant à services simplement comme un POJO qui ne contient que la logique métier du composant : les primitives de l'approche à services dynamique (publication, découverte, sélection, liaison) sont masquées. Les développeurs peuvent ainsi se concentrer seulement sur les descriptions des composants et sur l'implémentation de leur logique métier.

Les différentes matérialisations de composant à services, primitifs et composites, sont des éléments de premier ordre qui peuvent être décrits de manière abstraite, développés/composés, et configurés. Ces éléments peuvent être packagés afin d'être déployés, exécutés et gérés par notre environnement d'exécution : les différentes matérialisations sont toutes des instances qui coexistent dans l'environnement d'exécution.

Les concepts présents dans notre métamodèle ont une sémantique constante tout au long du cycle de vie des composants à services. Par conséquent, en plus d'être utilisé dans un but prescriptif, notre métamodèle est utilisé dans un but descriptif afin d'obtenir un modèle homogène qui représente l'état d'exécution des composants à services[65].

[65] Les relations *selectedSpec*, *selectedImpl*, *selectedInst* permettent de capturer une partie de l'information descriptive du modèle (les spécifications, implémentations et instances sélectionnées lors du processus de résolution), à la différence des relations *hasSpec*, *hasImpl* et *hasInst* qui capturent une partie de l'information prescriptive du modèle.

4 CONSTRUCTION D'APPLICATIONS

Dans cette section, nous présentons la démarche de construction d'applications à composants à services proposé dans notre approche, en détaillant chacune des phases qui la composent : conception, développement et exécution.

4.1 CONCEPTION

Notre approche propose des langages qui facilitent et fiabilisent la conception d'applications. Nous présentons brièvement notre langage de composition, puis le langage d'expression de contraintes et de préférences de sélection.

4.1.1 LANGAGE DE COMPOSITION

Nous proposons un langage pour la spécification d'applications. Ce langage permet de déclarer des spécifications composites avec leurs propriétés structurelles et sémantiques. Notre langage de composition manipule ainsi certains concepts du métamodèle de composants à services présenté précédemment, notamment les concepts liés aux spécifications composites. Il se base sur le langage d'expression de contraintes, présenté ci-dessous, pour la définition des contraintes et des préférences.

En utilisant le langage de composition, l'application *HomeMediaCenter* présentée précédemment peut être spécifiée par la description suivante :

```
CompositeSpecification HomeMediaCenter {
    property resolution = closed;
    property visibility = blackbox;
    property instantiable = true;
    attribute provider type String;
    provides media.services.manager.MediaManagerService from static MediaManager;
    contains MediaPlayer instance (usedMemory < "200");
    contains MediaRecorder (codec = "g711" AND codec = "x264");
    requires multiple MediaServer (language = "FR" OR language = "EN") id ms;
    requires optional org.osgi.log.LogService (version >= "1.0.0") id log;
}
```

Tout d'abord, cette description définit la spécification composite *HomeMediaCenter* comme une spécification instanciable (avec la propriété prédéfinie *instantiable*). Ensuite, elle définit la propriété *provider* (mot clé *attribute*). L'interface *MediaManager* est spécifiée comme la ressource fournie de *HomeMediaCenter* (mot clé *provides*). Cette interface fait partie des ressources fournies de la spécification *MediaManager* qui est spécifiée comme la spécification principale de la spécification *HomeMediaCenter* (mot clé *from*).

Les spécifications *MediaPlayer* et *MediaRecorder* sont définies comme les spécifications contenues de *HomeMediaCenter* (mot clé *contains*), et pour lesquelles des contraintes contextuelles de sélection sont spécifiées (au niveau instance et implémentation, respectivement). Ensuite, deux dépendances de *HomeMediaCenter* sont définies (mot clé *requires*), l'une (obligatoire et multiple) vers la spécification *MediaServer* et l'autre (optionnelle et simple) vers l'interface *LogService*. En addition, des contraintes de sélection ont été définies sur ces deux dépendances.

Enfin, des propriétés pour la résolution de la spécification *HomeMediaCenter* (mots clés *static* et *closed*) ainsi que pour la visibilité de ses composants (mot clé *black-box*) sont spécifiées.

La validité (cohérence) des descriptions créées avec le langage de composition est vérifiée et assurée, afin créer la spécification composite correspondante (toutefois, la validité de toute spécification composite est vérifiée et assurée lors de sa phase de compilation, packaging, et déploiement). La syntaxe et la sémantique du langage de composition sont détaillées dans l'Annexe A.

4.1.2 LANGAGE DE SELECTION

Notre langage de sélection est un langage générique qui, comme le langage OCL (*Object Constraint Language*) utilisé par UML, permet d'exprimer des contraintes et de préférences sur les éléments d'un modèle. Le langage permet la navigation de modèles (des éléments liés) et l'évaluation de contraintes et de préférences sur les propriétés des éléments. Comme dans OCL, les contraintes peuvent être utilisées pour assurer la cohérence d'un modèle. Contrairement à OCL, les contraintes peuvent être associées à des types et à des instances. Ainsi, dans notre métamodèle, des contraintes et des préférences peuvent être associées à des spécifications et des implémentations (en tant que types), mais aussi à des implémentations et à des instances (en tant qu'instances respectivement des types spécification et implémentation).

Notre langage s'appuie sur le concept de groupe d'équivalence. Les expressions sont ainsi fortement typées : une expression peut faire référence à la propriété *a* d'un élément *X* seulement si *X* (étant un type) ou le type de *X* (i.e., son représentant) a déclaré la propriété *a*. Ainsi, la validité des expressions peut être vérifiée statiquement, et la compatibilité entre éléments peut donc être assurée.

Les contraintes et les préférences peuvent être exprimées en utilisant des expressions similaires aux expressions LDAP, des expressions de navigation, ou des constructions complexes (voir [84] [85] pour plus de détails). Les contraintes définies par un élément peuvent être utilisées pour imposer des propriétés fonctionnelles ou non fonctionnelles à d'autres éléments. Par exemple, l'implémentation *ADELE-MediaManager* définit les contraintes suivantes :

```
requires MediaServer (streaming=true);
requires MediaRecorder (audio=true AND video=true);
```

La première contrainte exprime que l'implémentation *ADELE-MediaManager* requiert un fournisseur de la spécification *MediaServer* ayant la propriété *streaming* égale à *true.* Cette contrainte est valide parce que la propriété *streaming* est définie par la spécification *MediaServer*. La deuxième contrainte établit que *ADELE-MediaManager* requiert un fournisseur de la spécification *MediaRecorder* ayant les propriétés *audio* et *video* égales à *true*. Cette contrainte est aussi valide car les propriétés *audio* et *video* sont définies par la spécification *MediaRecorder*.

Les contraintes définies par un composant, comme dans les deux exemples précédents, sont des contraintes intrinsèques au composant et doivent être respectés dans n'importe quel contexte ou application où le composant soit utilisé.

Les composants composites peuvent définir des contraintes afin d'imposer des propriétés aux éléments qu'il utilise (i.e., aux composants qui participeront à sa composition). Ces contraintes, nommées contraintes contextuelles, ne sont pertinentes que pour le composant composite qui les spécifie. Considérons par exemple le composite *ADELE-MediaCenter* présenté précédemment (voir la Figure 59). Étant associé avec la définition de composite *HomeMediaCenter*, ce composite hérite des contraintes contextuelles suivantes :

```
contains MediaPlayer instance (usedMemory < "200");
contains MediaRecorder (codec = "g711" AND codec = "x264");
requires multiple MediaServer (language = "FR" OR language = "EN") id ms;
requires optional org.osgi.log.LogService (version >= "1.0.0") id log;
```

Les implémentations contenues dans un composite doivent satisfaire les contraintes contextuelles et intrinsèques du composite mais aussi des implémentations concernées contenues dans le composite. Par exemple, le composite *ADELE-MediaCenter* possède la contrainte contextuelle (*language="FR" OR language="EN"*) associée à sa dépendance *MediaServer*. L'implémentation *ADELE-MediaManager*, contenue dans ce composite, a la contrainte intrinsèque (*streaming=true*) associée à sa dépendance *MediaServer*. Ces contraintes seront agrégées afin d'être évaluées lors de la sélection d'une implémentation *MediaServer* pour ce composite.

De manière similaire, les préférences peuvent être intrinsèques ou contextuelles. Les préférences sont évaluées seulement si plusieurs fournisseurs satisfont l'ensemble des contraintes donné. Le fournisseur satisfaisant les plus de préférences sera sélectionné. Par exemple, le composite *ADELE-MediaCenter* (voir la Figure 59) définit une préférence contextuelle qui exprime que la sélection d'un fournisseur *MediaServer* satisfaisant l'expression (*language="FR")* est préférable.

```
prefers MediaServer (language="FR");
```

Les contraintes et les préférences, intrinsèques et contextuelles, sont vérifiées et validées au moment de leur définition, et évaluées lors de la composition du composite et de l'instance de composite associés, garantissant la sélection des implémentations et des instances qui satisfont les propriétés requises et préférables.

4.2 DEVELOPPEMENT ET EXECUTION

Notre approche propose des mécanismes pour le développement et l'exécution d'applications décrites avec notre langage de composition. Dans la phase de développement, ces mécanismes permettent la composition (dite statique) d'applications. Dans la phase d'exécution, ces mécanismes permettent la composition et l'évolution (dites dynamiques) d'applications.

4.2.1 COMPOSITION

Le processus de composition d'applications à services, réalisé dans la phase de développement (composition statique) ou dans la phase d'exécution (composition dynamique), se base sur le même processus de résolution de groupes de services. Rappelons que le principe de résolution permet de passer d'un niveau d'abstraction à un niveau plus concret en considérant les éléments contenus dans un espace de sélection.

Résoudre une spécification (un groupe *Specification*) consiste à sélectionner une ou plusieurs implémentations (primitives ou composites), parmi un ensemble d'implémentations contenues dans un ou plusieurs espaces de sélection, qui satisfont les contraintes de cardinalité et de sélection données, ou à les générer/créer. Lors du processus de résolution d'une spécification, une implémentation peut être générée si la spécification est instanciable (la spécification est associée à une fabrique capable de générer des implémentations qui satisfont un ensemble de contraintes).

De façon similaire, résoudre une implémentation (un groupe *Implementation*) consiste à sélectionner une ou plusieurs instances (primitives ou composites), parmi un ensemble d'instances contenues d'un ou plusieurs espaces de sélection, qui satisfont les contraintes données, ou à les créer si la sélection échoue. Une instance peut être créée lors du processus de résolution d'une implémentation si l'implémentation est instanciable. Dans la phase de développement, résoudre une implémentation consiste à sélectionner ou à créer une ou plusieurs déclarations d'instances.

Pour les spécifications composites et les implémentations composites, le processus de résolution est le même. Cependant, la création de membres implique la résolution des groupes internes du composite. Dans notre approche, les spécifications composites et les implémentations composites sont instanciables, permettant ainsi la création d'implémentations composites et d'instances composites. La résolution d'une spécification composite a pour résultat une ou plusieurs implémentations composites (sélectionnés ou crées) qui satisfont les propriétés, les contraintes et les préférences données, y comprises celles contextuelles. La création d'une implémentation composite requiert de définir au moins son implémentation principale.

De manière similaire, la résolution d'une implémentation composite, a pour résultat une ou plusieurs instances composites (sélectionnés ou crées) qui satisfont les propriétés, les contraintes et les préférences données, y comprises celles contextuelles. La création d'une instance composite requiert de résoudre au moins son instance principale.

4.2.1.1 COMPOSITION STATIQUE

La composition d'applications réalisée lors de la phase de développement, appelée composition statique, permet d'obtenir – à partir d'une spécification composite – une implémentation composite et une configuration composite représentant les architectures statiques de l'application, en termes d'implémentations et de déclarations d'instances. Lors de la composition statique d'une application :

- la résolution d'un service est réalisée par défaut de façon partielle mais elle peut être spécifiée comme étendue et/ou comme complète (cf. point (1) de la résolution réalisée au développement, défini dans la section 3.2 du Chapitre 5) ;
- la résolution est réalisée de façon immédiate pour les services définis comme statiques (cf. point (2)) ;
- la résolution d'un service peut être fermée et/ou coopérative considérant ainsi l'espace de travail et/ou des dépôts locaux/distants de composants (cf. point (3)).

4.2.1.2 COMPOSITION DYNAMIQUE

La composition d'applications réalisée lors de la phase d'exécution, appelée composition dynamique, permet d'obtenir une implémentation composite et une instance composite représentant les architectures de l'application exécutée. Lors de la composition dynamique d'une application :

- la résolution d'un service est réalisée de façon complète : le résultat de la résolution est toujours une instance de service (sélectionné ou créée) (cf. point (1) de la résolution réalisée à l'exécution, défini dans la section 3.3 du Chapitre 5) ;
- la résolution d'un service est réalisée par défaut de façon paresseuse, mais elle peut être spécifiée comme immédiate, ou comme adaptative (cf. point (2)) ;
- la résolution d'un service peut être fermée, coopérative mais aussi opportuniste (cf. point (3)).

La composition dynamique entraîne la réalisation des différentes actions sur le modèle d'état. De telles actions sont associées aux propriétés et aux opérations de manipulation des composants. Les opérations de manipulation permettent ainsi de modifier l'architecture de l'état d'exécution et d'enrichir sa sémantique. En termes de groupes, les opérations permettent :

- la création d'un membre à partir de son représentant,
- la destruction d'un membre, et
- l'ajout, la modification et la suppression de propriétés et de relations (liaisons). Ces opérations sont limitées aux aspects suivants :
    - un membre ne peut pas supprimer/modifier une propriété statique héritée de son représentant,
    - un membre peut modifier une propriété configurable héritée de son représentant, mais jamais la supprimer, et
    - certaines propriétés configurables définies par un représentant peuvent seulement être configurées à la création d'un membre et ne peuvent pas être modifiées.

Pour les instances, les opérations de manipulation sont associées aux objets qui représentent les services et qui permettent leur invocation. En effet, une instance est associée à un objet qui représente le service réel.

Nos mécanismes de composition, présents dans les phases de développement et d'exécution, garantissent la conformité des compositions. De plus, nos mécanismes présents dans la phase d'exécution, vérifient la conformité de l'état d'exécution d'une application (i.e., du modèle d'état de l'application) à son modèle d'application suite aux changements dans l'état d'exécution.

En effet, le modèle d'application et le modèle d'état d'une application possèdent des relations de correspondance entre leurs composants qui permettent de contrôler la composition de l'application. De telles relations permettent d'établir la relation de conformité du modèle d'état au modèle d'application : le modèle d'état d'une application est ***conforme*** à un instant donné au modèle d'application si aucune propriété spécifiée par le modèle d'application n'est fausse dans le modèle d'état. Face à la non-conformité du modèle d'état, et dans le but de garantir sa conformité, des actions (création, destruction, modification de composants et/ou de liaisons) doivent être effectuées pour l'adapter.

4.2.2 EVOLUTION DYNAMIQUE

Les informations produites à la conception et au développement, étant connues et gérées à l'exécution, l'**évolution dynamique** d'une application est possible. Cette propriété est fondamentale pour les applications dynamiques et autonomiques afin de corriger, améliorer, étendre ou réduire leurs fonctionnalités.

Notre approche permet l'évolution dynamique d'une application par la modification de son modèle d'application. L'évolution de l'application implique la vérification de la conformité du modèle d'état, pouvant entraîner l'adaptation de l'application.

4.3 VISIONS TOP-DOWN ET BOTTOM-UP

Notre approche intègre les visions *top-down* et *bottom-up* pour la construction d'applications. La vision *bottom-up* est la vision traditionnelle des plates-formes à services. Dans cette vision il n'y a pas de définition préalable d'application. Les services en exécution demandent à la plate-forme de se lier à d'autres services. L'exécution de l'application « émerge » de cet ensemble de liaisons établies dynamiquement à la demande des composants. Dans notre approche, les applications en cours d'exécution sont réifiées dans un modèle homogène et causal conforme à notre métamodèle de composants à services : le modèle d'état.

Dans une vision *top-down*, les informations produites à la conception, développement et configuration d'une application sont connues et gérées lors de l'exécution. La composition dynamique d'une application est réalisée par rapport à sa définition et en considérant les informations effectives du modèle d'état produit par la vision *bottom-up*.

5 SYNTHESE

Dans les approches traditionnelles de construction d'applications, une application est définie comme une entité composite constituée de tous les composants qui la composent ainsi que leurs connections. Comme nous l'avons vu, cette façon de définir une application n'est pas adéquate pour les applications où les composants qui peuvent participer à leur composition sont inconnus au moment de la conception ; et où la disponibilité des composants peut varier de manière imprévisible au cours de l'exécution.

Nos travaux visent ainsi à construire des applications à services flexibles afin de faire face au dynamisme des services, tout en rendant leur construction plus simple et plus fiable. Nous proposons de définir une application à services par intention, et de réaliser sa composition de façon graduelle et contrôlée dans la phase de développement et/ou dans la phase d'exécution. Pour ce faire, nous avons défini le concept de composant à services à trois niveaux d'abstraction : spécification, implémentation et instance, couvrant ainsi le cycle de vie des composants à services.

Les différentes matérialisations du concept de composant à services, primitifs et composites, sont des entités de premier ordre qui peuvent être décrites, développées/composées et exécutées.

Notre approche permet de spécifier une application de façon traditionnelle, par un ensemble d'implémentations interconnectées. Par rapport aux approches traditionnelles, nous ajoutons deux niveaux d'abstraction. Une application peut être spécifiée :

- par des contraintes architecturales uniquement (i.e., une spécification composite), ou
- de façon très précise en termes d'instances interconnectées (i.e., une instance composite).

Par rapport aux approches traditionnelles, nous ajoutons deux mécanismes fondamentaux :

- tout composite peut être incomplet : il peut ne pas indiquer tous les composants qu'il doit contenir, et
- tout composite (quel que soit son niveau, complet ou non) est exécutable. Dans tous les cas, la plate-forme d'exécution assure une exécution conforme.

Le modèle de développement proposé simplifie la conception et l'implémentation des composants ; les architectes et les développeurs peuvent ainsi se concentrer « seulement » sur les descriptions des composants et sur l'implémentation de leur logique métier.

CHAPITRE 7
REALISATION

Ce chapitre présente la mise en œuvre de notre proposition. Nous présentons notre système COMPASS[66] qui gère les phases de conception, de développement et d'exécution des applications. COMPASS est composé de différents environnements associés à ces phases (voir la Figure 62). Dans une première partie, nous introduisons la technologie CADSE qui permet la création d'environnements dédiés à domaines, puis, nous présentons les caractéristiques principales de notre environnement COMPASS-CADSE pour la conception et le développement de services et d'applications à services. Dans la deuxième section, nous présentons l'environnement d'exécution, nommé COMPASS-RT. Cet environnement repose sur APAM, une plate-forme extensible pour l'exécution d'applications à services. COMPASS-RT étend les fonctionnalités d'APAM afin de contrôler l'exécution des applications construites avec COMPASS-CADSE.

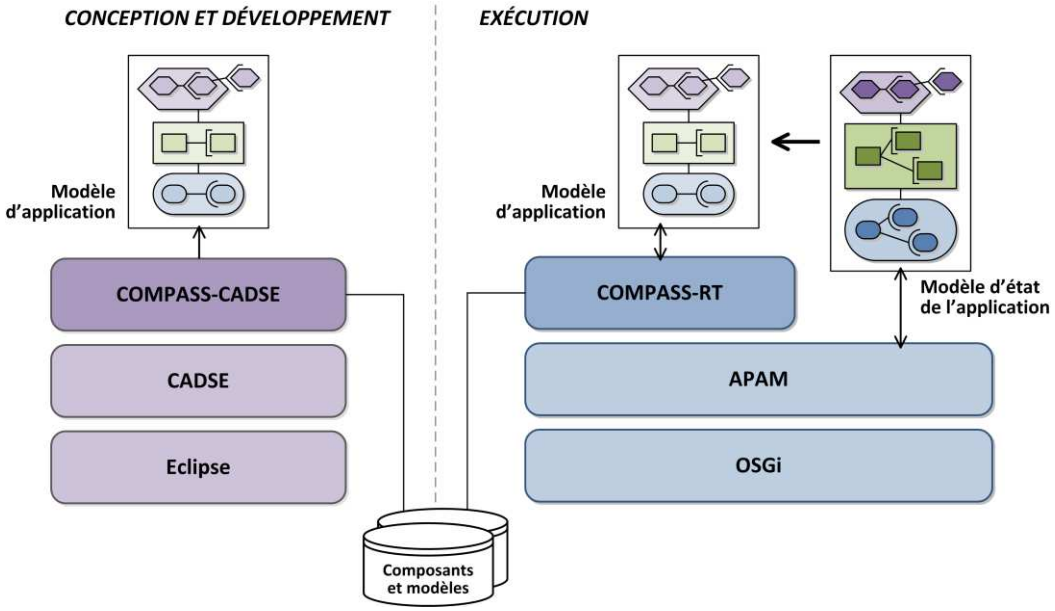


Figure 62. COMPASS

[66] Le nom COMPASS (mot anglais pour compas) s'inspire de la définition propre du mot (instrument qui donne une référence de direction) et réfère à un système qui guide la construction d'applications

1 CONCEPTION ET DEVELOPPEMENT

La construction d'applications est une tâche complexe face à laquelle les architectes et les développeurs sont confrontés. Le besoin d'environnements et d'outils de conception et de développement dédiés se fait ressentir afin d'assister les architectes et les développeurs et de faciliter ainsi la construction d'applications.

Toutefois, la réalisation d'environnements dédiés est complexe et couteuse. En effet, ces environnements doivent satisfaire les exigences suivantes :

- **la spécialisation au domaine d'application** : un environnement doit permettre aux utilisateurs de manipuler les concepts (du domaine) qui leur sont familiers.
- **l'abstraction** : un environnement doit abstraire la complexité technologique liée aux modèles de programmation.
- **la séparation des préoccupations** : un environnement doit fournir un ensemble de vues permettant de séparer les concepts et les fonctionnalités en fonction des rôles des utilisateurs et des tâches qu'ils doivent réaliser.
- **l'extensibilité** : un environnement doit être facilement extensible pour permettre l'introduction de nouveaux concepts et/ou de nouvelles fonctionnalités.

Dans l'objectif de créer des environnements satisfaisant ces besoins avec un temps et un coût de réalisation raisonnables, notre équipe de recherche ADELE propose une approche générative d'environnements dédiés, nommés CADSEs[67] [86].

En utilisant cette approche, nous fournissons des environnements qui mettent en œuvre notre proposition pour la construction d'applications à services. Ces environnements sont spécialisés pour la réalisation des tâches spécifiques à chaque phase du cycle de vie d'une application. Notre environnement COMPASS-CADSE intègre ces environnements couvrant ainsi de la conception au déploiement des applications. Dans les sections suivantes nous détaillons l'approche CADSE et notre environnement COMPASS-CADSE.

1.1 CADSES : ENVIRONNEMENTS DEDIES

CADSEg (CADSE *generator*) est un environnement dirigé par les modèles qui permet de spécifier des environnements dédiés à la réalisation d'applications spécifiques à un domaine particulier, nommés CADSEs, et de les générer à partir de ces spécifications. CADSEg est lui-même un CADSE dédié à la réalisation d'environnements spécialisés au domaine particulier des CADSEs.

CADSEg a été conçu au-dessus de l'environnement Eclipse, l'étendant ainsi avec un ensemble de concepts et de fonctionnalités. CADSEg est un logiciel libre fourni sous la forme de *plug-ins* et de *features* pour Eclipse (téléchargeable depuis le site http://cadse.imag.fr).

CADSEg fournit un ensemble d'outils génériques incluant des éditeurs de modèles, des validateurs de modèles et de générateurs de code à partir des modèles. Sur la base de ces outils génériques, CADSEg permet de spécifier un domaine particulier (le modèle du domaine) et de générer des outils spécifiques au domaine : des éditeurs du modèle, des *builders*[68] automatiques. CADSEg gère automatiquement la synchronisation entre les modèles de domaine et les artefacts générés correspondants (projets, répertoires, fichiers,…) afin de garantir leur cohérence.

CADSEg est constitué de l'ensemble des modèles suivants :

- **le modèle de données** (*data-model*) qui permet de spécifier l'ensemble des concepts propres au domaine (y comprises des entités composites regroupant un ensemble de

[67] CADSE est l'acronyme en anglais pour *Computer Aided Domain Specific Environment*
[68] Un *builder* est un objet qui manipule des artefacts dans l'environnement et crée d'autres artefacts

concepts), leurs attributs et leurs relations (de dépendance, d'agrégation, de composition, d'annotation, de groupe) avec d'autres concepts. Les concepts propres à un domaine se basent sur l'ensemble des concepts de base de CADSEg : *ItemType*, *AttributeType* et *LinkType* (voir la Figure 63).

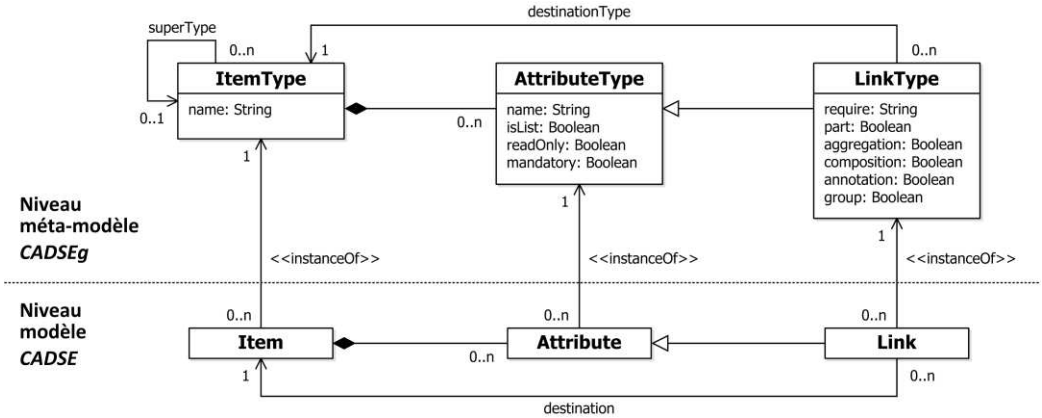


Figure 63. Métamodèle et modèle de données de CADSE

- **le modèle de correspondances** (*mapping*) qui permet de définir les correspondances entre les concepts du modèle de données et les artefacts Eclipse, ainsi que la synchronisation qui sera effectuée. Un modèle de correspondances (*ContentItemType*) est donc associé à chaque type d'entité (*ItemType*). Toutefois, un type d'entité peut ne pas disposer de modèle de correspondances (voir la Figure 64).

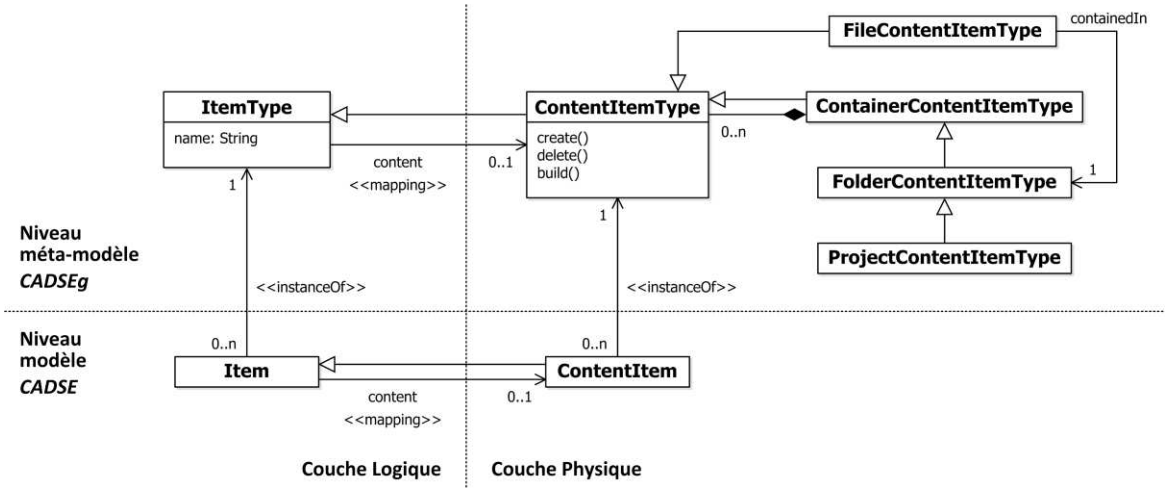


Figure 64. Métamodèle et modèle de correspondances de CADSE

- **le modèle d'interactions** (*view-model*) qui permet de définir des éditeurs avec des actions propres au domaine (vues spécialisées). Par exemple, un CADSE peut définir dans son modèle d'interactions des vues spécialisées pour développer les entités d'une application, les tester et les déployer.
- **le modèle de construction** (*build-model*) qui permet de spécifier les mécanismes de construction, de compilation et de déploiement mis en œuvre dans le domaine.
- **le modèle d'évolution** qui permet de définir les politiques d'évolution et de gestion de versions des entités du domaine.

En utilisant ces modèles, CADSEg permet aux experts du domaine de spécifier un CADSE. La Figure 65 montre les vues des modèles de données, de correspondances et d'interactions spécifiant notre CADSE de développement de services.

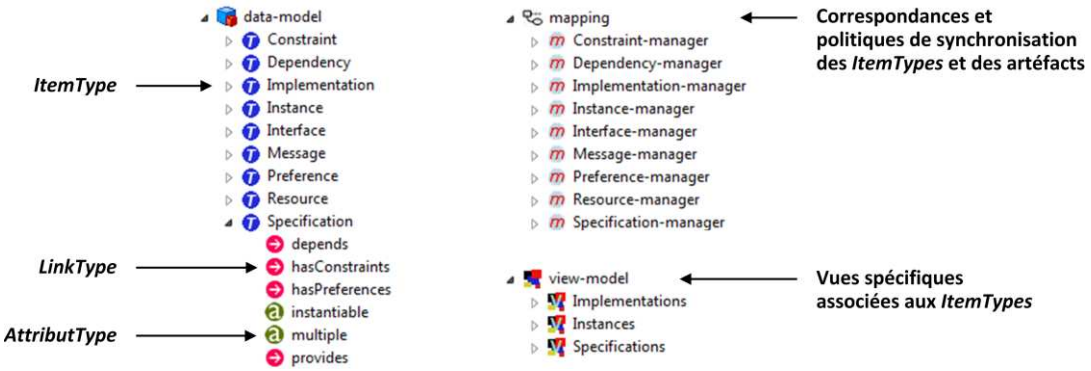


Figure 65. Vues des modèles d'un CADSE

CADSEg génère à partir des modèles spécifiés l'environnement CADSE correspondant (voir la Figure 66).

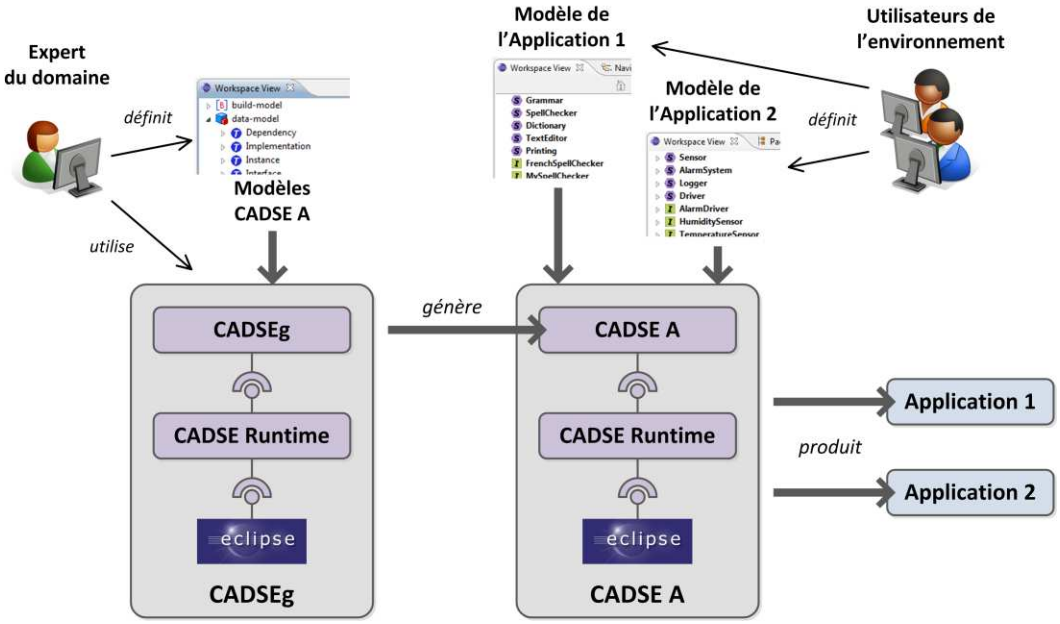


Figure 66. Approche générative de CADSE

Les utilisateurs d'un CADSE manipulent ainsi les concepts du domaine plutôt que de manipuler directement les artéfacts Eclipse.

Les architectes et les développeurs d'applications peuvent définir et réaliser plus facilement des applications, en définissant des modèles d'application. Selon ses objectifs, un CADSE peut générer, à partir d'un modèle d'application, le code exécutable de l'application et les artéfacts Eclipse nécessaires. Le code généré peut être étendu afin de préciser le comportement souhaité d'une application.

Un CADSE est associé à un espace de travail (*workspace*) contentant l'ensemble des éléments du domaine et les artéfacts Eclipse correspondants. CADSE gère automatiquement l'espace de travail grâce à la synchronisation entre les éléments du domaine et les artefacts Eclipse correspondants : les

modifications effectuées sur les éléments du domaine sont traduites en modifications sur les artefacts associés, et inversement. De plus, CADSE fournit des mécanismes d'importation et d'exportation des éléments d'un espace de travail : les éléments crées dans l'espace de travail d'un CADSE peuvent être exportés, et importés dans l'espace de travail d'un autre CADSE.

En résumé, l'approche CADSE fournit des fonctionnalités et de mécanismes puissants qui permettent la réalisation rapide et simple d'environnements dédiés à un domaine spécifique. Parmi ses avantages nous remarquons :

- **l'augmentation du niveau d'abstraction** : les utilisateurs manipulent les concepts du domaine plutôt que les artéfacts Eclipse.
- **la gestion des correspondances entre les éléments du domaine et les artéfacts Eclipse** : la synchronisation entre les éléments et les artéfacts est automatiquement gérée afin d'assurer leur cohérence.
- **la génération de code** : du code exécutable peut être généré à partir d'un modèle d'application et précisé par les utilisateurs. Le modèle d'application étant lié par des relations de correspondance au code généré, leur cohérence est assurée.
- **la séparation de préoccupations** : différentes vues séparant les concepts et les fonctionnalités des utilisateurs, en fonction de leurs rôles et des tâches qu'ils doivent réaliser, peuvent être mises à disposition.
- **l'extensibilité** : des nouveaux concepts, attributs, relations, fonctionnalités,... peuvent être ajoutés.
- **la composition** : des environnements complexes peuvent être définis à partir des environnements existants.

Motivés par ces avantages, nous avons utilisé l'approche CADSE pour créer des environnements dédiés à la réalisation des tâches spécifiques aux différentes phases du cycle de vie des applications à services. Ces environnements mettent en œuvre notre approche et facilitent son utilisation. De plus, ils nous permettent de réaliser des expérimentations afin de valider notre approche.

1.2 COMPASS-CADSE

COMPASS-CADSE est un environnement dédié à la conception et le développement d'applications à services réalisés selon notre approche. COMPASS-CADSE permet aux architectes de définir en intention des applications et des sous-systèmes à services à travers la description de spécifications composites. En utilisant notre langage de composition, le processus de définition d'applications est non seulement plus simple pour les architectes, mais aussi plus sûr : la validité des descriptions de spécifications composites est vérifiée et assurée.

COMPASS-CADSE permet de définir des applications en extension par les processus de développement et de configuration. Ces processus peuvent être réalisés de façon manuelle (les développeurs définissent les implémentations et les déclarations d'instances d'une application) et/ou automatisée (le mécanisme de composition sélectionne/crée les implémentations et les déclarations d'instances nécessaires, en considérant les composants contenus dans un ensemble de dépôts disponibles, y compris l'espace de travail). COMPASS-CADSE assure la conformité de la composition d'une application (de son développement et de sa configuration) par rapport à sa spécification.

COMPASS-CADSE permet de définir un modèle d'application à partir d'une définition en intention et d'une définition en extension correspondante. Le modèle d'application est un modèle exécutable interprété à l'exécution par notre gestionnaire d'applications COMPASS-RT.

COMPASS-CADSE permet le packaging de modèles d'application ainsi que le placement des packages dans des dépôts de composants afin de les déployer et les exécuter sur la plate-forme d'exécution sous-jacente.

Comme nous l'avons mentionné précédemment, notre machine d'exécution est basée sur la plate-forme APAM (présentée dans la section 2.1). De ce fait, le développement et le packaging implantés dans COMPASS-CADSE sont spécifiques à APAM. Toutefois, COMPASS-CADSE est basé sur une architecture modulaire : les concepts cœur de notre approche sont indépendants des concepts de développement et de packaging, ce qui permet d'implanter d'autres modules de développement et de packaging spécifiques à d'autres plates-formes d'exécution (comme OSGi, ou iPOJO).

De plus, le concept de composant à service primitif peut être géré indépendamment du concept de composant à service composite : l'environnement de conception et de développement de composants primitifs est indépendant de celui des composants composites. Séparer ces environnements offre ainsi la possibilité de définir et de développer des composants primitifs indépendamment des applications ou des sous-systèmes qui peuvent les utiliser.

La Figure 67 présente l'architecture de COMPASS-CADSE.

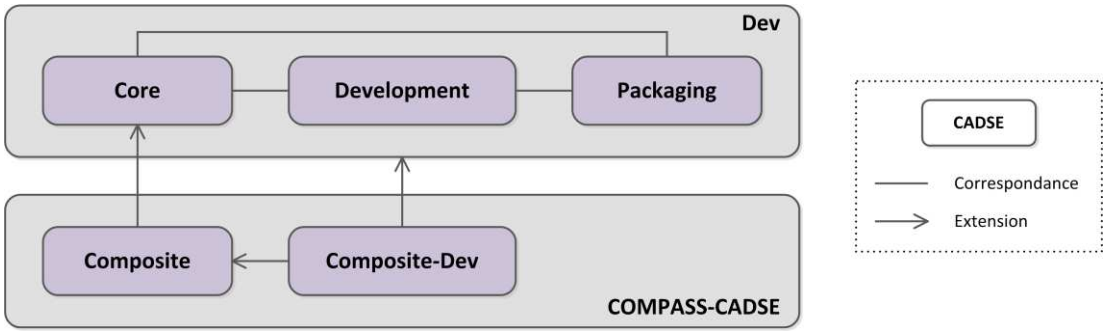


Figure 67. Architecture de COMPASS-CADSE

Nous distinguons les environnements CADSE suivants :

- **Core** permet la création de spécifications, d'implémentations et de déclarations d'instances de composants à services. Son modèle de données correspond à notre métamodèle de composants à services primitifs (présenté dans la section 3.1 du chapitre 6). Core utilise un outil pour l'analyse et la validation syntaxique des contraintes et des préférences des composants. Il offre des vues, des menus et des pages pour la création et l'affichage des composants.
- **Development** permet la génération d'artefacts de développement de base : fichiers, répertoires, projets, etc. Grâce au mécanisme de composition d'environnements de CADSE, des relations de correspondance entre les concepts de Core et ceux de l'environnement de développement peuvent être définies. En effet, les concepts de spécification, d'implémentation et d'instance correspondent à des projets Maven (Java), et à des fichiers de description XML spécifiques à la plate-forme APAM contenus dans ces projets.
- **Packaging** permet de créer des unités de packaging et de déploiement. Des relations de correspondance peuvent être définies des concepts de Core et Developpement vers les concepts de l'environnement de packaging afin de spécifier la façon de les packager et de les déployer. Concrètement, les concepts de spécification, d'implémentation et d'instance (avec leurs projets correspondants) sont packagés dans des fichiers JAR correspondant à des *bundles* de la plate-forme APAM.
- **Dev** compose les environnements présentés ci-dessus, Core, Developement et Packaging, en établissant des relations de correspondance entre leurs concepts. Il permet ainsi la création de composants à services primitifs et la création automatique de leurs artéfacts de développement pour la plate-forme APAM, par exemple, lors de la création d'une spécification un projet Maven est créé contenant des fichiers Java (interfaces et classes), un fichier XML avec la description de la spécification. Il assure la synchronisation entre

les composants à services et leurs artéfacts de développement. De plus, il permet le packaging des composants (création des *bundles* APAM).

- **Composite** permet la définition de spécifications, d'implémentations et de instances composites. Son modèle de données étend celui de l'environnement Core en intégrant les concepts correspondants aux composants à services composites (présentés dans la section 3.2 du chapitre 6). Cet environnement inclut divers outils pour la création de composants composites : analyseur et validateur syntaxique de spécifications composites, analyseur et validateur syntaxique de contraintes et de préférences, moteur de composition. Il offre des vues, des menus et des pages pour la création, l'affichage, l'édition de composants composites.
- **Composite-Dev** étend les environnements Composite et Dev afin de permettre la création d'artefacts de développement (projets, fichiers de description) et de packaging (*bundles* APAM) correspondants aux composants composites. Il gère la synchronisation entre les composants composites et leurs artéfacts de développement.

L'environnement **COMPASS-CADSE** est constitué des environnements Composite et Composite-Dev et supporte ainsi la création de composants à services composites, et de leurs artefacts de développement et de packaging. Il permet en addition la création de modèles d'application ainsi que leur packaging.

Comme tout CADSE, COMPASS-CADSE est extensible, permettant ainsi à d'autres environnements d'ajouter des concepts, des fonctionnalités et des outils. Nous évoquons particulièrement des environnements permettant la définition de préoccupations non-fonctionnelles pour les applications à services.

La syntaxe de nos langages de composition et d'expression de contraintes et préférences a été exprimée dans le métalangage EBNF[69]. L'outil JavaCC[70] a été utilisé pour la mise en œuvre de nos langages. JavaCC est un logiciel pour Java qui prend comme entrée la spécification d'une grammaire et génère le parseur correspondant. Par conséquent, les parseurs correspondants à nos langages permettent d'analyser et de valider la syntaxe des contraintes et des préférences exprimées, ainsi que des déclarations des spécifications composites.

Étant donné que la syntaxe du langage d'expression de contraintes et préférences est générique, son analyseur syntaxique (fourni sous la forme de plugin Eclipse et *bundle* OSGi) est indépendant de nos environnements et réutilisable. En effet, nos environnements utilisent cet analyseur en le personnalisant à leurs besoins particuliers.

1.3 EXEMPLE : MISE EN ŒUVRE D'UNE APPLICATION DANS COMPASS-CADSE

Considérons l'application de gestion multimédia pour des passerelles résidentielles décrite dans la section 1 du Chapitre 6. L'objectif de cette application est de permettre aux utilisateurs de naviguer, de sélectionner et de reproduire des fichiers multimédia (image, son, vidéo) en utilisant les différents équipements électroniques audio et vidéo disponibles dans la maison (téléviseurs, haut-parleurs, écrans, tablettes, etc.).

Avec COMPASS-CADSE :

- l'application est définie par une description en utilisant notre langage de composition ;
- la validité de la description de l'application est assurée : compatibilité des composants contenus, validité des propriétés, contraintes et préférences exprimées ;

[69] *Extended Backus-Naur Form*
[70] **http://javacc.java.net/**

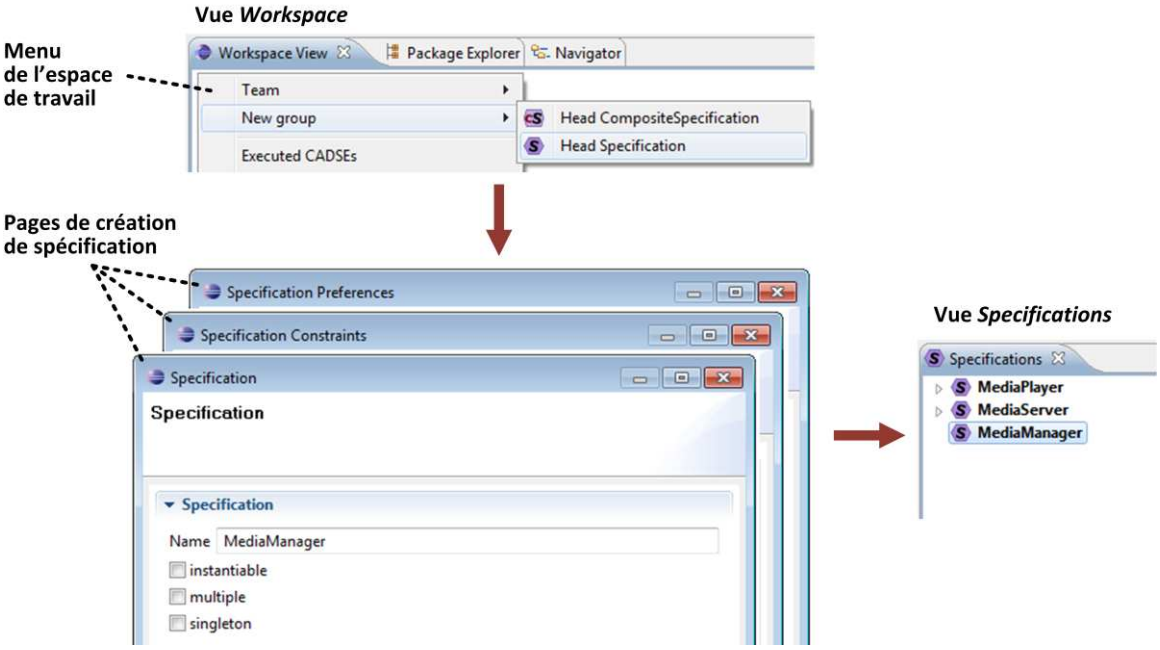
- la conformité de la composition de l'application (de son développement et de sa configuration) par rapport à sa spécification est assurée ;
- les artéfacts de développement nécessaires (projets et fichiers de description) sont générés automatiquement : le développeur est juste chargé de réaliser l'implémentation des fonctionnalités de l'application ;
- les artéfacts de packaging sont générés automatiquement afin de les déployer sur la passerelle résidentielle.

Les figures suivantes montrent différentes captures d'écran de l'environnement COMPASS-CADSE (menus, pages, vues) lors de la réalisation de l'application de gestion multimédia. Nous montrons tout d'abord la création de certains composants primitifs qui permettront ensuite la création de l'application.

La Figure 68 montre le processus de construction de la spécification *MediaManager* en deux étapes : (1) la création de l'item *MediaManager* de type *Specification*, et (2) l'ajout de ses caractéristiques (ressources fournies, propriétés et ressources requises). Toutes ces informations permettent de générer le fichier de description XML présenté dans la Figure 52.

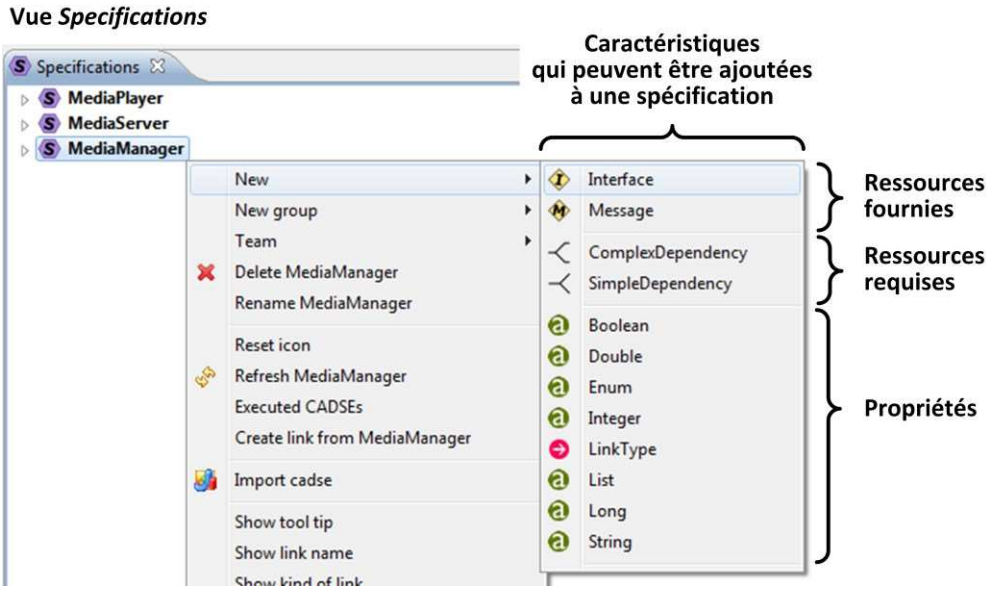
(1) Création de l'item ***MediaManager*** :

Une spécification est créée en utilisant le menu de la vue « Workspace ». Différentes pages de création sont présentées afin de définir les propriétés de la spécification (nom, propriétés prédéfinies, contraintes, préférences). La vue « Specifications » montre les spécifications présentes dans l'espace de travail.



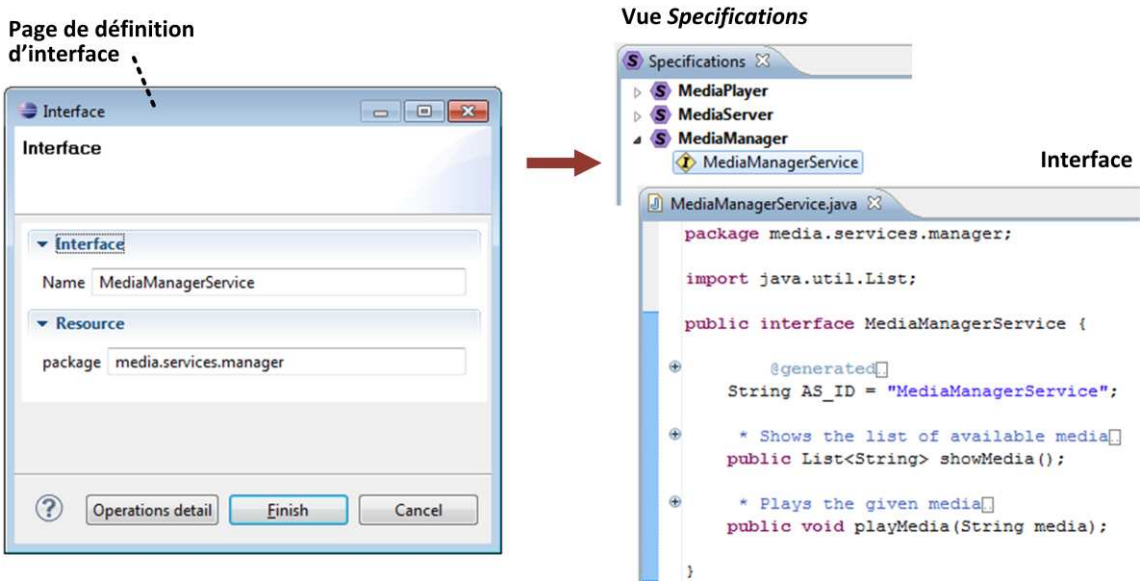
(2) Ajout de caractéristiques à ***MediaManager*** :

Diverses caractéristiques peuvent être ajoutées à une spécification en utilisant son menu : ressources fournies (interfaces, messages), ressources requises (dépendances simples) spécifications requises (dépendances complexes), propriétés de types différents (Boolean, Double, Enum, Integer, ...).



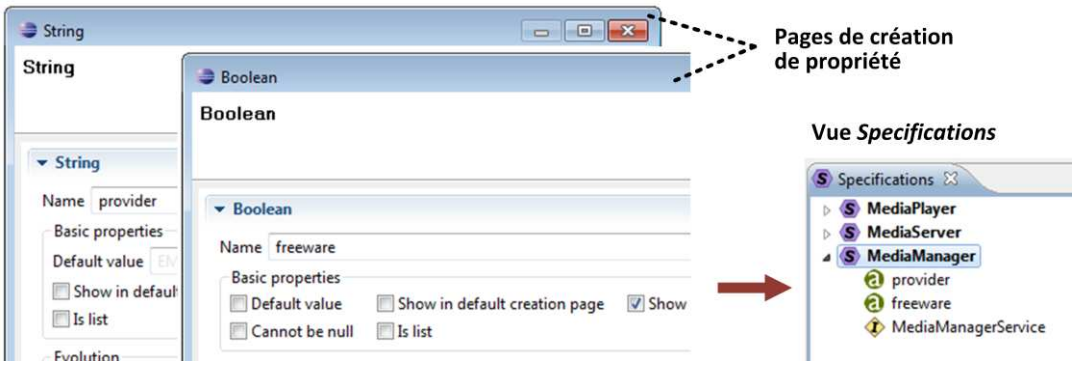
- Ajout de l'interface fournie ***MediaManagerService*** :

Une interface est spécifiée par son nom et son package.



- Ajout des propriétés ***provider*** (de type *String*) et ***freeware*** (de type *Boolean*) :

  Une propriété est définie, à partir d'un type, avec son nom et ses propriétés (valeur par défaut, multiplicité, possibilité de valeur nulle, visibilité dans les pages de création/modification, ...).



- Ajout des dépendances vers les spécifications ***MediaServer*** et ***MediaPlayer*** :

  Une dépendance peut être simple ou complexe. Une dépendance d'une spécification est définie par son nom, sa destination (une ressource dans le cas d'une dépendance simple ; une spécification dans le cas d'une dépendance complexe), sa multiplicité, ses contraintes et ses préférences.

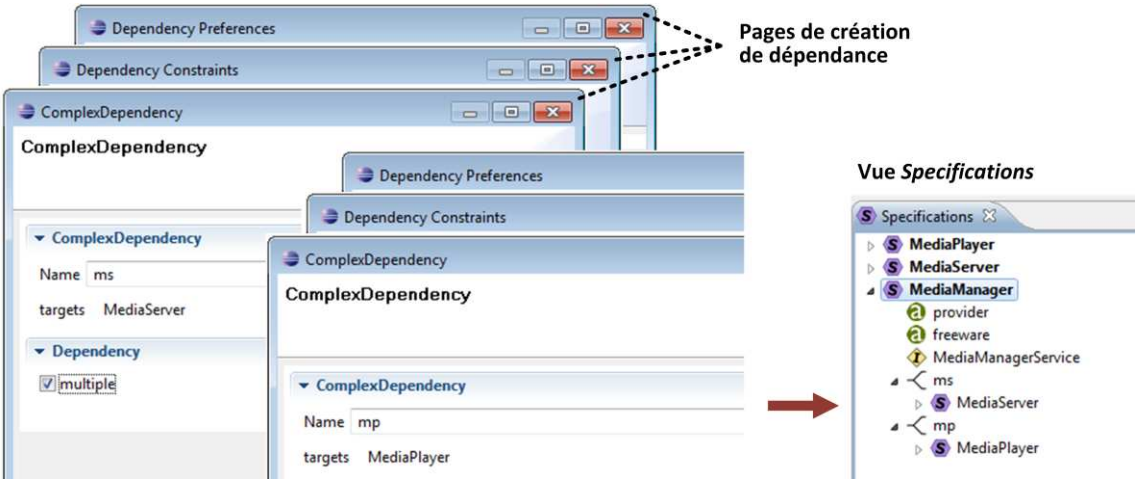
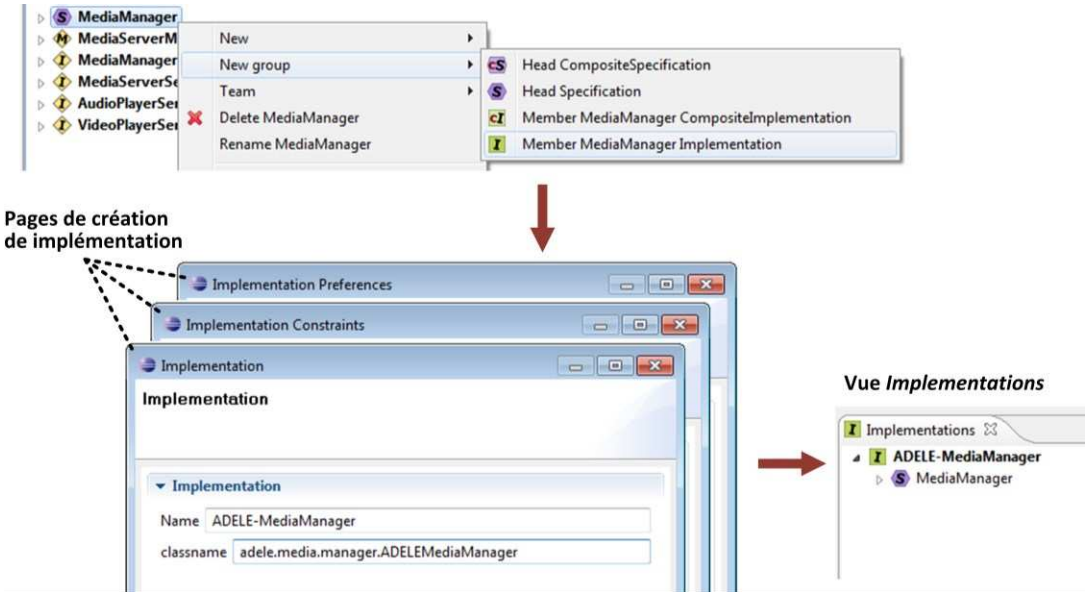


Figure 68. Création de la spécification *MediaManager*

La Figure 69 montre différentes étapes du processus de création de l'implémentation *ADELE-MediaManager* de la spécification *MediaManager* : (1) création de l'item *ADELE-MediaManager* de type *Implementation*, (2) ajout de caractéristiques additionnelles (ressources fournies, propriétés, ressources requises), (3) spécialisation de caractéristiques héritées de sa spécification, et (4) réalisation de l'implémentation. La conformité de l'implémentation avec sa spécification est garantie par construction. Le fichier de description XML généré à partir de ces informations est présenté dans la Figure 54.

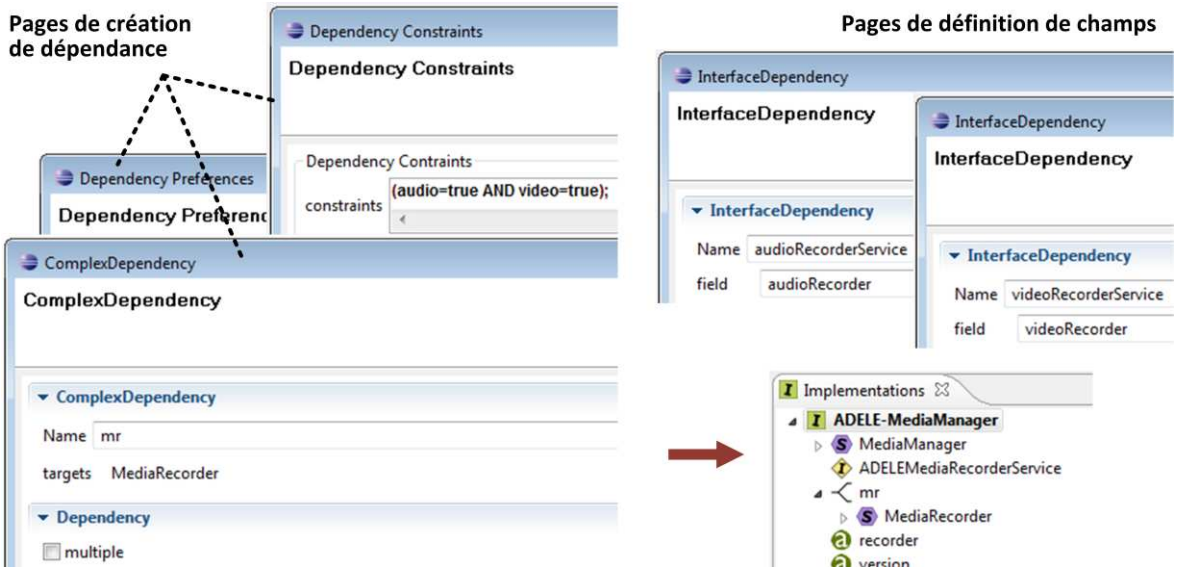
(1) Création de l'item ***ADELE-MediaManager*** à partir de la spécification *MediaManager* :

Une implémentation est créée à partir d'une spécification. Différentes pages de création sont présentées afin de définir ses propriétés (nom, propriétés prédéfinies, contraintes, préférences). La vue « Implementations » montre les implémentations présentes dans l'espace de travail.



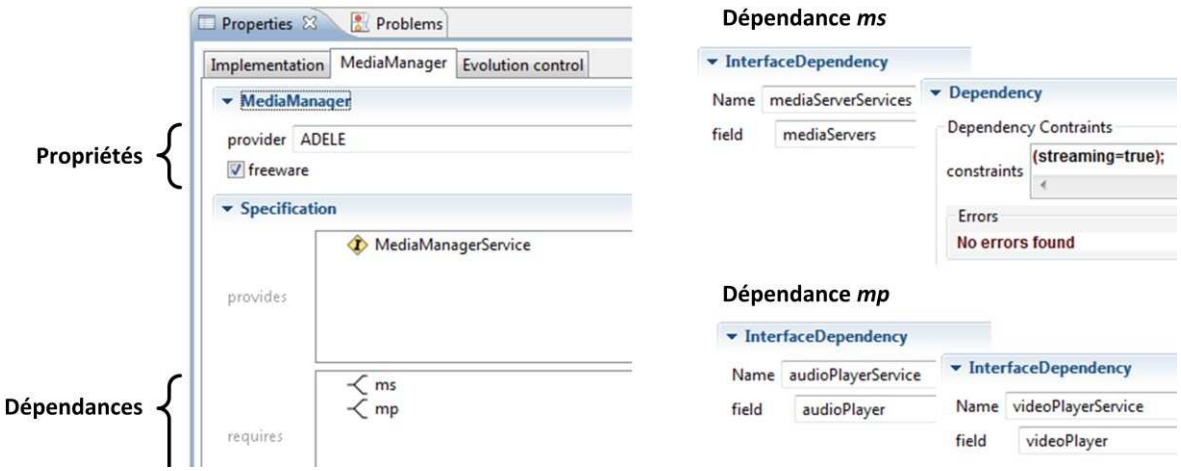
(2) Ajout de caractéristiques additionnelles à ***ADELE-MediaManager*** :

Diverses caractéristiques additionnelles peuvent être ajoutées à une implémentation en utilisant son menu : ressources fournies, dépendances simples et complexes, propriétés.
La figure ci-dessous montre l'ajout de la dépendance vers la spécification ***MediaRecorder*** *(avec la définition et de la contrainte (audio=true AND video=true), et la définition des champs associés : audioRecorder et videoRecorder).*

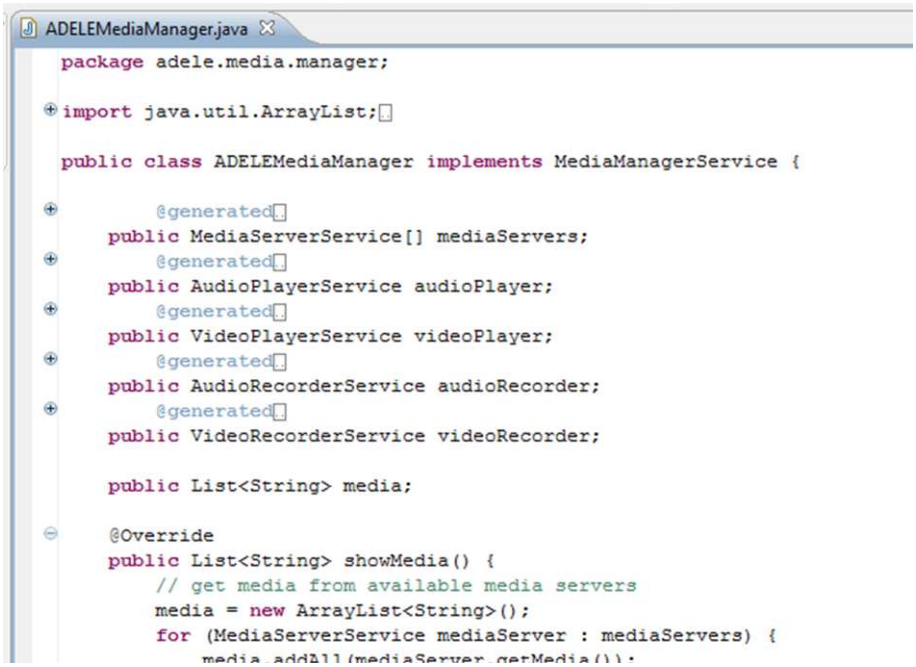


(3) Personnalisation des caractéristiques héritées de la spécification *MediaManager* :

Une implémentation peut personnaliser les caractéristiques héritées de sa spécification.
*La figure ci-dessous montre la personnalisation des propriétés **provider** et **freeware**, de la dépendance **ms** (**MediaServer**) avec la définition de la contrainte (streaming=true) et du champ mediaServers, et de la dépendance **mp** (**MediaPlayer**) avec la définition des champs audioPlayer et videoPlayer.*



(4) Réalisation de la classe d'implémentation ***ADELEMediaManager*** :

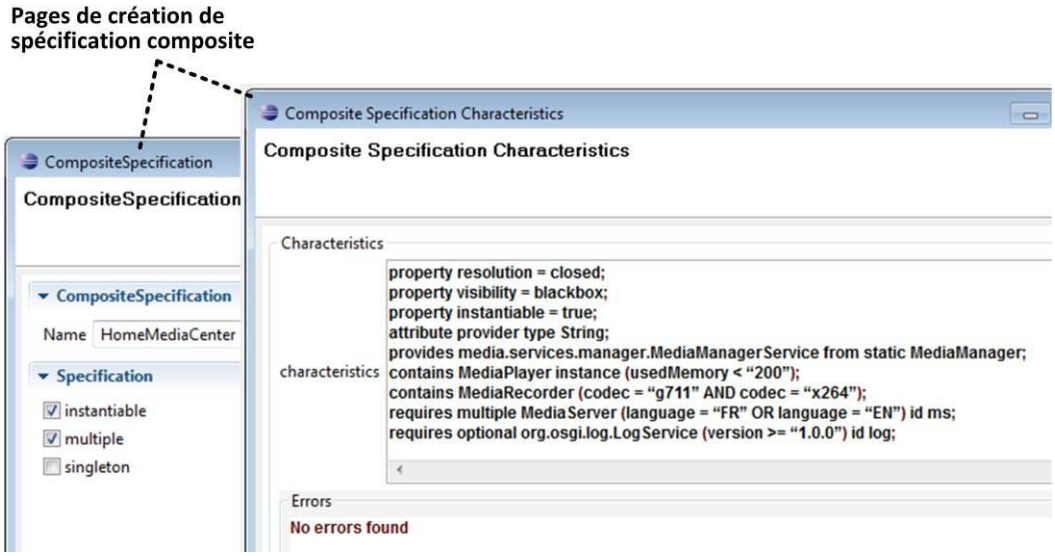


```
ADELEMediaManager.java
package adele.media.manager;

import java.util.ArrayList;

public class ADELEMediaManager implements MediaManagerService {

        @generated
    public MediaServerService[] mediaServers;
        @generated
    public AudioPlayerService audioPlayer;
        @generated
    public VideoPlayerService videoPlayer;
        @generated
    public AudioRecorderService audioRecorder;
        @generated
    public VideoRecorderService videoRecorder;

    public List<String> media;

    @Override
    public List<String> showMedia() {
        // get media from available media servers
        media = new ArrayList<String>();
        for (MediaServerService mediaServer : mediaServers) {
            media.addAll(mediaServer.getMedia());
```

Figure 69. Création de l'implémentation *ADELE-MediaManager*

La Figure 70 montre (1) la définition de l'application *HomeMediaCenter*, spécifiée en utilisant notre langage de composition, dont la spécification principale est la spécification *MediaManager* construite précédemment. Une description valide permet la création de la spécification composite correspondante avec toutes les caractéristiques définies (2).

(1) Définition de ***HomeMediaCenter*** :



(2) Les caractéristiques de la spécification composite ***HomeMediaCenter*** :

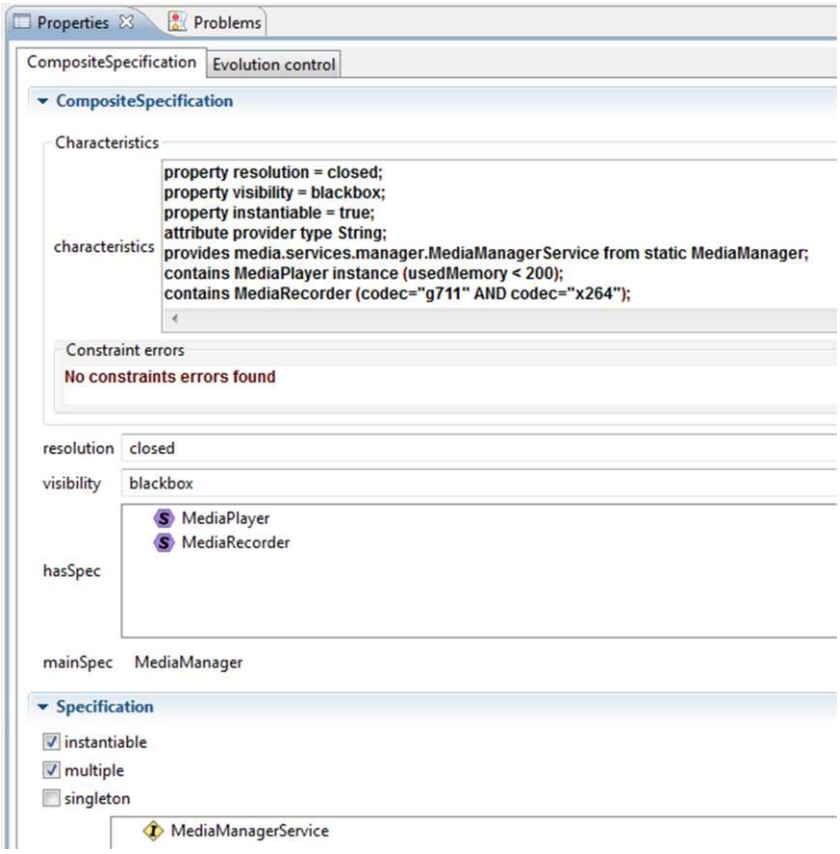


Figure 70. Spécification composite *HomeMediaCenter*

Sur la base de cette spécification composite, nous pouvons créer des implémentations composites. La Figure 71 illustre l'implémentation *ADELE-MediaCenter* de cette spécification. Elle hérite des caractéristiques de *HomeMediaCenter* et ajoute des caractéristiques propres (par exemple, la préférence *language="FR"* sur la dépendance vers la spécification *MediaServer*). Son implémentation principale est l'implémentation *ADELE-MediaManager* créée précédemment.

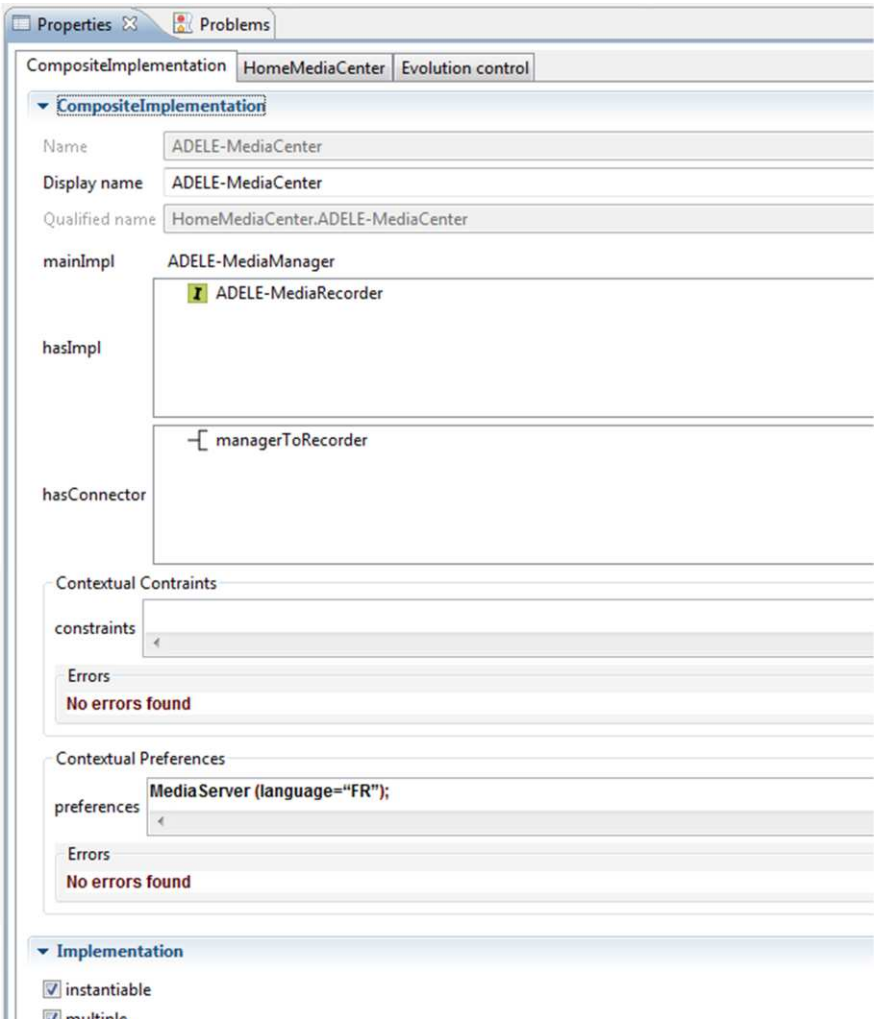


Figure 71. Implémentation composite ADELE-*MediaCenter*

Le contenu de cette implémentation peut être défini manuellement par le développeur (en sélectionnant des implémentations et en établissant des connecteurs entre elles), et/ou de façon automatisée par l'algorithme de composition. La conformité de la composition par rapport à sa définition est vérifiée et assurée. La description XML générée de cette implémentation est présentée dans la Figure 59. Sur la base de cette implémentation, différentes instances composites peuvent être créées et composées de façon manuelle et/ou automatisée en utilisant les instances disponibles dans l'espace de travail.

Le modèle de l'application est construit sur la base de la spécification composite *HomeMediaCenter* et de l'implémentation composite *ADELE-MediaCenter*. Un modèle d'application et ses composants sont packagés et mise à disposition dans un dépôt de composants disponible à l'exécution. Ainsi, le modèle d'application peut être déployé sur la plate-forme d'exécution et interprété afin d'exécuter l'application et de la gérer en conformité.

2 EXECUTION

Dans l'objectif d'exécuter des applications à services définies selon notre approche (en utilisant COMPASS-CADSE) nous fournissons un environnement d'exécution nommé COMPASS-RT. Cet environnement permet l'exécution des applications à partir de leur modèle d'application. De plus, il contrôle l'exécution des applications (i.e., leur composition dynamique et adaptative) en conformité à leur modèle d'application. De plus, il permet l'évolution des applications (par la modification de leur modèle d'application).

COMPASS-RT repose sur la plate-forme APAM qui fournit des fonctionnalités de base pour l'exécution d'applications à services. Dans cette section, nous présentons tout d'abord la plate-forme APAM, et puis nous détaillons les caractéristiques et les fonctionnalités de COMPASS-RT.

2.1 APAM

APAM[71] [87][88], acronyme de *Application Abstract Machine*, est notre plate-forme d'exécution d'applications à services. APAM étend les fonctionnalités des *frameworks* iPOJO et OSGi. et utilise le *framework* RoSe[72] [89] pour la découverte automatique de dispositifs et pour leur réification comme des services OSGi. L'objectif principal d'APAM est de permettre la construction dynamique des applications tout en contrôlant leur exécution.

L'ensemble du système APAM est constitué par la machine APAM qui fournit les fonctionnalités de base pour contrôler l'exécution d'applications ; et par un nombre de gestionnaires (*managers*) qui fournissent des fonctionnalités additionnelles pour contrôler des aspects spécialisés des applications, comme le déploiement ou l'adaptation (voir la Figure 72). Les gestionnaires associés à la machine APAM visent à augmenter (voire remplacer) la fonctionnalité de base qu'elle fournit.

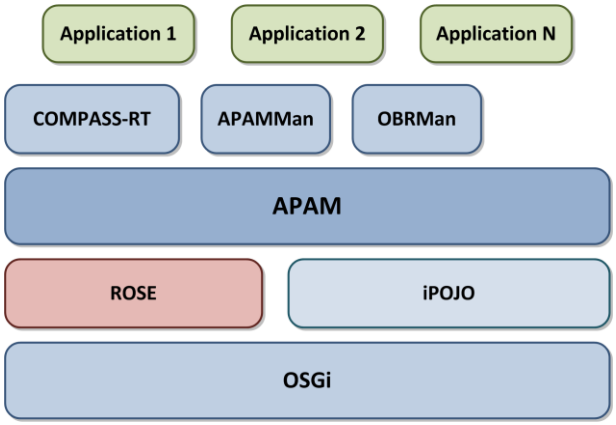


Figure 72. Architecture du système APAM

2.1.1 CONCEPTS

La machine APAM implémente un modèle de composants à services, proche du modèle de composants présenté dans le Chapitre 6, qui définit des composants à trois niveaux d'abstraction : **spécification**, **implémentation** et **instance**. Ce modèle introduit le concept de composant composite au niveau implémentation et instance, nommés **type composite** et **instance composite**.

[71] **https://github.com/AdeleResearchGroup/ApAM/wiki**
[72] **http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose**

Un composant APAM, primitif ou composite, peut fournir des interfaces et des messages, avoir des propriétés, et avoir des dépendances (avec des contraintes et des préférences définies par des filtres LDAP) vers des spécifications, des interfaces ou des messages. Les composants composites peuvent en addition avoir des propriétés de composition (des contraintes et des préférences contextuelles spécifiées avec des filtres LDAP) et de protection (la visibilité de leur contenu ainsi que la portée).

Un composant APAM (une implémentation) est implanté comme un POJO qui ne contient que la logique métier du composant : les primitives de l'approche à services dynamique (publication, découverte, sélection, liaison) sont masquées pour les développeurs et gérées par APAM. En effet, chacun de ces composants a un conteneur chargé de gérer des aspects non-fonctionnels ou récurrents. Ces aspects sont définis et gérés par des composants appelés *handlers*. APAM fournit des *handlers* de base pour la gestion du dynamisme, notamment pour la gestion de dépendances. De plus, les mécanismes d'APAM et d'iPOJO pouvant coexister dans le même environnement d'exécution, le conteneur d'un composant APAM peut être composé par des *handlers* APAM et iPOJO.

Chaque composant APAM (spécification, implémentation, instance et type composite) est décrit par un fichier XML. En utilisant le plugin ApamMavenPlugin[73] fourni par APAM, ces fichiers de description sont interprétés et validés (en termes de syntaxe et de cohérence), lors du packaging des composants, afin de générer les objets APAM correspondants ainsi que les fichiers de description XML pour les OBRs (*OSGi Bundle Repositories*). De plus, lors de cette phase, les POJOs sont manipulés (au niveau du *bytecode*) et instrumentés afin de permettre l'interception des méthodes et des accès aux champs des classes.

Les composants APAM sont ainsi des entités de première classe qui peuvent être décrits, packagés, déployés sur différents dépôts, installés et activés sur la plate-forme d'exécution. A l'exécution, ces entités sont toutes des objets APAM qui permettent la construction dynamique d'instances composites.

2.1.2 MECANISMES

APAM fournit un ensemble de mécanismes de base qui permettent l'exécution d'applications dynamiques. Nous présentons ces mécanismes par la suite.

2.1.2.1 INTROSPECTION ET MANIPULATION

L'introspection est un requis important dans le contexte des applications dynamiques. L'architecture de ces applications évoluant à l'exécution, il est primordial d'être capable de visualiser leur architecture d'exécution courante. Pour cela, APAM réifie/maintient un modèle nommé ASM (*Application State Model*) qui représente l'architecture d'exécution actuelle des applications (en termes de composants APAM et de connecteurs entre eux). Ce modèle est lié causalement à l'architecture du système en cours d'exécution (sur la plate-forme sous-jacente OSGi) : toute modification du modèle cause une modification de l'architecture en cours d'exécution, et toute modification de l'architecture en cours d'exécution cause une modification du modèle.

APAM fournit une API qui permet la visualisation et l'utilisation de l'ASM au travers d'opérations d'introspection et de manipulation de composants. Cette API est utilisée par les différents gestionnaires APAM et par l'administrateur de la plate-forme. Grâce à l'introspection, des problèmes dans l'architecture actuelle d'une application peuvent être détectés, voire corrigés, par l'administrateur ou par les différents gestionnaires.

[73] ApamMavenPlugin est un plugin Maven pour la compilation, le packaging et le déploiement de composants APAM

2.1.2.2 INJECTION ET INTERCEPTION

Les implémentations des composants (les classes d'implémentation) sont manipulées lors du packaging des composants (par l'ApamMavenPlugin). La manipulation encapsule tous les accès aux champs et toutes les méthodes d'une classe par des appels de méthodes délégués au conteneur du composant qui peut alors superviser les objets. Lors de la manipulation, certaines informations de la classe sont collectées et utilisées pour vérifier la cohérence de la classe d'implémentation et de la description du composant.

Les classes manipulées, utilisées à l'exécution à la place des classes d'origine, permettent ainsi d'intercepter les accès aux champs et aux méthodes. Par exemple, quand un accès à un champ est intercepté, APAM est appelée pour réaliser la résolution du composant requis et obtenir ainsi un objet approprié. Enfin, APAM injecte l'objet (si obtenu) dans le champ correspondant de la classe.

2.1.2.3 RESOLUTION ET CONTROLE

APAM fournit un résolveur de dépendances qui délègue la résolution aux différents gestionnaires concernés (des gestionnaires de type *DependencyManager*). APAM promeut l'utilisation opportuniste des composants. De ce fait, son gestionnaire de base pour la résolution de dépendances, appelé APAMMan, considère les composants contenus dans l'ASM (le modèle d'état) ainsi que ceux disponibles sur la plate-forme (utilisés par aucune application). APAMMan réalise la résolution en fonction des propriétés de composition (contraintes et préférences) et de protection (visibilité et scope) des composants.

En effet, APAM conserve la description des composants disponibles sur la plate-forme. Cela permet de guider/vérifier l'exécution des composants afin d'assurer leur conformité par rapport à leur définition. De plus, conserver la description des composants à l'exécution peut permettre l'évolution dynamique des composants par la modification de leur description.

2.1.2.4 NOTIFICATION

Les composants peuvent apparaître, disparaître ou être modifiés (affectation de leurs propriétés) dans l'environnement d'exécution. APAM fournit des mécanismes de notification : les gestionnaires intéressés sont notifiés des arrivées et des départs des composants (*DynamicManager*) ou des modifications de leurs propriétés (*PropertyManager*) afin d'agir en conséquence.

2.1.2.5 EXTENSION

APAM est une machine extensible : sa fonctionnalité de base peut être étendue (voire être remplacée) par des gestionnaires spécialisés dans la gestion de différents aspects des applications. Un tel gestionnaire peut être dirigé par un modèle (aspect spécifique) qui définit les propriétés à gérer et/ou à satisfaire lors de l'exécution d'une application. Une application peut ainsi être associée à différents modèles, chacun géré par un gestionnaire spécialisé.

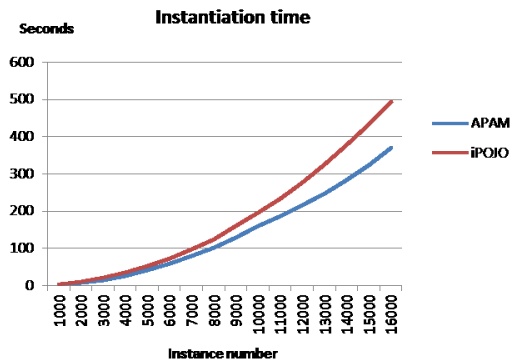
APAM délègue la gestion des différents aspects d'une application aux gestionnaires concernés. De plus, divers gestionnaires peuvent être impliqués dans la gestion d'un aspect particulier, par exemple dans la résolution de dépendances. APAM fournit un protocole de coordination de gestionnaires afin de contrôler la délégation : APAM arbitre les gestionnaires impliqués dans la gestion d'un aspect par des chemins (*paths*) qui établissent leur ordre d'invocation.

Par exemple, la résolution d'une dépendance implique plusieurs étapes: d'abord, APAM contacte tous les gestionnaires avec la demande de résolution afin de calculer le chemin de résolution (*resolutionPath*) et d'obtenir aussi la liste des contraintes et des préférences à satisfaire ; ensuite, APAM invoque les gestionnaires dans l'ordre établi par le *resolutionPath* délégant ainsi la résolution ; finalement, une fois la résolution réussie par un gestionnaire, APAM retourne au client l'instance ou les instances obtenues (au cas où la résolution échoue, un message d'erreur est retourné).

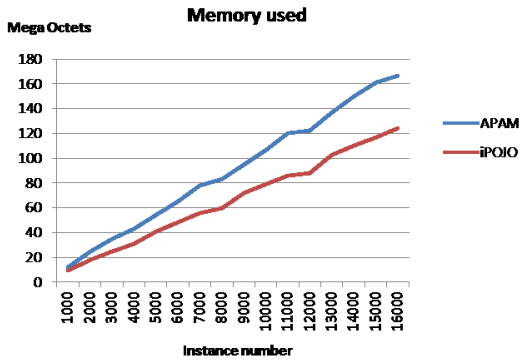
Actuellement, différents gestionnaires ont été implantés, étendant ainsi le fonctionnement de base d'APAM avec :

- **des mécanismes de déploiement dynamique** via le gestionnaire OBRMan (*DependencyManager*) qui supporte la résolution de composants requis dans différents dépôts de composants, ainsi que le déploiement dynamique des composants (des *bundles* contenant les composants) sur la plate-forme.
- **des mécanismes d'adaptation dynamique** via le gestionnaire DynaMan (*DynamicManager*) qui maintient à jour les dépendances des composants en les adaptant suite à l'apparition, à la disparition des composants.

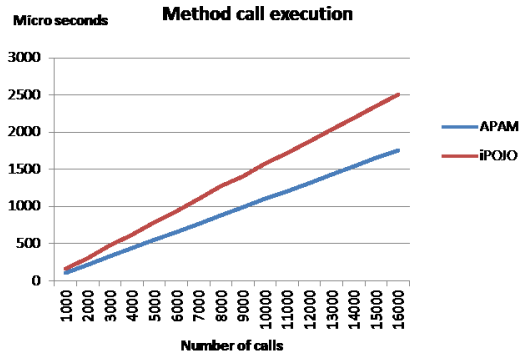
Les expérimentations réalisées jusqu'à présent montrent que la machine APAM est efficace (voir la Figure 73). Par rapport à la plate-forme iPOJO, APAM est 10% plus rapide en ce qui concerne le temps d'instanciation (en effet, APAM n'utilise pas le registre OSGi : l'ASM est le registre d'APAM). Par rapport à la consommation de la mémoire en fonction du nombre d'instances créées, APAM a un surcoût d'environ 10%, dû essentiellement à l'état réifié. Enfin, par rapport au temps d'appel, APAM est aussi plus performante qu'iPOJO.



(a) Temps d'instanciation



(b) Consommation de la mémoire



(c) Temps d'invocation

Figure 73. Résultats des expérimentations APAM vs iPOJO

Les travaux actuels autour d'APAM visent à rendre la plate-forme plus stable et robuste afin de pouvoir être utilisée comme une plate-forme de recherche ouverte.

Grâce aux mécanismes de base fournis par APAM et à ses capacités d'extension, nous définissons COMPASS-RT comme un gestionnaire APAM capable de contrôler l'exécution des applications construites selon notre approche en fonction de leur modèle d'application.

2.2 COMPASS-RT

COMPASS-RT est un gestionnaire APAM. Son objectif est d'exécuter sur APAM les applications construites selon notre approche (à l'aide de notre environnement COMPASS-CADSE) en conformité à leur modèle d'application.

Les concepts, les modèles et certains mécanismes utilisés par COMPASS-RT sont les mêmes utilisés par COMPASS-CADSE dans les phases de conception et de développement[74]. En effet, un des objectifs de notre approche, et donc de notre système COMPASS, est d'étendre l'utilisation des modèles et des mécanismes de conception et de développement d'applications à la phase d'exécution, et d'estomper ainsi la frontière entre les phases de développement et d'exécution. L'environnement d'exécution doit ainsi pouvoir gérer des concepts de conception et de développement.

COMPASS-RT peut être vu comme un type spécial de CADSE (indépendant d'Eclipse) dédié à l'exécution d'applications. Il adopte les mêmes concepts, modèles et mécanismes utilisés pour la conception et le développement d'applications, permettant ainsi d'effectuer certaines activités de conception et de développement lors de l'exécution d'une application.

Comme tout CADSE, COMPASS-RT administre un espace de travail qui contient des éléments propres au domaine : spécifications, implémentations, instances, modèles d'application. De tels éléments peuvent être ajoutés à l'espace de travail grâce aux mécanismes d'exportation et d'importation d'éléments fournis par CADSE : les éléments exportés de l'espace de travail d'un COMPASS-CADSE peuvent ainsi être importés dans l'espace de travail de COMPASS-RT.

COMPASS-RT gère la création, la destruction, la modification des éléments de son espace de travail. Il l'utilise principalement pour réaliser la résolution de composants lors de l'exécution d'une application.

Afin d'exécuter une application, COMPASS-RT fournit des mécanismes qui permettent :

- le démarrage de l'application à partir de son modèle d'application,
- la gestion de l'exécution de l'application, en cohérence et conformité avec son modèle d'application, par :
  - la résolution des services requis par l'application, et
  - l'adaptation de l'application suite à des variations dans son environnement d'exécution, et
- l'évolution de l'application via la modification de son modèle d'application.

2.2.1 DEMARRAGE D'UNE APPLICATION

COMPASS-RT **démarre l'exécution d'une application à partir de son modèle d'application**. COMPASS-RT peut obtenir un modèle d'application à exécuter (par nom) depuis son espace de travail. COMPASS-RT permet aussi la création d'un modèle d'application à partir d'une spécification et complémentairement d'une implémentation et d'une instance composites contenues dans l'espace de travail.

Les méthodes fournies par COMPASS-RT pour la création de modèles d'application et pour le démarrage de leur exécution sont les suivantes :

[74] Toutefois, une partie du code a été adapté afin de le rendre indépendant d'Eclipse et de pouvoir l'exécuter sous une plate-forme OSGi.

```
/* Methods for creating an application model :
 * their components (Specification, CompositeImplementation and CompositeInstance)
 * are retrieved (by their name) from the COMPASS-RT workspace */
public ApplicationModel createApplicationModel(String applicationModelName, String specificationName);
public ApplicationModel createApplicationModel(String applicationModelName, String specificationName,
                        String compositeImplementationName);
public ApplicationModel createApplicationModel(String applicationModelName, String specificationName,
                        String compositeImplementationName, String compositeInstanceName);

/* Methods for starting an application by its application model */
public void startApplication(String applicationModelName);  // the application model is retrieved
                                                            // from the COMPASS-RT workspace
public void startApplication(ApplicationModel applicationModel);
```

Ainsi, à partir d'un modèle d'application, COMPASS-RT effectue les actions nécessaires pour le démarrage de l'application. La stratégie traditionnelle d'exécution d'applications réunit (déploie/active/lie) tous les composants requis d'une application afin de pouvoir la démarrer et l'utiliser. Dans iPOJO par exemple, le démarrage (et la validité) d'une application dépend de la disponibilité de tous les composants requis : une application incomplète n'est pas utilisable. Pourtant, dans un contexte dynamique, entre l'instant t_0 auquel une application est démarrée et l'instant t_1 auquel elle a besoin d'un service S, des variations dans le contexte d'exécution peuvent avoir lieu : S peut devenir indisponible ou inadéquat pour l'application.

De ce fait, COMPASS-RT adopte par défaut une stratégie paresseuse (*lazy*) pour l'exécution d'applications : la résolution, le déploiement, l'activation et la liaison d'un service requis par une application sont effectués lors de la demande du service : une application est donc composée/déployée graduellement suite à la demande des services requis. En conséquence, seuls les services vraiment utilisés par une application seront résolus, déployés, activés et/ou liés à l'application. Cette stratégie permet, d'une part, d'affronter la nature dynamique des services, et d'autre part, d'économiser les ressources de la plate-forme d'exécution, comme la consommation de la mémoire.

COMPASS-RT supporte également une stratégie immédiate (*eager*) pour l'exécution d'applications : la résolution, le déploiement, l'activation et/ou la liaison d'un service requis par une application peuvent être réalisés de façon immédiate lors de son démarrage. Cette stratégie d'exécution permet de réduire le temps d'invocation des services, notamment lors de leur première invocation. Néanmoins, sous cette stratégie, un service est déployé, activé, lié et géré même s'il n'est jamais utilisé par l'application, consommant donc plus de ressources de la plate-forme d'exécution.

Le processus de démarrage d'une application effectué par COMPASS-RT consiste ainsi à créer, si nécessaire[75], les composants APAM correspondants aux composants indispensables[76] et aux composants immédiats définies dans son modèle d'application.

Pour les composants indispensables et immédiats contenus dans le modèle d'application, **COMPASS-RT réalise directement la création des composants APAM** correspondants par le déploiement et l'activation de leur *bundle*[77] sur la plate-forme d'exécution sous-jacente. Par contre, si un composant indispensable (par exemple l'instance principale de l'application) ou immédiat (par exemple l'instance d'une spécification dont la propriété de résolution est définie comme immédiate) n'est pas contenu dans le modèle d'application, COMPASS-RT cherche à l'obtenir par résolution.

[75] Un composant APAM est créé seulement s'il n'existe pas encore (s'il n'est pas disponible) sur la plate-forme d'exécution. La création d'un composant APAM résulte ainsi dans un composant disponible à l'exécution.
[76] Les composants indispensables pour le démarrage d'une application sont la spécification principale (*mainSpec*), l'implémentation composite avec l'implémentation principale (*mainImpl*), et l'instance composite avec l'instance principale (*mainInst*).
[77] Les composants possèdent un attribut (*bundleURL*) avec la localisation du *bundle* généré lors de leur packaging.

Si la résolution d'un composant (d'une spécification ou d'une implémentation) est spécifiée comme fermée, **COMPASS-RT effectue directement la résolution** en utilisant son espace de travail comme espace de résolution. Lors de ce processus, COMPASS-RT peut sélectionner un composant (qui satisfait les contraints et les préférences de résolution) depuis son espace de travail, ou il peut le créer (à l'aide d'une fabrique). Pour un composant sélectionné/créé, COMPASS-RT crée le composant APAM correspondant (ce dernier est retourné comme résultat de la résolution). Le composant est ensuite ajouté au modèle d'état de l'application et ses correspondances (avec un composant de l'espace de travail et/ou avec un composant APAM) sont établies.

Si la résolution d'un composant est spécifiée comme opportuniste ou collaborative, **COMPASS-RT délègue la résolution au résolveur APAM** afin d'être réalisée par un autre gestionnaire (par APAMMan ou par OBRMan par exemple). La résolution a pour résultat un composant APAM à partir duquel le composant correspondant dans le modèle d'état de l'application est créé.

Le modèle d'état d'une application à démarrer est créé à partir de son modèle d'application. COMPASS-RT réalise donc le démarrage d'une application en trois étapes principales :

1. **La création des spécifications APAM :**

   i) Si la spécification du modèle d'application (i.e., la définition en intention) est primitive, COMPASS-RT crée directement la spécification APAM correspondante.

   ii) Sinon (i.e., si la spécification du modèle d'application est composite), COMPASS-RT crée directement les spécifications APAM correspondantes à la spécification principale (*mainSpec*) et aux spécifications contenues dans la spécification composite dont la propriété d'activation est définie comme immédiate (*eager*).

2. **La création des implémentations APAM :**

   COMPASS-RT crée, directement ou par résolution, le type composite APAM (*CompositeType*) correspondant à l'implémentation composite de l'application :

   i) Si le modèle d'application contient l'implémentation composite de l'application, COMPASS-RT crée directement le type composite APAM correspondant. Pour cela, il crée directement l'implémentation APAM correspondant à l'implémentation principale (*mainImpl*) contenue dans l'implémentation composite. De plus, il crée directement les implémentations définies comme immédiates et les ajoute/lie au type composite APAM créé.

   ii) Sinon, COMPASS-RT déclenche la résolution de la spécification du modèle d'application afin d'obtenir un type composite APAM avec son implémentation principale APAM (obtenue aussi par résolution). COMPASS-RT réalise directement cette résolution si la propriété de résolution de la spécification est définie comme fermée, sinon il délègue la résolution au résolveur APAM.

   De plus, si la spécification du modèle d'application est composite, COMPASS-RT déclenche la résolution des spécifications définie comme immédiate contenues dans la spécification composite et ajoute/lie les implémentations APAM issues de la résolution au type composite APAM.

3. **La création des instances APAM :**

   COMPASS-RT crée, directement ou par résolution, l'instance composite APAM (*Composite*) correspondant à l'instance composite de l'application :

   i) Si le modèle d'application contient l'instance composite de l'application, COMPASS-RT crée directement l'instance composite APAM correspondante. Pour ce faire, il crée directement l'instance APAM correspondante à l'instance principale (*mainInst*) contenue dans l'instance composite. COMPASS-RT crée aussi directement les

instances APAM correspondantes aux instances contenues dans l'instance composite dont la propriété d'activation est définie comme immédiate, et les ajoute/lie à l'instance composite APAM créée.

ii) Sinon, COMPASS-RT déclenche la résolution de l'implémentation composite de l'application afin d'obtenir une instance composite APAM avec son instance principale APAM (obtenue aussi par résolution). COMPASS-RT effectue directement la résolution si la propriété de résolution est définie comme contrainte, autrement il délègue la résolution au résolveur APAM.

En outre, COMPASS-RT déclenche la résolution des implémentations contenues dans l'implémentation composite dont la propriété d'activation est définie comme immédiate, et ajoute/lie les instances APAM issues de la résolution à l'instance composite APAM.

Le modèle d'état d'exécution d'une application démarrée contient donc les composants indispensables et immédiats mappés à leur composant APAM correspondant.

Une fois une application démarrée, son exécution est pilotée et gérée par APAM à l'aide de l'ensemble des gestionnaires disponibles, COMPASS-RT y compris.

2.2.2 GESTION D'UNE APPLICATION

COMPASS-RT, étant un gestionnaire de type *DependencyManager*, **contrôle et réalise la résolution des services** requis par une application lors de leur demande.

COMPASS-RT **contrôle la résolution d'un service** via le chemin d'ordre de délégation suivi par APAM : le *resolutionPath* ; et via les propriétés, les contraintes et les préférences de résolution à satisfaire. En effet, afin d'établir le *resolutionPath* lors de la résolution d'un service, APAM interroge les gestionnaires de type *DependencyManager* : chaque gestionnaire signale sa position dans le *resolutionPath* ainsi que ses contraintes et ses préférences pour la résolution en question. COMPASS-RT se positionne en première position dans le *resolutionPath* si la propriété de résolution du service à résoudre (propriété gérée par COMPASS-RT) est spécifiée comme fermée. Sinon, il se place derrière APAMMan et OBRMan (si disponibles). APAM demande donc la résolution du service aux gestionnaires dans l'ordre établi par le *resolutionPath*.

COMPASS-RT **réalise la résolution d'un service** suite à la demande transmise par APAM. Il résout un service d'abord dans le modèle d'application correspondant car celui peut contenir des services déjà sélectionnés (résolus) mais qui ne sont pas encore activés sur la plate-forme d'exécution ; ensuite (si la résolution dans le modèle d'application n'a pas réussi) il résout dans l'espace de travail.

COMPASS-RT peut donc sélectionner un composant contenu dans le modèle d'application ou dans l'espace de travail, ou il peut le créer à l'aide d'une fabrique. Pour un composant sélectionné/créé, COMPASS-RT crée le composant APAM correspondant (résultat de la résolution), l'ajoute au modèle d'état de l'application et établit ses correspondances avec les composants correspondants de l'espace de travail et APAM. La résolution des services requis par une application gérée par COMPASS-RT est réalisée ainsi en conformité à son modèle d'application.

De plus, COMPASS-RT **vérifie et garantit la conformité de l'exécution d'une application** vis-à-vis de son modèle d'application.

En effet, en plus d'être un gestionnaire de type *DependencyManager*, COMPASS-RT est de type *PropertyManager* et *DynamicManager*. Il est donc notifié par APAM des changements (ajout, suppression, modification) des propriétés des composants APAM, ainsi que de leur retrait du modèle d'état.

Ainsi, suite à la notification du changement d'une propriété d'un composant participant à une application gérée par COMPASS-RT, il vérifie si le composant en question satisfait toujours les propriétés, les contraintes et les préférences de l'application spécifiées dans son modèle d'application.

Lorsqu'un composant ne satisfait plus les propriétés de l'application à laquelle il participe, ou lorsqu'un composant participant à une application est retiré de la plate-forme d'exécution, COMPASS-RT retire le composant (avec ses liaisons) du modèle d'état actuel de l'application et si nécessaire du modèle d'état géré par APAM.

Toutefois, le gestionnaire DynaMan propose différentes stratégies pour l'adaptation dynamique d'une application suite à l'apparition, à la disparition ou à l'échec de la résolution d'un composant. Par exemple, suite à l'échec de la résolution d'un composant (*fail*) ou à la disparition d'un composant participant à une application, l'application peut être mise en attente (*wait*) jusqu'à l'apparition d'un fournisseur approprié, ou une exception spécifique peut être levée.

2.2.3 EVOLUTION D'UNE APPLICATION

COMPASS-RT **supporte l'évolution dynamique d'une application** au travers de modifications dans son modèle d'application (*top-down*). Le but de l'évolution dynamique est de corriger ou de perfectionner une application pendant son exécution.

L'évolution des applications s'est déplacée ces dernières années du code source aux modèles des architectures logicielles (*architecture-centric evolution*). En effet, le code source ne peut pas fournir une vue complète d'une application ni refléter les décisions originales de conception. Les modèles des architectures logicielles fournissent ainsi une base plus gérable et plus efficace pour l'évolution des applications : ils permettent de changer des éléments logiciels d'une application, tels que des composants et des connecteurs entre composants.

Le modèle d'application proposé dans notre approche possède des informations sémantiques et structurelles d'une application. Ainsi, afin de faire évoluer une telle application, des modifications apportées dans son modèle d'application à différents niveaux d'abstraction (spécification, implémentation et instance).

L'évolution dynamique d'une application est cependant un processus complexe sujet à erreurs. COMPASS-RT implémente le processus suivant pour l'évolution d'une application :

- la vérification de la validité des modifications à effectuer (détection d'erreurs de syntaxe par exemple),
- la vérification de la conformité et la cohérence entre les différentes architectures du modèle d'application (i.e., la détection d'erreurs/incohérences de sémantique et de structure) et la propagation de modifications (i.e., l'adaptation des architectures), et
- la projection des modifications du modèle d'application sur le modèle d'état (i.e., l'adaptation des architectures d'exécution de l'application).

COMPASS-RT permet de faire évoluer une application grace aux modifications suivantes effectuées au niveau spécification, implémentation et/ou instance du modèle d'application :

- **modifications sémantiques** : ajout, modification et suppression de contraintes et de préférences contextuelles.
- **modifications structurelles** : ajout et retrait de composants et de liaisons entre composants.

De telles modifications, conçues par l'architecte ou par l'administrateur de l'application, sont vérifiées et validées par COMPASS-RT afin de les réaliser effectivement dans le modèle d'application. Ces modifications, dites externes, sont donc responsabilité de l'architecte et/ou de l'administrateur de l'application.

Toutefois, même si elles sont valides, ces modifications peuvent entraîner l'érosion architecturale[78] du modèle d'application. En effet, une modification réalisée à un niveau d'abstraction peut affecter les autres niveaux et doit donc être propagée (par exemple, une modification effectuée au niveau spécification entraîne en général des modifications aux niveaux implémentation et instance).

COMPASS-RT réalise la propagation de modifications, guidée par les relations de conformité et de cohérence entre les architectures, afin d'éviter l'érosion architecturale du modèle d'application et de garantir sa validité.

Une fois un modèle d'application validé, COMPASS-RT projette les modifications nécessaires sur le modèle d'état de l'application afin de le rendre conforme au modèle d'application. Toute modification dans le modèle d'état géré par COMPASS-RT entraîne une modification dans le modèle d'état géré par APAM (car liés causalement) provoquant donc l'adaptation de l'architecture de l'application en cours d'exécution.

Grâce à l'information contenue dans un modèle d'application et aux fonctionnalités fournies par COMPASS-RT, des activités de conception et de développement peuvent être réalisées à l'exécution, entraînant donc l'évolution dynamique de l'application.

3 SYNTHESE

Nous avons présenté la mise en œuvre de notre approche à travers de notre système COMPASS. Nous avons détaillé les environnements qui le composent : COMPASS-CADSE qui supporte la conception et le développement d'applications, et COMPASS-RT qui supporte leur exécution.

Notre système, en suivant les principes du génie logiciel, a comme caractéristiques principales la modularité, l'extensibilité et la séparation des préoccupations.

[78] L'érosion architecturale est causée par des modifications réalisées dans une architecture qui violent les propriétés de l'architecture de haut niveau associée.

CHAPITRE 8
CONCLUSION ET PERSPECTIVES

1 SYNTHESE

L'essor d'Internet et l'évolution des dispositifs communicants pousse de nombreuses industries à proposer des applications intégrant le monde informatique et le monde réel. Cependant, ce nouveau contexte complexifie considérablement le développement des applications. En effet, les applications doivent s'exécuter dans des contextes hétérogènes, distribués et ouverts qui sont en constante évolution. Dans de tels contextes, la disponibilité des services et des dispositifs, les préférences et la localisation des utilisateurs peuvent varier à tout moment pendant l'exécution des applications.

La variabilité des contextes dans lesquels s'exécutent les applications rend difficile, voire impossible, d'avoir une connaissance complète, lors de la conception et du développement des applications, des conditions précises dans lesquelles les applications seront utilisées : les services et les dispositifs qui seront disponibles à l'exécution ou ceux qui seraient mieux adaptés à un moment donné. Même si les décisions prises à la conception d'une application sont exactes au début de son exécution, l'évolution du contexte peut requérir son adaptation dynamique afin de lui permettre de continuer à fonctionner ou de tirer parti des services et des dispositifs apparus en cours d'exécution. **Le dynamisme est ainsi un besoin crucial pour la construction des applications qui requièrent de s'adapter à leur contexte d'exécution.**

L'approche à services offre des concepts particulièrement intéressants pour la construction d'applications dynamiques, et de nombreux travaux s'intéressent à la construction d'applications par composition de services. Toutefois, la plupart des approches entrainent de nombreuses contraintes et ne répondent pas complètement aux besoins et aux défis des applications dynamiques. En effet, le dynamisme soulève de nombreux besoins et défis pour la construction d'applications à services comme par exemple :

- la définition d'une application par l'ensemble des propriétés et des contraintes à satisfaire lors de sa composition,
- la composition incrémentale et dynamique de l'application,
- la vérification de la cohérence et de la conformité de la composition vis-à-vis de sa définition,
- l'adaptation dynamique de la composition face à la variabilité du contexte d'exécution,
- l'évolution de la définition d'une application.

Afin de répondre à ces besoins et défis, nous avons proposé une approche qui combine des concepts de l'approche à composants, de l'approche à services, des architectures logicielles et de l'ingénierie dirigée par les modèles. Nous proposons ainsi un modèle à composants à services qui

aborde des aspects de conception, de développement et d'exécution d'applications. Ce modèle a plusieurs propriétés importantes telles que :

- l'utilisation de concepts de l'approche à service dans le modèle à composants,
- la notion de composant à service à trois niveaux d'abstraction : spécification, implémentation et instance,
- la notion de dépendance de service au niveau des spécifications,
- la composition d'applications à trois niveaux d'abstraction (au travers de composants composites : spécifications composites, implémentations composites et instances composites).

Les concepts du modèle sont présents et ont une sémantique constante tout au long du cycle de vie des applications, de la phase de conception à la phase d'exécution. Cela permet de transférer à la phase d'exécution l'information (i.e., les métadonnées) produite dans les phases de conception et de développement des composants : l'architecture prescriptive d'une application est connue à l'exécution, permettant de garantir l'état d'exécution conforme de l'application (son architecture descriptive). Cela permet aussi, voire surtout, de suivre la même démarche d'abstraction, de concrétisation, de catégorisation et de composition de services avant et durant la phase d'exécution des applications.

En utilisant le modèle proposé, une application est construite en trois phases :

1. **La phase de conception** où l'application est définie en intention en termes de spécifications de service ainsi que de propriétés, contraintes et préférences qui établissent les règles de composition de l'application.
2. **La phase de développement** où l'application est définie partiellement ou complètement en extension en termes d'implémentations et d'instances de services qui satisfont la définition en intention. La définition en intention et en extension d'une application constituent le modèle d'application (i.e., son architecture prescriptive).
3. **La phase d'exécution** où l'application est déployée et exécutée sur l'environnement d'exécution conformément à son modèle d'application. L'application est composée graduellement (lors de la demande de services) en termes de spécifications, d'implémentations et d'instances[79]. De plus, l'application peut être adaptée suite à la variabilité du contexte d'exécution, et peut évoluer afin de corriger, améliorer, étendre ou réduire ses fonctionnalités.

Avec notre approche, une application peut être définie de manière suffisamment flexible afin de pouvoir l'adapter (en conformité avec sa définition) dynamiquement à son contexte d'exécution et aux variations de celui-ci, mais aussi de manière suffisamment précise afin de pouvoir contrôler son exécution de façon stricte.

Nous estimons ainsi que notre approche contribue au domaine des applications dynamiques en apportant les propriétés suivantes :

- **Une application peut être définie en intention, en extension ou les deux.** Le concept de service composite défini à différents niveaux d'abstraction (spécification, implémentation et instance) permet la définition d'applications en intention, en extension ou les deux. Certaines parties de la définition d'une application peuvent rester abstraites jusqu'à son exécution, donnant plus de flexibilité à l'application face à des environnements dynamiques et peu prévisibles. Notre approche supporte ainsi la définition d'applications dynamiques, statiques et mixtes (semi-statiques et semi-dynamiques).

[79] Les concepts de spécification, implémentation et instance sont des entités de premier ordre présentes à l'exécution.

- **La définition d'une application est validée statiquement.** Les propriétés, les contraintes et les préférences contextuelles d'une application portent sur les propriétés de ses services abstraits. Le concept de service abstrait étant formellement défini, la validité des propriétés, des contraintes et des préférences contextuelles peut être vérifiée statiquement afin d'assurer la validité de la définition d'une application.
- **La composition d'une application peut être réalisée avant ou pendant la phase d'exécution.** Une application peut être composée graduellement avant son exécution : composition statique ; et/ou pendant son exécution : composition dynamique.
- **La conformité de la composition d'une application est garantie.** La composition statique d'une application est guidée par sa définition en intention afin de garantir la conformité de son développement et de sa configuration. La composition dynamique d'une application est guidée par le modèle de l'application (i.e., par les définitions en intention et en extension) afin de garantir la conformité de son exécution.
- **La protection d'une application est garantie.** Une application peut spécifier les règles de partage et de visibilité de ses services (implémentations et instances). Les composites représentant l'architecture d'une application en termes d'implémentations (implémentation composite), configurations (configuration composite) et instances (instance composite), sont des mécanismes d'encapsulation qui permettent la protection des services qu'ils contiennent.
- **Une application peut évoluer dynamiquement.** La définition d'une application peut évoluer dynamiquement.

Grâce à ces propriétés, notre approche rend plus simple et plus sûre la construction d'applications, de leur conception à leur exécution. Toutefois, nous considérons que notre approche aborde partiellement les défis des applications dynamiques.

2 PERSPECTIVES

Notre travail propose une approche pour faciliter la construction d'applications à services dynamiques. Bien sûr, diverses améliorations peuvent être réalisées, mais surtout, nous considérons que notre approche ouvre la voie à diverses perspectives.

2.1 OUTILS D'EXECUTION INTEGRES AUX ENVIRONNEMENTS DE DEVELOPPEMENT

L'observation de l'état d'exécution des systèmes logiciels joue un rôle important lors du processus de développement et d'administration des systèmes. En effet, elle facilite l'analyse et l'identification des problèmes dans le fonctionnement des systèmes.

Pour cela, nous considérons important de fournir des environnements intégrés capables de concevoir, développer, déployer et surveiller des applications dynamiques. Divers aspects d'exécution, liés notamment au dynamisme, doivent être visibles dans le processus de développement des applications afin de permettre aux architectes et développeurs de mieux comprendre leur exécution et de rendre leur évolution plus efficace.

De même, des environnements de test et de simulation qui fournissent des informations aux architectes et aux développeurs (erreurs, impacts, conflits) peuvent être intégrés aux environnements de développement afin de rendre plus sûr le développement et l'évolution des applications.

2.2 CALCUL DE « LA MEILLEURE COMPOSITION POSSIBLE »

Notre algorithme de composition d'applications, basé sur la notion de groupes d'équivalence et de résolution, automatise la sélection des composants participant à la composition d'une application tout en garantissant que les composants sélectionnés satisfont les contraintes de composition de l'application. Le processus de composition permet donc de dériver une composition valide (conforme) d'une application. Toutefois, un nombre potentiellement infini de compositions valides pourraient être dérivées.

Actuellement, notre approche permet de garantir des contraintes et des préférences, mais ne permet pas de définir un concept de composite « meilleur » ou « optimal ». Nous considérons qu'être capable d'exprimer et de choisir « la meilleure composition possible » parmi l'ensemble des compositions valides serait une amélioration importante.

Nous estimons que la connaissance des composants disponibles dans les dépôts (OBR), notamment de leurs propriétés et de leurs dépendances, peut permettre de calculer/prévoir la composition la plus adaptée au contexte d'exécution courant d'une application et la moins encline à échouer : les dépendances des composants sélectionnés participant à la composition pourraient être résolues par des composants disponibles dans les dépôts qui devront donc être déployés. Toutefois, d'autres contraintes comme l'échec du déploiement des composants, l'indisponibilité et la variabilité des dépôts devront être prises en compte.

2.3 INTEGRATION D'AUTRES PREOCCUPATIONS NON-FONCTIONNELLES

Nous considérons qu'un système doit être représenté par plusieurs modèles, chacun représentant une préoccupation du système à un certain niveau d'abstraction, i.e., un point de vue du système dans une phase de son cycle de vie.

Nous avons abordé la construction d'applications à services du point de vue métier en proposant un modèle qui couvre le cycle de vie des applications en considérant le dynamisme des services. D'autres préoccupations, comme la persistance, la sécurité ou la qualité de service, n'ont pas été abordées. Toutefois, nous estimons que notre métamodèle et nos environnements, étant extensibles, permettent d'ajouter les concepts propres à des propriétés non-fonctionnelles ainsi que les fonctionnalités pour les gérer.

En suivant le principe de séparation des préoccupations, les concepts propres à une propriété non-fonctionnelle peuvent être définis séparément par un métamodèle spécifique, et liés aux concepts de notre métamodèle central de composants à services.

Une application pourra donc être spécifiée, développée, exécutée et gérée en utilisant plusieurs modèles représentant des préoccupations différentes. Au niveau spécification par exemple, des modèles correspondants à des préoccupations différentes (persistance, sécurité, qualité de service,…) seront définis et liés au modèle métier de l'application. A l'exécution, l'application sera gérée par un ensemble de mécanismes spécifiques (gestionnaires et *handlers*), chacun gérant le modèle à préoccupation correspondant.

Un des avantages majeurs de la séparation des préoccupations en plusieurs métamodèles liés à notre métamodèle central est que les modèles peuvent être maintenus plus facilement. Toutefois, l'ajout des diverses propriétés non-fonctionnelles peut poser des problèmes d'interopérabilité qui devront être considérés (conflits, priorités,…).

2.4 MONTEE EN ABSTRACTION

Dans le niveau d'abstraction le plus élevé, nous avons proposé d'architecturer les applications en termes de spécifications de services, de propriétés, contraintes, et préférences de composition qui doivent être garanties par la composition de l'application.

Les dépendances au niveau des spécifications permettent la substitution d'implémentations qui fournissent la même spécification (i.e., des implémentations qui fournissent les mêmes ressources et qui dépendent des mêmes ressources).

Or une interface Java ne représente pas toujours un niveau d'abstraction suffisant ; on pourrait préférer une définition plus abstraite, informelle ou semi-formelle, à laquelle plusieurs interfaces seraient conformes : une interface est alors une interprétation conforme de la définition abstraite. On pourrait aussi définir la notion de gabarit de composant (*template*). Un gabarit définit les ressources fournies d'un composant à services. L'architecture d'une application peut être définie en termes de gabarits et de connecteurs entre eux. De tels connecteurs, exprimant des dépendances de ressources, sont établis (exclusivement) lors de la définition de l'architecture de l'application.

Concrétiser une architecture en termes de gabarits, consiste ainsi à sélectionner pour chaque gabarit contenu une spécification qui fournit les ressources fournies définies par le gabarit, et qui dépend de l'ensemble des ressources défini par les connecteurs sortants du gabarit. Cependant, les choix réalisés dans une étape de sélection peuvent être des mauvais choix lors d'une étape ultérieure. Un algorithme de sélection avec des fonctionnalités efficientes de retour en arrière (*backtracking*) est donc nécessaire.

Monter le niveau d'abstraction permet ainsi la sélection et la substitution de spécifications, donnant plus de souplesse à la composition d'applications tout en préservant le contrôle sur la composition. De plus, une telle solution permet d'utiliser la même démarche d'abstraction et de concrétisation fournie par le mécanisme de groupes d'équivalence.

2.5 EVOLUTION PROACTIVE

Nous avons adopté une hypothèse simplificatrice pour l'évolution des applications : les changements réalisés sur un modèle d'application sont réalisés par l'administrateur ou l'utilisateur de l'application, et les opérations autorisées sont limitées.

Le traitement des changements proactifs programmés (conçus pour une application et réalisés quand un événement ou une condition dévient vraie) ou non-programmés (créés dynamiquement par exemple à partir des buts d'haut niveau) n'a pas été abordé et reste une perspective de ce travail.

Dans cette optique, nous nous intéressons à l'informatique autonomique dans l'intention d'explorer la réalisation d'applications capables de s'auto-administrer et d'auto-évoluer en fonction de leur contexte d'exécution.

2.6 GENIE LOGICIEL@RUN.TIME

La séparation claire entre la phase de développement et d'exécution des systèmes logiciels est un des dogmes de l'informatique traditionnelle. La recherche en génie logiciel s'est concentrée principalement sur les phases de conception, développement et déploiement de systèmes dans le but de produire de logiciels de qualité, i.e., des systèmes qui effectuent correctement leurs fonctionnalités et dont le comportement est identique et reproductible sur toutes les installations des clients. Par conséquent, de nombreuses méthodes, techniques et outils ont été proposés pour assurer la satisfaction des exigences fonctionnelles et non-fonctionnelles des systèmes avant leur exécution afin de garantir une exécution conforme aux exigences.

A cause du succès de ces méthodes, techniques et outils, moins d'attention a été apporté à la phase d'exécution. En effet, l'hypothèse traditionnelle implicite est que les logiciels sont exécutés dans un monde fermé et connu statiquement. Non seulement il n'y a aucune raison de modifier les logiciels pendant l'exécution mais, bien au contraire, les environnements d'exécution veillent à ce que ce ne soit pas possible, afin de garantir la cohérence et l'immuabilité du comportement à l'exécution. Malheureusement, non seulement les contextes d'exécution sont de plus en plus ouverts et variables, mais de plus en plus souvent les applications sont exigées d'assurer une disponibilité quasiment

ininterrompue : les systèmes doivent devenir flexibles, dynamiquement adaptables, reconfigurables, auto-administrables,… sans pour autant renoncer à la propriété fondamentale qui est de garantir une exécution conforme aux exigences.

De ce fait, ces dernières années et toujours dans le but de produire des systèmes de qualité, l'attention de la communauté de génie logiciel s'est focalisée aussi sur la phase d'exécution [48]. Diverses approches, comme les models@run.time, ont été proposées dans le but de pouvoir concevoir, programmer, déployer,… exécuter ces systèmes, mais surtout de pouvoir les gérer et les adapter lors de leur exécution.

Plus précisément, l'approche des modèles à l'exécution vise à effacer la frontière entre les phases de développement et d'exécution des systèmes, et se propose ainsi d'étendre à l'exécution l'utilisation des modèles produits dans la phase de développement.

L'approche proposée dans ce travail de thèse participe à cette vision : le même métamodèle est utilisé de la conception jusqu'à l'exécution des applications ; permettant ainsi de concevoir, de développer et d'exécuter des applications à services tout en les adaptant dynamiquement à leur contexte d'exécution et aux variations de ceci. Notre approche étend cette vision dans la mesure où non seulement les modèles et les métadonnées sont présents à l'exécution, mais aussi l'environnement de développement et ses outils sont disponibles.

Ainsi, en accord avec les travaux récents en génie logiciel, il devient possible, lors de l'exécution, de réaliser de nombreuses activités, telles que la génération de tests, le débogage automatique, la génération de *proxies*, de médiateurs (syntaxiques, structurels ou sémantiques) ou même de composants métiers comme dans les lignes de produits.

Dans ce contexte, nous pouvons imaginer que des applications pourront s'adapter au contexte matériel fluctuant, via la sélection ou la génération de *proxies*, leur compilation et leur composition dans l'architecture de l'application. Nous pouvons aussi imaginer que l'application se trouvant confrontée à la présence d'autres applications, à priori inconnues, soit en mesure de générer des médiateurs réalisant des alignements syntaxiques et/ou sémantiques permettant d'établir des collaborations, négociations, symbioses inédites.

Nous pensons que le génie logiciel se dirige vers cette nouvelle façon de concevoir et construire des systèmes logiciels, pour lesquels une partie importante, voire principale, du génie logiciel sera réalisée automatiquement à l'exécution. Les défis sont immenses, à la mesure des besoins à satisfaire et des avancées qui seront ainsi permises. Nous estimons que nos travaux sont un pas de plus vers le génie logiciel à l'exécution.

ANNEXE A
LANGAGE DE COMPOSITION ET DE CONTRAINTES

1 LANGAGE DE COMPOSITION

```
<CompositeSpecification> ::= 'CompositeSpecification' <CompositeSpecificationName> '{'
    <Properties>
    <Attributes>
    <Capabilities>
    <Content>
    <Dependencies>
    <Preferences>
'}'

<Properties> ::=  ( 'property' <Resolution> )? ( 'property' <Visibility> )? ( 'property' <Instantiable> )?
                  ( 'property' <Multiple> )? ( 'property' <Shared> )? ( 'property' <Singleton> )?
<Resolution> ::= 'resolution' <Equals> ('closed' | 'open') ';'
<Visibility> ::= 'visibility' <Equals> ('blackbox' | 'whitebox') ';'
<Instantiable> ::= 'instantiable' <Equals> <Boolean> ';'
<Multiple> ::= 'multiple' <Equals> <Boolean> ';'
<Singleton> ::= 'singleton' <Equals> <Boolean> ';'
<Shared> ::= 'shared' <Equals> <Boolean> ';'

<Attributes> ::= ( <Attribute> )*
<Attribute> ::= 'attribute' <AttributeName> ( <Equals> <AttributeValue> )? 'type' <AttributeType> ';'

<Capabilities> ::= ( <Capability> )+
<Capability> ::= 'provides' <InterfaceName> 'from' <ResolutionMode>? <SpecificationName> ';'

<Content> ::= ( <Content_> )*
<Content_> ::= 'contains' <Cardinality>? <Optionnality>? <ResolutionMode>? < SelectionExpression > ';'

<Dependencies> ::= ( <Dependency> )*
<Dependency> ::= 'requires' <Cardinality>? <Optionnality>? <ResolutionMode>? <SelectionExpression> 'id' <ID> ';'

<Preferences> ::= ( <Preference> )*
<Preference> ::= 'prefers' <SelectionExpression> ';'

<ResolutionMode> ::= 'static' | 'dynamic'
<Cardinality> ::= 'multiple' | 'simple'
<Optionnality> ::= 'optional' | 'mandatory'
<Boolean> ::= 'true' | 'false'
```

2 LANGAGE DE SELECTION

```
<Expression> ::= <ConstraintExpression> | <SelectionExpression>
<ConstraintExpression> ::= <ContextualConstraint> | <IntrinsicConstraint>
<ContextualConstraint> ::= <Rule> ';'
<IntrinsicConstraint> ::= <FilterExpression> ';'

<SelectionExpression> ::= <Rule> ';'
<Rule> ::= <Set> ( <Navigation> )* | <FilterExpression>
<Set> ::= ( <Class> | '#' <VariableName> ) (<Level>)? ( <FilterExpression> )?
<Navigation>::= ('.' | '..') <RelationName> ( ['<Cardinality>'] )? ( <FilterExpression> )?
<Class> ::= any type T| 'Self'
<Level> ::= any level L
// Pour COMPASS, T désigne un type Specification ou Implementation, et
// L désigne un niveau Implementation ou Instance (le niveau où l'expression sera évaluée)

<FilterExpression> ::= <Logical> | <Relational>
<Logical> ::= ( <NotOperator> <FilterExpression> ) | '(' <FilterExpression> <BinaryOperator> <FilterExpression> ')'
<Relational> ::= '(' <Attribute> <RelationalOperator> '"'? <Value> '"'? ')'

<NotOperator> = 'NO'
<BinaryOperator> = 'AND' | 'OR'
<RelationalOperator> ::= '=' | '!=' | '<' | '>' | '<=' | '>='
```

ANNEXE B
PROPRIETES APAM

	Propriété	Sémantique
Résolution	***borrow*** | ***implementation instance*** [*true* | *false* | *Expression*]	*Cette propriété spécifie si un composant composite (implémentation composite et instance composite) peut utiliser/importer des composants (implémentations et instances respectivement) appartenant à d'autres composants composites (true), ou s'il doit utiliser/déployer ses propres composants (false) lors du processus de résolution. L'expression Expression définit (avec un filtre LDAP) les composants qui peuvent être importés : tout composant importé doit satisfaire l'expression. La valeur de cette propriété est par défaut true.*
Visibilité et Partage	***friend*** | ***implementation instance*** [*true* | *false* | *Expression*]	*Cette propriété indique si un composant composite (implémentation composite et instance composite) prête/exporte ses composants (implémentations et instances respectivement) à des composants composites particuliers (true), ou non (false). L'expression Expression spécifie les composants qui peuvent être utilisés par des composants composites particuliers (liés au composant composite par une relation friend) : l'expression doit être satisfaite par tout composant prêté à un composant composite lié par la relation friend.Par défaut, la valeur de cette propriété est false.* *Note : par défaut, tous les composants d'un composant composite peuvent être utilisés par tout autre composant composite dans la même plate-forme.*
	local | ***implementation instance*** [*true* | *false* | *Expression*]	*Cette propriété spécifie si un composant composite (implémentation composite et instance composite) ne prête pas ses composants (implémentation et instances respectivement) à d'autres composants composites (true). L'expression Expression définit les composants qui ne peuvent pas être utilisés par d'autres composants composites : tout composant satisfaisant l'expression n'est pas prêté.La valeur de cette propriété est par défaut false.*
	application instance [*true* | *false* | *Expression*]	*Cette propriété indique si une instance composite prête ses instances à toute autre instance composite appartenant à la même application (true), ou non (false). L'expression Expression spécifie les instances qui peuvent être utilisées par des instances composites appartenant à la même application : l'expression doit être satisfaite par toute instance prêtée à une instance composite de la même application.Par défaut, la valeur de cette propriété est false.*

Tableau 11. Propriétés contextuelles prédéfinies dans APAM

REFERENCES

[1] M. Weiser, "Ubiquitous Computing," *Computer*, vol. 26, no. 10, pp. 71–72, 1993.

[2] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *Personal Communications, IEEE*, no. August, pp. 10–17, 2001.

[3] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering (FOSE)*, 2007, pp. 37–54.

[4] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] D. A. Taylor, *Object technology: a manager's guide*. Addison-Wesley, 1998.

[6] K.-K. Lau and Z. Wang, "Software Component Models," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[7] M. P. Papazoglou, "Service-Oriented Computing: Concepts, Characteristics and Directions," *International Conference on Web Information Systems Engineering*, pp. 3–12, 2003.

[8] M. D. McIlroy, "Mass-Produced Software Components," *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[9] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. USA: Addison-Wesley, 2001.

[10] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 187–197, 2002.

[11] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software," vol. 3, 2001.

[12] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," *SERA*, 2006.

[13] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron, "A Classification Framework for Component Models," *Management*, pp. 1–20, 2008.

[14] E. Bruneton, T. Coupaye, and J.-B. Stefani, "The Fractal Component Model," 2004.

[15] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and its Support in Java," *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 2006.

[16] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye, "A Component Model Engineered with Components and Aspects," in *Component-Based Software Engineering*, 2006, pp. 139–153.

[17] W.-J. Papazoglou, Mike P. and Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, 2007.

[18] A. Arsanjani, "Service-oriented modeling and architecture: How to identify, specify, and realize services for your SOA." 2004.

[19] OSGi Alliance, "OSGi Service Platform Core Specification," 2007. [Online]. Available: http://www.osgi.org/Specifications/.

[20] G. Alonso, F. C. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[21] H. Cervantes, "Vers un Modèle à Composants orienté Services pour Supporter la Disponibilité Dynamique," 2004.

[22] H. Cervantes and R. S. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," in *26th International Conference on Software Engineering (ICSE'04)*, 2004, pp. 614–623.

[23] Osg. Alliance, "OSGi Service Platform Service Compendium." 2007.

[24] M. Offermans, "Automatically managing service dependencies in OSGi," *Design*, pp. 1–12, 2005.

[25] A. M. Coyler, H. Hildebrand, C. Leau, and A. Piper, "Spring Dynamic Modules Reference Guide," 2009.

[26] M. Beisiegel, A. Miller, J. Marino, and L. Waterman, "SCA Service Component Architecture," *International Business*, 2007.

[27] C. Escoffier, "iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques," 2008.

[28] E. Simon, "SAM : un environnement d'exécution pour les applications à services dynamiques et hétérogènes," 2011.

[29] D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Knowledge Creation Diffusion Utilization*, no. January, 1994.

[30] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[31] Bass, Clements, and Kazman, *Software Architecture in Practice*. Addison-Wesley, 1997.

[32] S. Engineering and S. Committee, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems," *October*, 2000.

[33] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[34] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[35] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, Second Edi. Addison-Wesley, 2010.

[36] D. L. Parnas, "On a 'Buzzword' Hierarchical Structure," *Proc IFIPS Congress*, pp. 336–339, 1974.

[37] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[38] P. Kruchten, "Architecture blueprints---the '4+1' view model of software architecture," *Tutorial proceedings on TRI-Ada "91 Ada"s role in global markets: solutions for a changing complex world - TRI-Ada '95*, vol. 12, no. November, pp. 540–555, 1995.

[39] P. Herzum and O. Sims, *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. Wiley, 2000.

[40] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2005.

[41] D. Garlan, R. T. Monroe, and D. Wile, "Acme: architectural description of component-based systems," in *Foundations of component-based systems*, M. Leavens, Gary T. and Sitaraman, Ed. New York, NY, USA: Cambridge University Press, 2000, pp. 47–67.

[42] D. Luckham, "Rapide: A language and toolset for simulation of distributed systems by partial ordering of events," in *Partial order methods in verification: DIMACS workshop, Princeton University*, 1996, vol. 29.

[43] R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Configurations*, pp. 1–15, 1998.

[44] M. Riveill and A. Senart, "Aspects dynamiques des langages de description d'architecture logicielle." .

[45] J. Magee and J. Kramer, "Dynamic structure in software architectures," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 3–14, Nov. 1996.

[46] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill, "Component-based Programming and Application Management with Olan," *Work*.

[47] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani, "Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset," *29th International Conference on Software Engineering (ICSE'07)*, pp. 209–219, May 2007.

[48] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[49] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, "Models@run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[50] J.-M. Favre, J. Estublier, and M. Blay-Fornarino, *L'ingénierie dirigée par les modèles : au-delà du MDA*. Cachan, France: Hermes-Lavoisier, 2006.

[51] J. Bezivin and O. Gerbe, "Towards a precise definition of the OMG/MDA framework," *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 273–280.

[52] OMG, "Model Driven Architecture." .

[53] J. Ludewig, "Models in software engineering - an introduction," *Software and Systems Modeling*, vol. 2, no. 1, pp. 5–14, Mar. 2003.

[54] E. Seidewitz, "What models mean," *Software, IEEE*, vol. 20, no. 5, pp. 26–32, Sep. 2003.

[55] M. Fowler, *UML Distilled*. Addison-Wesley, 1999.

[56] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, "An Algebraic View on the Semantics of Model Composition."

[57] G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak, "Meta-modeling Runtime Models," *Models in Software Engineering*, pp. 209–223, 2011.

[58] T. Vogel, A. Seibel, and H. Giese, "Toward Megamodels at Runtime," in *Proc. of the 5th Intl. Workshop on Models@ run. time. CEUR-WS. org*, 2010, vol. 641, pp. 13–24.

[59] A. K. Dey and G. D. Abowd, "Towards a better understanding of context and context-awareness," *on the what, who, where, when, and how of context-awareness*, 2000.

[60] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[61] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications," *Workshop on Mobile Computing Systems and Applications*, pp. 85–90.

[62] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, Jul. 2004.

[63] P. Oreizy, "Issues in the Runtime Modification of Software Architectures," *System*, pp. 1–8, 1996.

[64] J. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," *Policies for Distributed Systems and Networks. POLICY 2004.*, 2004.

[65] J. Kramer and J. Magee, "The evolving philosophers problem: dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[66] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "An alternative to Quiescence: Tranquility," in *22nd IEEE International Conference on Software Maintenance*, 2006, pp. 73–82.

[67] L. Baresi, R. Heckel, S. Thone, and D. Varro, "Style-based refinement of dynamic software architectures," *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pp. 155–164.

[68] F. Plášil, D. Bálek, and R. Janec, "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating," *Foundations*.

[69] P.-C. David and T. Ledoux, "WildCAT: a generic framework for context-aware applications," in *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, 2005.

[70] P.-C. David, "Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation," 2005.

[71] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, pp. 46–54, 2004.

[72] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[73] S.-W. Cheng, A.-C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "An architecture for coordinating multiple self-management systems," *Conference of Software Architecture*, 2004.

[74] N. de Palma, S. Bouchenak, F. Boyer, D. Hagimont, S. Sicard, and C. Taton, "Jade: un environnement d'administration autonome," *Techniques et sciences informatiques*, vol. 27, no. 8, pp. 1225–1252, Oct. 2008.

[75] A. Phung-khac, M. Segarra, J. Gilliot, and A. Beugnard, "Dynamic Composition and Adaptation in Adapt-Medium," *Read*.

[76] J. Yang and M. Papazoglou, "Service components for managing the life-cycle of service compositions," *Information Systems*, vol. 29, no. 2, pp. 97–125, 2004.

[77] B. Orriens, J. Yang, and M. P. Papazoglou, "Model driven service composition," *Service-Oriented Computing-ICSOC*, 2003.

[78] G. Pedraza-Ferreira, "FOCAS : un canevas extensible pour la construction d'applications orientées procédé," 2009.

[79] E. Zimanyi, A. Pirotte, and T. Yakusheva, "Materialization : a powerful and ubiquitous pattern abstraction," pp. 630–641, 1994.

[80] M. Dahchour, A. Pirotte, and E. Zima, "Materialization and Its Metaclass Implementation," vol. 14, no. 5, pp. 1078–1094, 2002.

[81] C. Atkinson and T. Kühne, "The essence of multilevel metamodeling," *«UML» 2001—The Unified Modeling Language. ...*, 2001.

[82] J. J. Odell, *Advanced Object-Oriented Analysis and Design Using UML*. Cambridge University Press, 1998.

[83] C. Gonzalez-Perez and B. Henderson-Sellers, "A powertype-based metamodelling framework," *Software & Systems Modeling*, vol. 5, no. 1, pp. 72–90, Nov. 2005.

[84] J. Estublier, I. A. Dieng, E. Simon, and G. Vega, "Flexible Composites and Automatic Component Selection for Service-Based Applications," in *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2009.

[85] I. Dieng, "SELECTA : une approche de construction d'applications par composition de services," 2010.

[86] J. Estublier, G. Vega, P. Lalanda, and T. Leveque, "Domain Specific Engineering Environments," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2008.

[87] J. Estublier and G. Vega, "Managing Mutiple Applications in a Service Platform," in *Principles of Engineering Service Oriented Systems (PESOS)*, 2012.

[88] J. Estublier and G. Vega, "Reconciling Components and Services: The Apam Component-Service Platform," *2012 IEEE Ninth International Conference on Services Computing*, pp. 683–684, Jun. 2012.

[89] J. Bardin, P. Lalanda, and C. Escoffier, "Towards an Automatic Integration of Heterogeneous Services and Devices," in *IEEE Asia-Pacific Services Computing Conference (APSCC)*, 2010, pp. 171–178.

[90] B. Neumayr and M. Schrefl, "Comparison criteria for ontological multi-level modeling," *Dagstuhl Seminar on Conceptual Modelling*, no. 08, 2008.

PUBLICATIONS

Les travaux discutés dans cette thèse ont été présentés précédemment :

- Diana Moreno-Garcia, Jacky Estublier. **Model-Driven Design, Development, Execution and Management of Service-Based Applications**. In Proceedings of the 9th International Conference on Services Computing (SCC), Honolulu, Hawaii, USA, 2012.

- Diana Moreno-Garcia and Elmehdi Damou. **Model-driven execution of service-based applications**. In Proceedings of the 7es journées sur l'Ingénierie Dirigée par les Modèles (IDM), Lille, France, 2011.

- Jacky Estublier, Idrissa Dieng, Eric Simon and Diana Moreno-Garcia. **Opportunistic Computing Experience with the SAM platform**. In Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems (PESOS), Cape Town, South Africa, 2010.

- Eric Simon, Jacky Estublier and Diana Moreno-Garcia. **Extensible and General Service-Oriented Platform: Experience with the Service Abstract Machine**. In Proceedings of the 7th International Conference on Service Computing (SCC), Miami, Florida, USA, 2010.