ApAM : Un environnement pour le développement et l'exécution d'applications ubiquitaires

Elmehdi Damou

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : le 6 janvier 2005 -7 août 2006

Présentée par

Elmehdi DAMOU

Thèse dirigée par **Jacky Estublier** et
codirigée par **Germán Eduardo VEGA BAEZ**

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

ApAM : Un environnement pour le développement et l'exécution d'applications ubiquitaires

Thèse soutenue publiquement le **25 octobre 2013**,
devant le jury composé de :

M. Vivien Quema
Professeur à l'Institut Polytechnique de Grenoble, Président
M. Jean-Yves TIGLI
Maître de Conférences l'Université de Nice Sophia Antipolis, Examinateur
M. Michel Riveill
Professeur à l'Université de Nice Sophia Antipolis, Rapporteur
M. Lionel Seinturier
Professeur des Universités à l'Université de Lille, Rapporteur
M. Charles Consel
Professeur à l'Institut Polytechnique de Bordeaux, Examinateur
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

REMERCIEMENTS

Je tiens tout d'abord à remercier tous les membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je remercie plus particulièrement Lionel Seinturier et Michel Riveill d'avoir accepté de rapporter mon travail de thèse. Je remercie également Charles Consel et Jean-Yves Tigli pour avoir examiné mes travaux. Merci aussi à Vivien Quema, président de mon jury.

J'adresse mes sincères remerciements à German Vega et à Jacky Estublie pour leur confiance, leurs conseils, leur soutien et leur aide tout au long de ce travail. Je souhaite aussi remercier Philippe LALANDA, Didier Donsez et Pierre Yves Cunin qui m'ont soutenu et conseillé durant mon parcours au sein de l'équipe ADELE.

Je remercie aussi l'ensemble des membres de l'équipe ADELE, tout particulièrement ceux qui ont contribué à l'amélioration et l'avancement de mon travail : Diana Moreno, Jander Nascimento, Thibaud Flury. Merci aux collègues de l'équipe ADELE qui m'ont aidé dans mon travail à travers nos discussions et leurs conseils : Thomas Leveque, Clément Escoffier, Ozan Gunalp, Issac Garcia, Walter Rudamtkin, Jonathan Bardin, Kiev Gama, Idrissa Dieng, Pierre-Alain Avouac, Bassem Debbabi, Marc Quast, Gabriel Pedraza, Yu Jianqi, Eric Simon, Morgan Martinet, Lionel Touseau, Pierre Bourret, Etienne Gandrille, Yoann Morel, Vincent Lestideau, Ada Diaconescu et Stéphanie Chollet. Un merci aux collègues des autres équipes avec qui je n'ai pas directement travaillé pour leur la convivialité et pour la bonne ambiance: Mario Cortes, Nicolas Hili, Luz Maria Priego et à tous ceux que j'aurais pu oublier. Je voudrais remercier tous mes amis à Grenoble et les familles Jacquot et Dufourd qui m'ont encouragé tout en long de ces années.

Je remercie énormément ma famille qui m'a toujours soutenu. Tout spécialement mes parents Touria et Abdeslam et aussi mes sœurs Maha et Nour El Houda.

Merci Bérénice pour ton soutien, ta patience et ta présence au quotidien à mes côtés ces dernières années.

RÉSUMÉ

Simplifier notre interaction avec les entités informatiques interconnectées de notre environnement et faciliter l'exploitation des informations générées par celles-ci est l'objectif des environnements et des applications ubiquitaires. Le comportement des applications ubiquitaires dépend de l'état et de la disponibilité des entités (logiciels ou dispositifs) qui composent l'environnement ubiquitaire dans lequel elles évoluent, ainsi que des préférences et localisations des utilisateurs.

Développer et exécuter des applications ubiquitaires est un véritable défi que notre approche essaie de relever au travers de l'environnement d'exécution ApAM. Considérant que l'environnement d'exécution est imprévisible, nous partons du principe qu'une application ubiquitaire doit disposer d'une grande flexibilité dans le choix de ses composants et que cette composition doit être automatique. Nous proposons une description abstraite et implicite de la composition (où les composants et les liens entre eux ne sont pas décrits explicitement), ce qui permet de construire l'application incrémentalement pendant la phase d'exécution. La plate-forme d'exécution ApAM implémente ces mécanismes de composition incrémentale et s'en sert pour conférer aux applications ubiquitaires la capacité de « résister » et de s'adapter aux changements imprévisibles de l'environnement d'exécution. Cette propriété dite de résilience est au cœur de notre approche car elle permet aux programmeurs de développer « simplement » des applications « résilientes » sans avoir à décrire les diverses adaptations à réaliser et même sans connaître toutes les perturbations de l'environnement auxquelles elles seront soumises.

Notre proposition offre le moyen de développer et d'exécuter des applications ayant un haut niveau de résilience vis-à-vis des évolutions de leur contexte d'exécution, grâce à des mécanismes automatiques capables de construire et de modifier à l'exécution l'architecture logicielle des applications ubiquitaire. Les mécanismes fournis sont génériques mais peuvent être étendus et spécialisés pour s'adapter plus finement à certaines applications ou à des domaines métiers spécifiques.

Mots-clés : environnement ubiquitaire – application dynamique – architecture logicielle – approche à composants – approche à services – composition de services – espace de résolution – résilience

ABSTRACT

The goal of ubiquitous environments and applications is to simplify our interaction with interconnected software and hardware entities, and to allow the exploitation of the information that they gather and generate. The behavior of ubiquitous applications depends on the state and the availability of the software and hardware entities that compose the ubiquitous environment in which they are constantly evolving, as well as, the preferences and locations of users.

Developing and executing ubiquitous applications is a difficult challenge that our approach attempts to address with the creation of the ApAM execution environment. Knowing that the execution environment is unpredictable, we believe that ubiquitous applications require a large amount of flexibility in choosing the components that compose the application, and that the composition should be automated. We propose an abstract and implicit description of the composition, where components and bindings are not explicitly described. This allows to incrementally building an application at runtime. The ApAM execution platform implements the mechanisms to achieve incremental composition and uses them to provide ubiquitous applications with the resilience and adaptability necessary to face unpredictable changes that originate in the execution environment. Resilience is a core property of our approach because it allows developers to easily build applications without the need to either describe nor predict the multiple adaptations required to support environmental disturbances which the applications will encounter.

Our proposal offers the means of developing and executing applications with a high level of resilience in regards to their continuously evolving context. This is possible thanks to the mechanisms described in this dissertation that allow building and changing, at runtime, ubiquitous applications. These mechanisms are generic but can be extended or specialized in order to solve domain or application-specific issues.

Keywords: ubiquitous computing – dynamic applications – software architecture – Component-based software engineering – service-oriented computing – service composition – resolution space – resilience

TABLE DES MATIÈRES

DEUXIÈME PARTIE : CONTRIBUTIONS

CHAPITRE 4. PROPOSITION

CHAPITRE 5. APAM : MODÈLE DE COMPOSANTS A SERVICES

TABLE DES FIGURES

TABLE DES TABLEAUX

CHAPITRE 1
INTRODUCTION

1. CONTEXTE ET PROBLÉMATIQUES

Les environnements ubiquitaires[Weis96] sont des environnements contenant un grand nombre de dispositifs interconnectés et dynamiques. Les environnements domotiques représentent un excellent exemple de cette invasion de l'informatique dans notre quotidien. Dans nos maisons, il est prévu que les Set-Top-Box (les box) soient au cœur de cet écosystème. Elles doivent faire cohabiter les applications dites ubiquitaires de tous genre : sécurité, confort, médical, etc. Ces applications vont interagir avec plusieurs appareils tels que les alarmes, les téléphones, les fenêtres, les portes, etc. dans le but de simplifier notre interaction avec les entités informatiques interconnectées de notre environnement et faciliter l'exploitation des informations générées par celles-ci.

Les applications ubiquitaires sont des nouveaux types d'applications. Elles doivent être capables de s'adapter dynamiquement aux changements du contexte d'exécution car celui-ci évolue dynamiquement en réaction à la disponibilité des multitudes objets communicants et de leurs mobilités. Le comportement des applications ubiquitaires dépend de l'état et de la disponibilité des entités (logicielles ou dispositifs) qui composent l'environnement dans lequel elles évoluent, ainsi que des préférences et localisation des utilisateurs. La reconfiguration de leurs architectures et leurs infrastructures doit être réalisé automatiquement.

Les méthodes habituelles du génie logiciel qui consistent à définir une application indépendamment du contexte d'exécution par une liste exhaustive de tous ses composants sont contraignantes, voire même impossible à appliquer pour les applications ubiquitaires. Ces applications ne peuvent pas être complétement figée avant leur exécution. Une application ubiquitaire doit être définie de manière suffisamment flexible afin de pouvoir s'adapter dynamiquement au contexte d'exécution. En même temps, cette description doit être suffisamment fiable et précise pour permettre de construire à l'exécution une « version » de l'application qui soit à la fois compatible avec le contexte courant et correcte du point de vue de ses fonctionnalités. Cette même description doit permettre de construire dynamiquement une nouvelle version de l'application lorsque l'évolution du contexte d'exécution rend la version courante inadaptée ou incohérente. Cette évolution dynamique confère à l'application un certain degré de résilience par rapport aux évolutions du contexte d'exécution.

Nous pensons que les applications ubiquitaires font face à deux défis principaux. Premièrement, le langage de définition de l'application devrait permettre de décrire des solutions pour un large éventail de contextes d'exécution possibles. Deuxièmement, comme l'application doit faire face à des changements imprévisibles, les décisions d'adaptation doivent être prises lors de l'exécution. En effet, ces applications doivent prendre en considération le caractère imprévisible du contexte d'exécution, ainsi que les aspects métier de l'application. La construction de ces applications se fait par composition de service, cependant elles doivent disposer d'une grande flexibilité dans le choix de leurs composants et la composition doit être automatique. Cette flexibilité permet d'augmenter les possibilités de trouver des solutions quand l'application fait face à des changements imprévisibles qui, autrement, provoqueraient l'échec de l'application.

2. OBJECTIFS

Nous partons du principe qu'une application ubiquitaire doit disposer d'une grande flexibilité dans le choix de ses composants et que cette composition doit être automatiques. Nous proposons une **description abstraite et implicite** de la composition (où les composants et les liens entre eux ne sont pas décrits explicitement), ce qui permet de **construire l'application incrémentalement pendant la phase d'exécution.** La plate-forme d'exécution ApAM implémente ces mécanismes de composition incrémentale et s'en sert pour conférer aux applications ubiquitaires la capacité de « résister » et de s'adapter aux changements imprévisibles de l'environnement d'exécution. Cette propriété dite de résilience est au cœur de notre approche car elle permet aux programmeurs de développer « simplement » des applications « résilientes » sans avoir à décrire les diverses adaptations à réaliser, et même sans connaitre toutes les perturbations de l'environnement auquel elles seront soumises.

Dans ce travail une application est définie à travers une **architecture de référence** à partir de laquelle la plate-forme d'exécution construit une **architecture concrète** qui représente l'application à l'exécution dans le contexte courant. La plate-forme garantit à la fois la **conformité** et la **cohérence** entre les deux architectures.

Nos objectifs sont les suivants :

- Assurer la conformité entre l'architecture de référence et l'architecture concrète :
  - Permettre plusieurs concrétisations de l'architecture de référence.
  - Permettre de modifier l'architecture à l'exécution et d'automatiser le passage d'une architecture à une autre.
  - Garantir, qu'à tout moment, l'architecture concrète est conforme à l'architecture de référence.
- Automatiser la sélection des composants entrant dans la construction de l'application :
  - Ouvrir le spectre dans lequel les composants peuvent être sélectionnés.
  - Réaliser la sélection de façon automatique et fiable.
- Réagir automatiquement aux changements du le contexte d'exécution par :
  - Le maintien des fonctionnalités de l'application en exécution.
  - Un comportement résilient face aux variations de celui-ci.

Les travaux de cette thèse visent ainsi à simplifier le développement, la construction et l'exécution des applications ubiquitaires et à leurs **offrir la propriété de résilience,** pour « résister » et s'adapter aux changements imprévisibles de l'environnement d'exécution. Pour cela, notre approche combine les avantages de plusieurs technologies et méthodes de génies logiciels tels que l'approche à composants, l'approche à services et les architectures logicielles.

3. ORGANISATION DU DOCUMENT

Le document est divisé en trois grandes parties : l'état de l'art, la contribution et finalement l'implémentation et les expérimentations.

L'état de l'art est présenté en deux chapitres :

- le **chapitre 2** présente l'informatique ubiquitaire en général et décrit brièvement les différents travaux et approches ayants abordés ce domaine.
- Le **chapitre 3** expose les approches à composants et à services ainsi que les architectures logicielles. Après une présentation générale de ces approches, nous soulignons le rôle de chacune d'elles à répondre aux défis de l'informatique ubiquitaires.

La contribution est présentée en deux chapitres :

- le **chapitre 4** présente la vision globale de notre approche pour la construction des applications ubiquitaires à l'exécution. Ce chapitre décrit aussi de façon générale, l'ensemble des mécanismes qui nous ont permis d'apporter plus de résilience aux applications.
- le **chapitre 5** détaille la mise en œuvre de notre approche. Nous présentons les principes et les mécanismes sur lesquels nous avons construit notre plate-forme ApAM. Nous déroulons un exemple d'application ubiquitaire illustrant les points clés de notre approche.

La partie implémentation et expérimentation est présentée en deux chapitres :

- le **chapitre 6** est présente l'implémentation de notre proposition pour la construction d'applications à services flexibles et résilientes. Il détaille les couches basses sur lesquelles ApAM a été réalisé et comment cette liaison a été établie entres les couches. Enfin, nous présentons quelques exemples de gestionnaires servant à étendre le mécanisme de résolution de la plate-forme
- Le **chapitre 7** survole les différents projets dans lequel ApAM est utilisé. Nous soulignons le rôle de notre plate-forme dans chacun de ces projets, puis nous évaluons notre approche du point de vue fonctionnel par rapport aux applications existantes à travers un exemple d'application. Nous calculons enfin le surcout en performance à l'exécution qu'engendre l'utilisation de la plate-forme ApAM.

Enfin, le **chapitre 8** fait le point sur nos principales contributions et indique quelques perspectives possibles de nos travaux.

PREMIÈRE PARTIE :

LES ARCHITECTURES LOGICIELLES POUR LES APPLICATIONS UBIQUITAIRES

CHAPITRE 2
L'INFORMATIQUE UBIQUITAIRE

1. INTRODUCTION

Dans cette partie, nous ferons le point sur l'informatique ubiquitaire. Dans un premier temps nous définirons ce qu'est un environnement ubiquitaire : les origines de ce domaine ainsi que les évolutions qu'il a connu. Puis, dans un second temps nous listerons les défis qu'affrontent les applications qui évoluent dans un tel environnement.

1.1. DÉFINITIONS : UBIQUITAIRE ET PERVASIF

L'informatique ubiquitaire a pour but de faciliter l'utilisation de l'informatique et des dispositifs pour rendre notre quotidien plus confortable. Le monde idéal que Mark Weiser a imaginé part d'une idée simple [Weis91] :

> ***« Make a computer so imbedded, so fitting, so natural, that we use it without even thinking about it. »***

C'est cette vision qui constitue l'informatique ubiquitaire. Cette idée tranche avec l'usage classique de l'informatique, où une machine offre un environnement virtuel dans lequel s'accomplit une tâche qui a un début et une fin [SaMu03]. L'environnement informatique doit être une description améliorée du monde réel et physique.

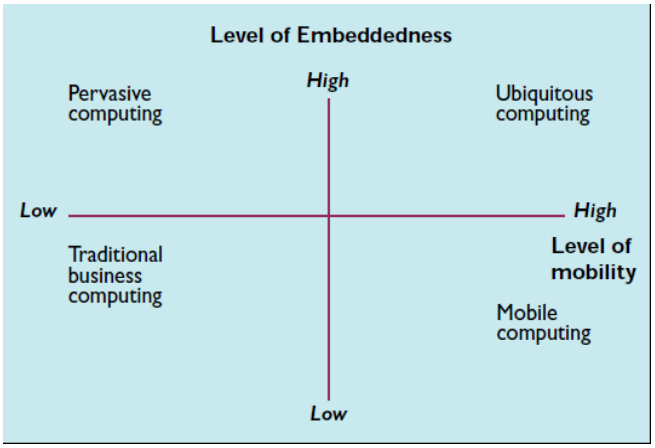


Figure 1 : Dimension de l'informatique ubiquitaire [LYOA02]

L'**informatique pervasive** et l'informatique ubiquitaire sont des domaines très proches dans la littérature. Ces termes sont d'ailleurs utilisés comme des synonymes, mais ils sont conceptuellement différents [LYOA02]. D'après la Figure 1, l'informatique pervasive se concentre sur l'abstraction du contexte, alors que l'informatique ubiquitaire combine ce modèle avec l'informatique mobile pour offrir à l'utilisateur une vision globale de l'environnement pendant ses déplacements [ZhWa10].

Dans la suite de cette thèse, nous nous intéresserons au domaine de l'informatique ubiquitaire et nous ne ferons pas de distinction entre les termes ubiquitaire et pervasive. Il existe d'autres domaines en liaison étroite avec ce modèle informatique : intelligence ambiante[1], internet des choses[2], systèmes sensibles au contexte[3], etc.

1 Ambient intelligence
2 Internet of things
3 Context-awareness systems

1.2. HISTORIQUE

Pour trouver l'origine de ce nouveau modèle informatique, il faut revenir à l'origine de l'informatique et à l'évolution de l'ordinateur. En effet, au début de l'informatique, les machines étaient très coûteuses et nécessitaient énormément de place. Une machine était donc partagée par plusieurs utilisateurs. Petit à petit, les machines sont devenues accessibles à tout le monde : c'est l'aire du Personnal Computer (PC). Chaque personne dispose ainsi de sa propre machine. C'est aussi l'époque de la première évolution des technologies de communications et des réseaux : les **systèmes distribués** ont alors émergé lors de cette première vague d'évolution.

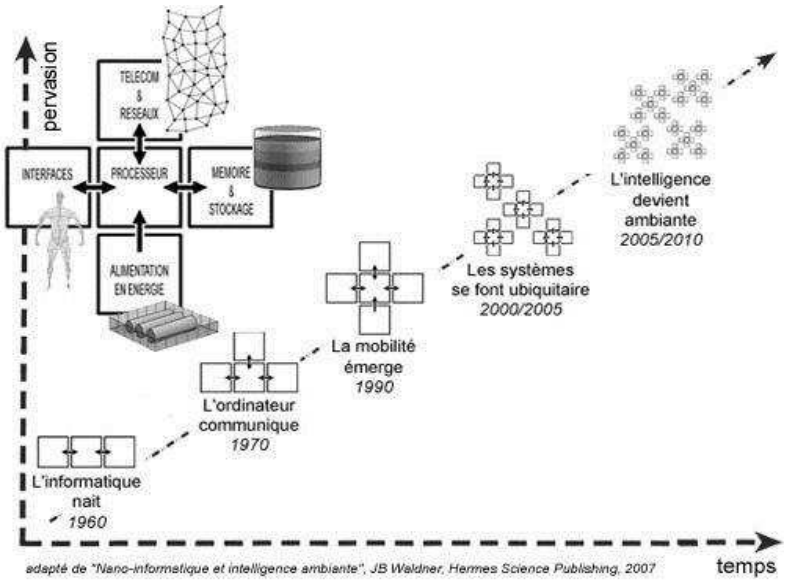


Figure 2 : L'évolution de l'informatique (tiré de Wikipedia)

Simultanément, les machines ont pris de moins en moins de place et ont envahi notre environnement quotidien. Les utilisateurs manipulent plusieurs machines à la fois. Ces dispositifs ont mis en avant de nouvelles technologies de communication (Bluetooth, Wifi, GSM, RFID, etc.) et ils ont su tirer profit de l'émergence d'internet [SaMB03] : c'est l'aire des **systèmes mobiles** (voir Figure 2).

L'informatique fait aujourd'hui parti de notre quotidien et les besoins des utilisateurs vis-à-vis des équipements évoluent. Les informations collectées sont de plus en plus volumineuses et les utilisateurs ont de plus en plus de mal à les interpréter. Être autonome, sensible au contexte, et capable d'offrir une information claire et utile est ce qu'on attend désormais des machines. Les **systèmes ubiquitaires** ont pour but de répondre à ces attentes.

Les systèmes ubiquitaires ont évolué avec l'évolution continue des besoins. Dans [LeCh11], trois générations de systèmes ubiquitaires ont été identifiées :

- La **première génération** des systèmes sensibles au contexte s'est concentrée sur la localisation des utilisateurs. Les travaux tels que : Active Badge System [WHFG92] et CyberGuide [AAHL97] ont été mis en place pour fournir une information utile selon l'emplacement des utilisateurs.
- La **deuxième génération** se veut plus générique en essayant de gérer différents types d'informations liées au contexte. Nous avons vu apparaître des Frameworks pour aider au développement des applications sensibles aux contextes. Parmi les systèmes les plus

connus : Context Toolkit [SaDA99], Hydrogen [HSPL03] ou plus spécialement Aura [GaSS02] et Gaia [RHCR02] qui offrent une vision similaire aux systèmes d'exploitation.

- Les systèmes de la **troisième génération** sont apparus avec l'adoption du Web Ontology Language (OWL) [McHa00] comme modèle pour représenter le contexte. Comme exemple de travaux nous pouvons citer CoBra [ChFJ03], SOCAM [GuPZ05] et CoCa [EjSB07]. Les préoccupations de ces travaux étant la montée en charge, la performance et la protection de la vie privée.

2. LES DÉFIS DE L'INFORMATIQUE UBIQUITAIRE

Quelques défis de l'informatique ubiquitaire résultent de la convergence de certains domaines : les systèmes distribués et les systèmes mobiles(voir Figure 3).

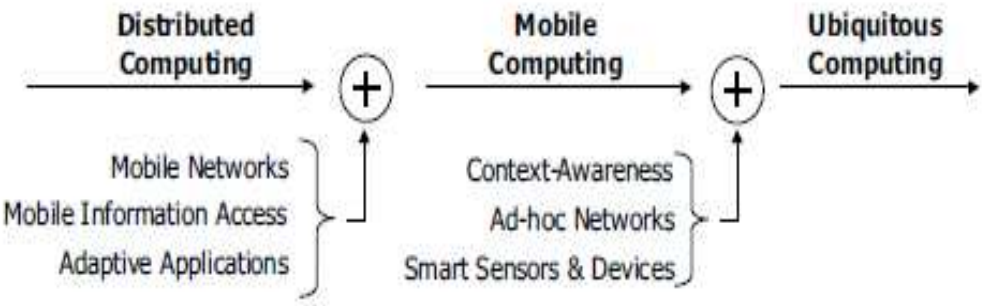


Figure 3 : Les problématiques de recherche dans l'informatique ubiquitaire [StLi04]

2.1. SYSTÈMES DISTRIBUÉS

Il est normal que l'on retrouve des problèmes liés aux systèmes distribués. Il s'agit d'une infrastructure incontournable pour le partage des ressources et pour l'accès distant aux informations. Il existe donc quelques défis semblables [Saty01] : accès distant aux informations, protocoles de communication, résistance aux pannes, disponibilité et sécurité.

Le domaine des systèmes distribués est largement abordé par de nombreux travaux de recherche [MaDK94][HoPu93]. Ce domaine représente l'infrastructure sur laquelle repose l'informatique ubiquitaire.

2.2. SYSTÈMES MOBILES

Les défis abordés par les systèmes mobiles sont liés à la gestion du déplacement de l'utilisateur. La gestion des réseaux mobiles, l'adaptation des applications, la gestion de l'énergie et la localisation sont les grandes catégories de défis où l'informatique mobile propose des solutions de plus en plus innovantes.

Ainsi, les travaux de ce domaine s'orientent vers la définition du contexte[ChKo00] ou la reconfiguration des composants et des connexions [McRo98]. Dans [FoZa94], les auteurs listes divisent les défis de l'informatique mobiles en trois catégories : La communication sans fil, la mobilité et la portabilité.

La reconfiguration dynamique, l'adaptation, les modes d'interactions et la sensibilité au contexte [GaKu03], font aussi des besoins que l'on retrouve dans l'informatique ubiquitaire.

3. LES DÉFIS SPÉCIFIQUES DU DOMAINE UBIQUITAIRE

C'est à partir des limitations de l'informatique mobile que l'on retrouve les réels défis de l'informatique ubiquitaire et pervasive.

3.1. DE L'ÉVOLUTIVITÉ À L'EXÉCUTION

3.1.1. LE CONTEXTE D'EXÉCUTION

Dans les systèmes mobiles, les équipements ne sont pas capables d'obtenir des informations sur le contexte dans lequel ils se trouvent de manière transparente et flexible. L'idée de l'informatique ubiquitaire est celle d'un monde qui intègre le contexte de façon transparente et utile (qui est le noyau central du système. Le concept même de contexte est très difficile à définir. Ce défi représente à lui seul un domaine de recherche. Les auteurs de [FHSE06] proposent une définition générale du contexte :

> ***« Context consists of elements directly representing the operating environment and more complex elements aggregated or derived from other elements. »***

La définition du contexte ne contient pas seulement les objets et les changements qui peuvent les atteindre. Il peut aussi se traduire par les emplacements et les identités des personnes aux alentours [ScTh94]. Une autre définition est donnée par [Dey01]. Le contexte est alors définit comme :

> ***« Any information that can be used to characterize the situation of entities (i.e., whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. »***

Le contexte est donc un concept vague et il ne s'agit pas ici de couvrir l'ensemble des définitions. D'une manière générale, le contexte regroupe plusieurs entités dans le but de les faire interagir (dans cette thèse on appelle « contexte » ou « contexte d'exécution » l'ensemble des entités qui représentent l'environnement ubiquitaire). Le premier défi de l'informatique ubiquitaire est de remédier à l'hétérogénéité des objets communicants ainsi qu'à leur distribution ou répartition dans l'environnement. La gestion de l'environnement implique d'autres défis pour l'informatique ubiquitaire. Représenter l'environnement est une tâche difficile, mais représenter son évolution est encore plus complexe. L'environnement ubiquitaire est un environnement proactif qui répond d'une manière transparente à l'évolution du contexte pour faciliter l'usage des technologies informatiques dans notre quotidien.

3.1.2. DYNAMISME DU CONTEXTE

Les environnements ubiquitaires ne sont pas figés dans le temps. Les entités de ce contexte évoluent constamment : elles sont dynamiques (elles peuvent apparaître et disparaître pendant l'exécution ou changer d'état). Ce comportement est imprévisible car dans la plupart des cas il est impossible de contrôler la disponibilité des entités du contexte. Les applications ubiquitaires ont alors fortement besoin de s'adapter durant leur exécution et réagir aux changements que le contexte peut subir (ex. : déplacement d'un utilisateur, disponibilité d'une ressource, changements de besoins, ou pour faire face à un échec [CGSS02]). Bien que la sensibilité de ces applications aux contextes et à ses différents acteurs soit un domaine très étudié [KLTO12] [DeCH12] [Perc00], développer des applications sensibles aux contextes reste encore complexe.

Tout ceci introduit le défi d'avoir un environnement « intelligent ». Celui-ci doit être sensible aux changements du contexte, réactif aux informations collectées à travers de multiples capteurs et capable de filtrer et de traiter cet océan d'informations dans l'environnement afin de fournir une information correcte et précise aux utilisateurs. La possibilité de faire évoluer les applications par

l'auto-adaptation à l'exécution est un des critères sur lequel nous comparerons les systèmes étudiés dans notre état de l'art.

3.2. ABSTRACTION

3.2.1. DU CONTEXTE

La complexité d'un environnement ubiquitaire est telle que nous ne pouvons pas raisonner directement sur les entités qui le compose. Ces entités sont différentes par leurs technologies et leurs modes d'interactions. Ainsi, une des difficultés des travaux liés à l'environnement ubiquitaire est de trouver la « bonne abstraction » pour représenter le contexte. L'abstraction nous permet alors de comprendre et de raisonner sur le contexte. De plus, ces informations peuvent être utilisées pour construire des systèmes plus intelligents et autonomes.

Un « bon » niveau d'abstraction permet d'obtenir une description homogène des entités qui composent l'environnement d'exécution. Les travaux s'orientent vers des approches basées sur les modèles ou sur les ontologies [RMCM03]. Ces approches sont similaires et elles n'ont pas cessé de se rapprocher au cours du temps. Les approches basées sur les ontologies se concentrent essentiellement sur la taxonomie des objets du contexte, leurs propriétés et leurs relations. On a vu apparaître plusieurs langages qui permettent de décrire le contexte tels que RDF [Webc04], OIL et DAML+OIL [Daml00] ou encore OWL [McHa00]. Dans les approches basées sur les modèles, le but reste le même. Cependant, il s'agit alors d'une approche plus proche des langages de développement [SeKo03] [StVo06], ce qui rend sa mise en place plus simple qu'une approche à base d'ontologies. Il existe d'ailleurs différents standards pour ces approches tels qu'UML [Seli05] ou MDA [Sole00]. Dans [StLi04] on trouve une comparaison entre les différentes techniques de modélisation.

On s'appuie dans cette thèse sur des modèles (et méta-modèles) pour représenter le contexte d'exécution de l'application et de son environnement.

3.2.2. DE L'HÉTÉROGÉNÉITÉ

Les défis liés aux réseaux et aux différents protocoles ainsi que les conflits qu'ils peuvent engendrer sont très complexes dans l'informatique ubiquitaire. Masquer cette couche en permettant aux applications de communiquer entre elles et avec les dispositifs d'une manière transparente constitue un réel enjeu. Il existe d'ailleurs des techniques pour unifier la communication entre des entités d'un même réseau. Par exemple, les techniques de médiations [Wied07] [HéTL05] permettent de faire le lien entre deux entités distinctes par l'intermédiaire de médiateurs.

Une autre difficulté résulte de l'hétérogénéité des systèmes sur lesquels s'exécute l'application. En conséquence, une application peut (ou doit) être disponible en plusieurs versions. Chaque version pouvant être spécifique d'un environnement ou d'un équipement donné. L'hétérogénéité croissante des environnements complique la tâche de développement de l'application pour chacune de ces plates-formes [SaMu03]. Depuis peu, des sociétés proposent des solutions pour pallier à l'hétérogénéité des systèmes d'exploitation mobiles. Cependant, cela se limite à une surcouche graphique qui va s'adapter au bon format du mobile ; c'est le cas par exemple de BkRender[4] de Backelite.

Un autre type d'hétérogénéité résulte du besoin de faire communiquer des applications hétérogènes. Il s'agit le plus souvent d'un besoin de communication et de collaboration entre les applications utilisées dans plusieurs entreprises. Des logiciels d'intégration d'application d'entreprise (Entreprise Application Integration ou EAI) ont alors vu le jour [LeSH03]. Puis, avec l'avènement des services web, les logiciels d'intégration ont évolué vers des bus de service d'entreprise (Entreprise Service Web ou ESB) [Vino03]. Ces approches permettent à des applications hétérogènes de communiquer par des envois de messages. Malheureusement, l'intégration reste très délicate à mettre

[4] http://bkrender.com/

en œuvre. Le coût d'une telle installation est très élevé. Une fois le système mis en place, il est très difficile de le reconfigurer. Nous considérons que l'intégration ne tient pas compte des propriétés importantes liées à l'ubiquitaire telle que l'évolution. Pour finir, ces approches ne s'adressent que rarement à l'intégration des dispositifs physiques avec les systèmes.

Dans notre thèse, nous nous intéressons d'une part à masquer l'hétérogénéité relative à la découverte et à la communication avec les équipements et d'autre part à intégrer différentes plates-formes d'exécution afin d'enrichir le contexte d'exécution des applications. Nous allons alors baser notre état de l'art des systèmes existants vis-à-vis de l'hétérogénéité sur ces deux points. La capacité de déployer et d'accéder à des machines distantes d'une manière transparente est un des enjeux majeur de cette thèse.

3.3. LA RÉSILIENCE

Les environnements ubiquitaires doivent faire face à un monde ouvert, dans lequel la disponibilité des éléments du contexte est dynamique. C'est le cas pour les équipements mobiles, tels que les téléphones qui peuvent apparaître ou disparaître à tout moment. La résilience est définie comme la capacité d'une application à résister (ou même à profiter) aux changements imprévisibles et aux conditions environnementales qui pourraient causer une perte de service inacceptable [Flor11][Meye09].

C'est dans le domaine de la physique, au début du XX^{ème} siècle, que nous trouvons la première définition scientifique de la résilience. Elle désigne alors la résistance d'un métal et sa capacité à reprendre sa forme initiale à la suite d'un choc [Char05]. Par la suite, ce terme devient pluridisciplinaire, on le retrouve par exemple en écologie [Holl73], en psychiatrie [Cyru99] et en informatique [CLLC05][Lapr08].

3.3.1. EMPLACEMENT DES COMPOSANTS

Dans les systèmes traditionnels, les applications sont auto contenues, l'assemblage des entités qui constitue l'application est réalisé à la compilation. Le résultat est souvent une application statique. L'application ne prend pas en considération son environnement d'exécution. L'édition de lien statique souvent utilisée par les compilateurs C par exemple, intervient dans la dernière phase de compilation d'un programme C et résout les références externes du code pour produire un seul fichier exécutable. Ces applications ne prennent pas en considération l'environnement à l'exécution.

Les chercheurs écologistes James Lovelock [Love00], C.S. Holling [Holl73] ou [Tilm96] ont montré l'importance de la biodiversité pour la résilience [Wiki00a] :

> ***« La diversité présente dans un milieu est le gage d'un meilleur auto entretien de l'écosystème. »***

Cette même observation peut être faite sur les systèmes informatiques : plus le système est capable de trouver des ressources alternatives (au-delà de son propre écosystème), plus il sera capable d'offrir de nouvelles solutions pour satisfaire à des besoins imprévus.

En informatique et plus particulièrement dans un environnement ubiquitaire, l'application doit continuer à évoluer à l'exécution. Elle doit alors être sensible au contexte qui l'entoure. Le choix des composants entrant dans la composition de l'application est alors guidé par la disponibilité des entités de l'environnement. Ce dernier pouvant être hétérogène et complexe, il peut être découpé en plusieurs emplacements. Chaque emplacement contient les entités (composants) nécessaires à une application pendant son exécution.

Les emplacements des composants peuvent être de plusieurs types. Pour une application auto contenue, il n'existe qu'un emplacement composé uniquement des entités qu'elle contient. La plate-forme d'exécution locale, les dépôts de code, les plates-formes distantes, etc., représentent d'autres emplacements. Ces emplacements peuvent être hétérogènes et différents mais les applications

ubiquitaires doivent prendre en considération les entités présentes dans ces emplacements pour profiter d'un large choix de composants pendant leur exécution.

3.3.2. STRATÉGIE D'ÉCHEC

En informatique, dans le domaine de « *Dependable Computing* » la résilience est souvent associée à la tolérance aux fautes [ALRL04][Lapr08]. De façon générale ce terme exprime la capacité d'un système à continuer à fonctionner, d'une manière correcte, face à des changements qui pourraient provoquer l'échec de l'application. Ainsi, dans [Lapr08], la résilience est définie comme :

> ***« The persistence of service delivery that can justifiably be trusted, when facing change. »***

La résilience a pour but de donner à l'application un degré d'autonomie pour faire face à des changements de contexte. Contrairement aux approches autonomiques [Horn01], nous ne cherchons pas dans cette thèse à rendre une application complétement autonome. Cependant, ce que nous proposons peut servir de base pour la réalisation de systèmes autonomiques plus complexes.

Dans cette thèse, nous nous intéressons à la résilience des applications face à un contexte hétérogène et dynamique. La résilience définit alors la capacité des applications à faire face aux changements dynamiques qui se produisent dans le contexte d'exécution de l'application. Nous étudierons dans l'état de l'art les politiques que les différents systèmes mettent en place pour offrir plus de résilience à leurs applications ainsi que les stratégies à adopter pour faire face à un changement imprévu du contexte.

4. SYNTHÈSE

La difficulté principale des systèmes ubiquitaires est de répondre simultanément aux divers défis identifiés ci-dessus. Dans la suite de la thèse nous nous intéresserons aux architectures logicielles et aux systèmes modulaires qui offrent des solutions aux défis de l'informatique ubiquitaire.

CHAPITRE 3
COMPOSANT, SERVICES ET ARCHITECTURE

1. INTRODUCTION

Le défi de l'ingénierie du logiciel est de réaliser et d'exécuter des systèmes complexes. Les environnements et les applications ubiquitaires sont des systèmes complexes. Nous proposons d'appliquer les pratiques et solutions acquises par l'ingénierie des logiciels aux environnements ubiquitaires.

Dans ce chapitre, nous présenterons brièvement les concepts pour les approches à composants, à services et le rôle qu'ils peuvent jouer dans les systèmes complexes tels que les environnements ubiquitaires. Ensuite nous aborderons les approches qui combinent les composants et les services pour tirer profit des bénéfices de chacune de ces approches. Puis, nous proposerons un état de l'art sur la notion d'architecture logicielle et le rôle qu'elle joue dans la conception et réalisation de logiciels complexes.

Comme conclusion de ce chapitre, nous comparerons les solutions qui se basent sur les différents concepts introduits dans cette partie (composant, service, architecture, etc.) et qui peuvent répondre aux défis de l'informatique ubiquitaire.

2. APPROCHES COMPOSANT

2.1. DÉFINITION ET CONCEPTS DES APPROCHES À COMPOSANT TRADITIONNEL

Les approches à composant (CBSE) visent à faciliter la réutilisation du code des applications logicielles par la décomposition des systèmes en briques logicielles fonctionnelles appelées composant [Szyp02]. On retrouve les premières références au concept de composant en 1968 dans [Mcil69], le composant était alors considéré comme la solution à la crise du logiciel. C'est dans les années 90 qu'émerge l'idée de séparer l'architecture de l'application (la composition) et les composants de base, permettant à chacun d'évoluer indépendamment. Pendant ce temps la définition du composant n'a pas cessé d'évoluer.

Une définition très simple du composant est présentée dans [NiDa95] :

> ***« A component is a static abstraction with plugs. »***

À partir de cette définition nous pouvons extraire trois termes importants :

- **Statique**, les auteurs voulaient exprimer le fait qu'un composant logiciel est une entité avec une longue durée de vie capable d'être sauvegardé dans un dépôt logiciel et cela indépendamment des applications dans lesquelles elle est utilisée.
- **Abstraction**, pour signifier qu'un composant met une frontière plus ou moins opaque autour du logiciel qu'il encapsule**.**
- **« Plugs »**, pour signifier que les moyens d'interagir et de communiquer avec le composant sont bien définis (comme les messages, les ports ou les contrats). Par conséquent, il devient réutilisable.

Une autre définition, largement acceptée par la communauté est décrite par Szyperski dans [Szyp02] :

> ***« 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. »***

L'auteur présente alors le composant comme étant une unité de composition qui expose des interfaces fonctionnelles et décrit des dépendances, ce qui lui permet d'être composé par des tiers. La cohérence d'une composition obéit à des règles pour la description et la conception du composant. Dans [HeCo01], Heineman propose la définition suivante :

> ***« 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. »***

D'après la définition ci-dessus, un composant doit être conforme à un modèle à composants. Le rôle de ce modèle a été aussi décrit par Heineman dans [HeCo01] comme :

> ***« 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. »***

Le modèle à composants définit donc l'infrastructure et le cadre nécessaire permettant la conception, le développement, le déploiement et l'exécution des composants. Pour avoir une idée sur le contenu d'un modèle a composant, il faut se tourner vers la définition de Bachmann dans [BBBC00] :

> ***« A component model is the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types. »***

D'après les différentes définitions citées précédemment, le modèle à composants contient les types de composant qui représentent la forme conceptuelle du composant. À partir de ce type il est possible de créer des unités d'exécution appelé instances de composant (ou tout simplement instances).

2.2. APPROCHE A COMPOSANTS ORIENTÉS SERVICE

L'approche à composant a évolué suite à l'apparition des approches à service. Cette nouvelle tendance est appelée approche à composants orientés service [Cerv04]. L'approche à services (Service-Oriented Computing [Papa03a] ou tout simplement « SOC »), est un paradigme 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. Nous mettons en évidence la complémentarité entre les deux approches. Enfin, nous allons extraire les nouvelles caractéristiques apportées par cette approche pour répondre aux défis du domaine de l'ubiquitaire.

2.2.1. DÉFINITION DU SERVICE

La notion de service est le concept de base de l'approche à service. Elle représente l'unité de construction pour les applications produites par cette approche. A l'instar des composants, les services permettent de construire et d'encapsuler des applications modulaires. Néanmoins, le service a un sens plus large que le composant : le service peut être soit un concept du monde réel (exemple : un service de restauration), soit un élément concret de l'application (exemple : un spooler d'impression). Ceci rend ambigu la signification du terme service et donne lieu à une multitude de définitions dans la littérature. Cependant, parmi les définitions qui sont fréquemment utilisées, on trouve celle données par Papazoglou [Papa03a] :

> ***« Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything from simple requests to complicated business processes. »***

Cette définition met en évidence la « description de service ». Cette description est indépendante de la plate-forme d'exécution, elle ne connaît pas le contexte dans lequel le service sera utilisé. Un service est donc utilisé sans connaître ni les détails de son contenu ni la technologie utilisée pour son implémentation. Cette indépendance introduit la plus importante propriété du SOC : le faible couplage entre le fournisseur et le consommateur de service. De façon générale [PaHe07] :

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

Cette définition aborde les principes d'interaction avec le service. La définition donnée dans [Arsa04] insiste sur le fonctionnement et les interactions à travers la description du service :

> ***« A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. 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. »***

Le service possède donc une description rendue accessible par les fournisseurs de service. Le consommateur recherche le service en utilisant la description. Une fois le service trouvé, le consommateur peut se lier au fournisseur de ce service et peut invoquer les fonctionnalités décrites dans la description de ce service. On notera aussi qu'un service possède un ensemble de politiques (ou propriétés) exprimées de façon déclarative. Elles permettent ainsi la reconfiguration dynamique du service.

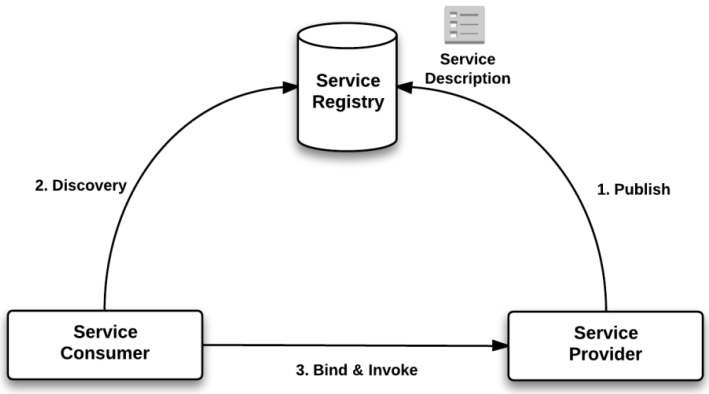


Figure 4 : Acteurs & principes d'interactions du SOC

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

> ***« Un service est une entité logicielle décrite à travers une spécification. 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'ensemble des fonctionnalités définies dans la spécification du service sont implémentés par le fournisseur. Une spécification de service est utilisée par les consommateurs pour rechercher un fournisseur qui correspond à ses besoins, le sélectionner et l'invoquer à travers sa spécification. »***

La définition précédente contient la majorité des principes d'interaction nécessaires pour construire des applications basées sur le SOC. Ces principes sont détaillés dans la partie suivante.

2.2.2. MÉCANISMES DU SOC

L'approche à service peut être vue comme un style architectural [ShGa96], qui utilise les services comme entités de base. Ce style permet de construire des applications en utilisant des entités faiblement couplées. Les différents acteurs qui participent à la mise en place d'une approche à service sont illustrés dans la Figure 4. Ces acteurs sont au nombre de trois :

- **Fournisseurs de services :** offrent les fonctionnalités des services décrits dans des spécifications de services.
- **Consommateurs de services** : requièrent et utilisent les fonctions des services offerts par les fournisseurs à travers la spécification de service.
- **Annuaire de services :** appelé aussi **registre de services**, fournit un ensemble de mécanismes permettant la publication, la recherche, la découverte et la sélection de services.

Le modèle d'interaction repose sur trois primitives :

1. **Publication de services :** les fournisseurs publient la spécification des services dans l'annuaire de services.
2. **Recherche et découverte :** Les consommateurs interrogent l'annuaire de services pour rechercher et découvrir les services qui correspondent à leurs besoins.
3. **Liaison entre un consommateur et un fournisseur de service :** le consommateur choisit un fournisseur de service, après la réalisation de la liaison, il est capable de l'invoquer afin d'utiliser les fonctionnalités fournies.

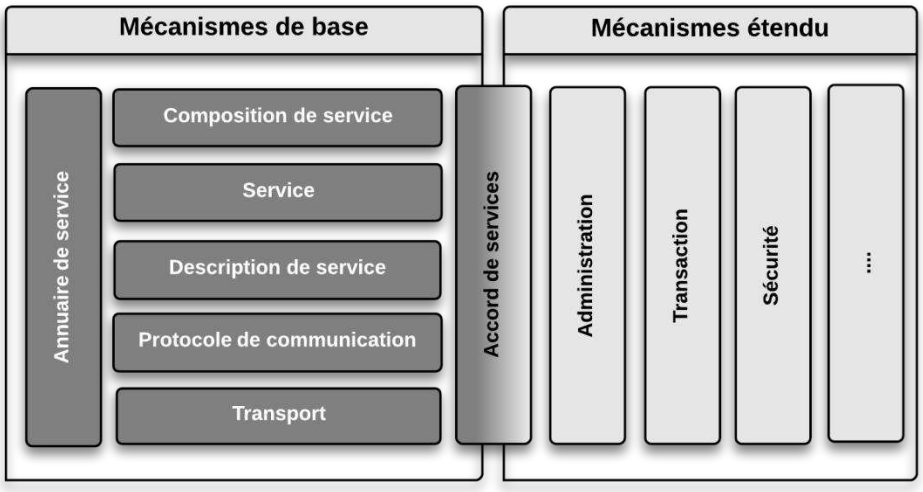


Figure 5 : Mécanismes du SOA [EAAC04]

L'annuaire de service introduit une couche d'indirection qui permet d'augmenter le découplage entre consommateurs et fournisseurs. Cette architecture permet de renforcer les caractéristiques importantes employées par l'approche à service :

- Le **faible couplage :** la spécification du service est la seule information qui relie le fournisseur et le consommateur.
- La **liaison tardive :** la liaison entre le consommateur et le fournisseur du service est réalisée uniquement à l'exécution et après que le service soit recherché et sélectionné depuis le registre de service.
- Le **chargement paresseux :** c'est un comportement évolué de la liaison tardive qui permet de charger les services seulement au dernier moment, lors de l'invocation du service.
- Le **masquage de la distribution** : avant l'exécution, la localisation du registre n'est pas connue et le consommateur ignore la localisation des services.

- **Le masquage de l'hétérogénéité :** les consommateurs n'ont pas besoin de connaître les détails de l'implémentation des fournisseurs et vice versa.

Dans cette thèse, nous nous intéressons à ses différentes propriétés. Elles représentent une solution aux besoins d'adaptation et d'évolution dynamiques des environnements ubiquitaires.

2.2.3. SOA : SERVICE ORIENTÉ ARCHITECTURE

Une architecture à service (SOA) est un environnement d'intégration et d'exécution des services. Le SOA permet de concevoir des systèmes logiciels capables de fournir des services aussi bien pour des applications pour utilisateur finaux que pour des autres services distribués dans un réseau [PTDL08].

Le SOA fournit en plus des mécanismes permettant la publication, la recherche, la découverte, la liaison et l'invocation de service. Le SOA fournit souvent des mécanismes additionnels pour supporter certains aspects non fonctionnels tels que la sécurité, la distribution, les transactions, ou l'administration (voir Figure 5). En effet, c'est pour adresser les problématiques de composition et d'administration nécessaires pour la création d'application flexibles que Papazoglou a proposé d'étendre le SOA [Papa03a]. Grâce à cette extension il est possible de coordonner des services, de gérer leur composition et leur conformité. Le service composite résultant peut être ensuite utilisé comme n'importe quel autre service « simple ».

2.2.4. LE DYNAMISME DANS LE SOC

Afin de supporter le dynamisme, deux nouvelles primitives ont été ajoutées au principe d'interaction de base (voir Figure 6). On parle alors d'approche à service dynamique [EsHL07]. Ces primitives sont :

- **Le retrait de services :** permet de signaler qu'un fournisseur de service n'est plus en mesure d'offrir son service.
- **La notification :** permet d'informer les consommateurs de l'arrivée ou du départ d'un fournisseur de service.

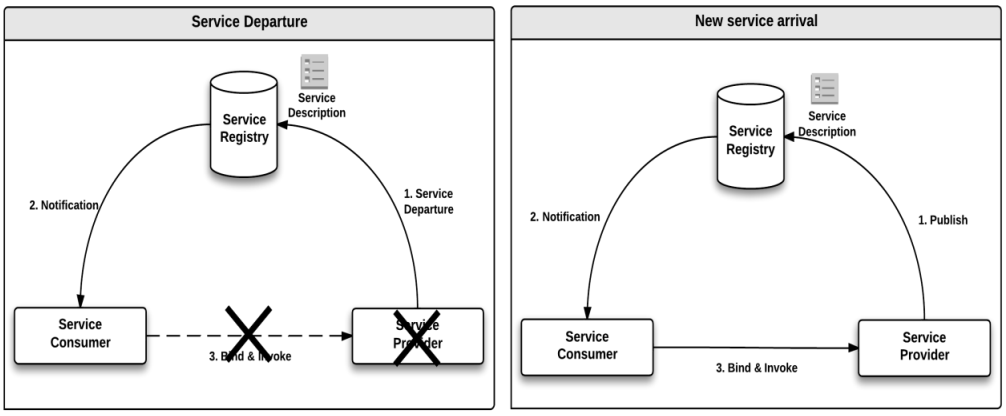


Figure 6 : Principe de retrait et d'ajout de service

Ces primitives sont intéressantes pour l'ubiquitaire car elles rentrent en jeux pendant l'exécution de l'application. Elles permettent au consommateur de choisir lors de l'exécution son fournisseur de service et d'en changer. L'application peut donc évoluer dynamiquement à l'exécution et peut donc faire face à des changements dans le contexte d'exécution.

2.2.5. LE MARIAGE ENTRE COMPOSANTS ET SERVICES

L'approche à services dynamique a donné naissance à un nouveau type de modèle à composants alliant 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. On parle alors de **composant orienté service (ou composant à service [Cerv04]).**

On retrouve dans les composants orientés services les concepts issus des deux approches, le composant permettant de décrire et d'encapsuler le code fonctionnel et les services qui représentent les entités permettant aux composants de s'interconnecter suivant la logique des services (publication, recherche, sélection, liaison, invocation, etc.).

Les principes de cette approche, décrits dans [CeHa04] sont :

- **Un service est une fonctionnalité fournie.** Un service est un ensemble d'opérations réutilisables.
- **Un service est décrit par une spécification de service** qui peut contenir des informations syntaxiques, comportementales et sémantiques, ainsi que des dépendances vers d'autres spécifications de services.
- **Un composant implémente une ou plusieurs spécifications de service** en respectant leurs contraintes. Les services sont ainsi le seul moyen d'interaction entre les instances de composants.
- **Les mécanismes d'interaction de l'approche à services sont utilisés pour résoudre les dépendances des composants**. Les services, fournis par des instances de composants, sont enregistrés auprès d'un registre de services. Ce registre 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 une spécification de services qui permet de sélectionner les composants concrets. Les liaisons ne sont pas déclarées de manière explicite : elles sont inférées à partir des dépendances de services.
- **Les spécifications de services sont la base de la substitution.** Dans une composition, un composant peut être remplacé par un composant alternatif qui respecte la même spécification.

Le modèle à composants orienté 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 (et non pas en termes d'implémentations comme dans les modèles à composants traditionnels). Grâce aux principes d'interactions hérités de l'approche à services, la résolution de dépendances permettant la sélection des implémentations (i.e., des fournisseurs de services), peut être retardée jusqu'à l'exécution (contrairement aux modèles à composants traditionnels qui effectuent la sélection avant l'exécution).

2.3. LES APPROCHES À COMPOSANTS ET LES DÉFIS DE L'UBIQUITAIRE

L'approche à composant est née bien avant l'émergence des environnements ubiquitaires. Elle aborde principalement le développement des composants. L'exécution et l'adaptation des applications qui sont des aspects importants pour l'ubiquitaire ne sont pas abordées (à l'origine) par l'approche. Les composants orientés services viennent combler ce manque et répondent à la majorité des défis de l'informatique ubiquitaire. Ainsi, par leur nature dynamique, ils peuvent faire face aux changements de contexte qui peuvent se produire à l'exécution de l'application.

Abstraction C'est un défi largement abordé par les approches à composant et à service. Il introduit plusieurs caractéristiques et propriétés :

- **Les niveaux d'abstractions :** le concept de composant ou de service est intéressant pour les environnements ubiquitaires car il permet de conceptualiser des systèmes complexes. Il introduit les types de composant pour mieux séparer la conceptualisation et le développement du système. Nous avons alors deux niveaux d'abstraction (type et instance) [JoWo96]. Le service permet d'introduire un nouveau niveau d'abstraction au-dessus du type et indépendant de l'implémentation.

- **Assemblage :** la composition est statique, elle est réalisée et validée avant l'exécution de l'application. Ceci offre une garantie sur le contenu du système car les liaisons entre les différents composants sont décrites explicitement.
- **Cohérence des descriptions :** les descriptions des composants et des dépendances sont vérifiées par les environnements au développement, ce qui limite les risques d'erreur à l'exécution.

Évolutivité L'approche à composant classique vise à faciliter la construction des applications par réutilisation des composants, l'évolution de l'application passe souvent par le redéveloppement. Les travaux réalisés autours de l'approche à composant orienté service essaient d'introduire plus de flexibilités en mettant en avant les propriétés suivantes :

- **Dynamisme :** l'approche à composant classique permet la création et destruction d'instances de types de composants et des connecteurs connus. L'introduction des services a permis de traiter le départ inopiné d'un composant (ou d'un service) qui n'est pas traitée par l'approche à composant traditionnel.
- **Adaptation :** le fait que le couplage entre les composants et souvent la description explicite des liaisons soient réalisés à la conception rend l'adaptation très limitée voire impossible. La plupart du temps, en cas d'évolution du contexte d'exécution, l'adaptation ne peut être réalisée qu'en redéveloppant le logiciel. Mais grâce aux services, il est possible de retarder la sélection des composant jusqu'à l'exécution, de réaliser la reconfiguration des instances de composant et finalement d'établir des liaisons tardives.

Résilience La résilience est aussi influencée par le degré de flexibilité de l'application. Plus l'application est adaptable plus il est possible de la rendre résiliente. Les systèmes réalisés par les approches à composants classique sont auto-contenus. Le contexte ne participe pas aux décisions d'assemblage, l'exécution est donc statique. Cependant, l'approche décrit l'utilité des dépôts de code pour le stockage et la réutilisation des composants. Mais ces dépôts sont uniquement utilisés pendant le développement de l'application. L'approche à service introduit un nouvel emplacement : il s'agit du « registre ». Grâce à lui, nous pouvons référencer les composants pour qu'ils soient accessibles aux autres applications.

Le tableau suivant est complété pour chacune des approches, il présente les potentiels de chaque approche de façon synthétique. Chaque tableau synthétique sera détaillé dans la dernière section pour les différents travaux existants présentés dans cet état de l'art.

Nous utiliserons les phases de la lune pour exprimer le potentiel d'une approche vis-à-vis d'un défi. Les phases vont de la nouvelle lune ● à la pleine lune ○. La nouvelle lune signifie que le potentiel de l'approche est capable de répondre au défi et la pleine lune signifie que le défi n'est pas supporté.

Abstraction	Évolutivité	Résilience
◐	◐	◐

Tableau 1 : Approche composant orientée service et les défis de l'ubiquitaire

2.4. TRAVAUX EXISTANTS

Dans cette section, nous présenterons brièvement des modèles à composants possédant des propriétés permettant de répondre aux défis de construire des applications ubiquitaires. La plupart de ces technologies en sont capable, cependant la complexité et l'effort nécessaire pour la mise en œuvre de telles applications peuvent vite devenir un frein à leur utilisation. Dans la dernière partie de ce chapitre, nous présenterons le résultat de la comparaison de ces modèles selon une liste de caractéristiques

obtenues à partir de l'analyse des défis des environnements ubiquitaires. Cette comparaison sera résumée dans les tableaux du dernier chapitre.

2.4.1. FRACTAL

Fractal [BCLQ06] est l'un des modèles a composant générique et extensible le plus utilisé dans la recherche. Il constitue la base de plusieurs travaux [LeQS04][DaLe05][BPHT06][RHTN10]. Il existe différentes implémentations de Fractal dans plusieurs langages de programmation : C (Think[FSLM02], Cecilia [Ow00a]), C++ (Plasma), Java (Julia [BCLQ06], AOKell [SPDC06], ProActive), Smalltalk (FracTalk) et .NET (FractNet). Chacun de ces langages apporte des propriétés supplémentaires à ce modèle et chacun est destiné à un domaine particulier. Par exemple son implémentation C (appelé Think) est destinée aux développements des applications embarquées et l'implémentation en Java (appelée ProActive) est une implémentation pour les grilles informatiques (Grid Computing). Cependant, c'est Julia (une implémentation Java) qui est considérée comme l'implémentation de référence de Fractal. Elle permet la spécification de composants en Java et permet leur manipulation à l'exécution.

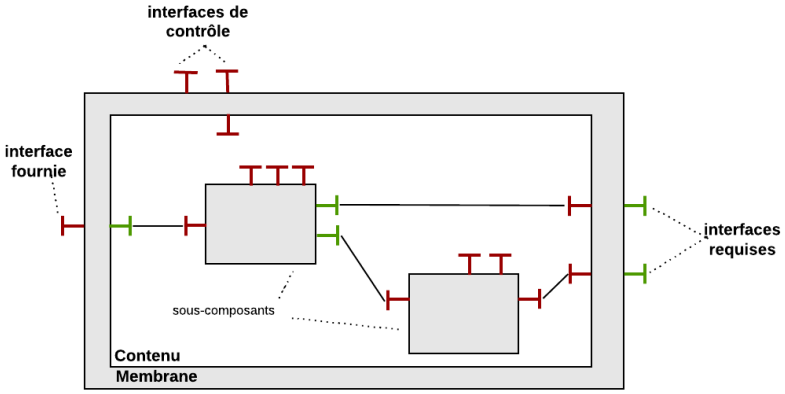


Figure 7 : Modèle à composant Fractal

On retrouve dans Fractal les concepts classiques des modèles à composants traditionnels (composant, interfaces, connexion). L'originalité de Fractal réside en la séparation du composant en deux entités :

- **Le contenue (Content) :** contient un ensemble fini de (sous) composants et de connexions.
- **La membrane :** définit des interfaces de contrôle pour réaliser l'introspection et la manipulation du composant. Elle définit aussi des interfaces de connexion pour exprimer les interfaces fournies et requises par le composant.

Fractal permet la construction d'applications hiérarchiques mais ne définit pas de mécanisme particulier pour le packaging ou le déploiement de composants car celui-ci dépend de l'implémentation choisie. Aussi, il offre, via son API, des mécanismes pour faire évoluer dynamiquement les applications : créer et détruire des instances et des liaisons.

2.4.2. EJB

La technologie Enterprise Java Beans (EJB) initialement développée par Sun MicroSystems [Gham97][MaHa99] continue à évoluer au sein de l'entreprise Oracle [Orac00a]. Elle propose un modèle à composants pour la construction d'applications métier distribués. La technologie est directement compatible avec les programmes et services Java existants (transactions, persistance, concurrence, sécurité). EJB est une technologie standardisée. La spécification de ses interfaces de programmation est publique [Orac00b]. Plusieurs entreprises et communautés open source proposent des implémentations concurrentes et interopérables de cette spécification (Glassfish, JBoss, Websphere, Jonas, Apache Tomcat). Ceci fait que les EJB sont largement utilisées pour la mise en place des serveurs d'applications et en

particulier ceux destinés au commerce électronique. Le succès des EJB est aussi dû à la présence d'outils de conception graphique et au support des sociétés Sun Microsystems et Oracle.

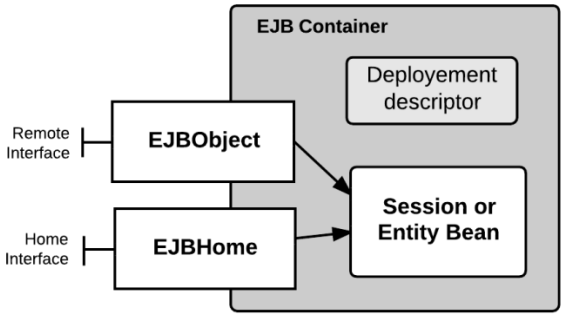


Figure 8 : Modèle à composant EJB

Le composant EJB s'appuie sur le concept de conteneur pour intercepter les appels vers les instances de composant. Ce conteneur fournit des services pour gérer le cycle de vie, les transactions, la persistance, etc. Il existe trois types de composants EJB (appelés aussi beans) :

- **Session (SessionBeans)**, réalise la communication synchrone avec les utilisateurs. Elle peut être avec ou sans état :
  - **Sans état (Stateless),** utilisée pour communiquer simultanément avec plusieurs clients.
  - **Avec état (Stateful),** elle est associée à un seul client et permet de conserver un état entre les différents appels. Les composants de ce type ne sont pas partageables.
- **Entité (EntityBeans),** représente les entités persistantes stockées dans une base de données.
- **Message (MessageDrivenBeans),** réalise la communication asynchrone par événements ou point à point.

Un composant EJB ne permet pas de décrire explicitement les interfaces requises d'un composant. L'implémentation du composant est réalisée par une seule classe Java. Elle implémente une interface spécifique aux EJB dite *« remote »* et cette interface permet de spécifier les méthodes qui contiendront les traitements proposés par le Bean. Les composants EJB sont déployés sous la forme d'archives « *jar* », chaque archive contenant toutes les classes qui composent un EJB (interfaces, classes qui implantent ces interfaces (en Java) et toutes les autres classes nécessaires aux EJB).

EJB ne permet pas de spécifier les interfaces requises ou la composition entre les composants, la composition de composants EJB est faite de manière impérative. De plus, EJB ne fournit aucun outil ou méthodologie pour assembler les implémentations ou les instances de composants. La gestion des connexions reste à la charge des développeurs. La pauvreté du langage de composition fragilise donc la construction des applications à base de composants EJB.

2.4.3. K-COMPONENT

K-Component est un modèle à composants qui s'est attaqué directement à la complexité des applications auto-adaptative [DoCa01][DoCC01]. Ce modèle a été développé au-dessus de CORBA [Vino97]. Ce dernier est d'ailleurs utilisé pour l'implémentation des composants. L'originalité de cette approche vient du fait que chaque composant dispose d'une vue partielle du système. Il permet à chaque composant de réaliser des adaptations individuelles à travers une coordination complexe et décentralisée.

L'évolution des applications réalisées à travers ce modèle à composants est dynamique : elle est réalisée pendant l'exécution de l'application. En effet, K-component définit des composants de haut niveau, appelés « *gestionnaires d'adaptation* », capables d'observer, raisonner et reconfigurer l'architecture réifiée (en termes de composants et de connecteurs) de l'application en exécution. La reconfiguration est réalisée par l'ajout, la suppression ou la mise à jour des composants et connecteurs.

Les règles d'adaptations sont décrites dans des contrats d'adaptation qui sont déclenchés par les événements émis par l'environnement ou par l'application en exécution.

Parmi les autres avantages de ce modèle à composants on peut citer le chargement et le déchargement dynamique de contrats d'adaptation. Cependant, ce modèle est incapable d'ajouter dynamiquement de nouveaux types de composants.

2.4.4. iPOJO

La technologie Apache iPOJO [EsHa07][EsHL07], acronyme de « injected Plain Old Java Object » est un modèle à composants orienté services développé au sein de l'équipe Adèle du Laboratoire d'Informatique de Grenoble (LIG). Actuellement, elle est un sous-projet du projet Apache Felix [Apac00a], qui est lui-même une implantation open source de la spécification OSGi™ [Osgi00a]. iPOJO est considéré comme le successeur de Service Binder [CeHa04].

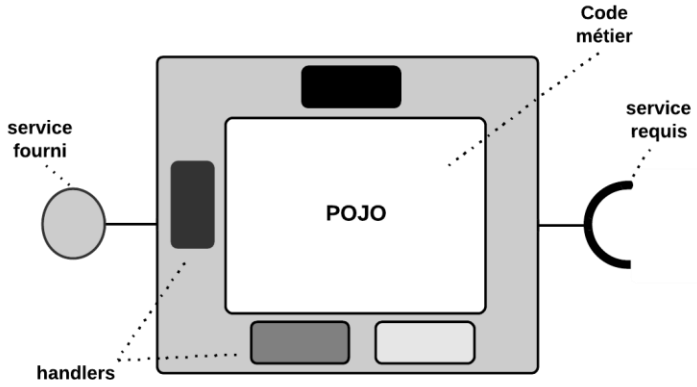


Figure 9 : Modèle à composants iPOJO

L'objectif principal d'iPOJO est de fournir une plate-forme d'exécution pour faciliter le développement d'applications basé sur le principe de l'architecture à services dynamiques dans OSGi™. Pour cela il utilise la notion de conteneur pour gérer les interactions entre les composants et avec la plate-forme. Le conteneur d'un composant iPOJO prend en charge :

- **Les aspects techniques**, comme l'enregistrement de services dans l'annuaire, la découverte et la sélection de services et la gestion des dépendances d'un composant.
- **Les propriétés non-fonctionnelles,** des composants iPOJO. Un conteneur iPOJO est composé de plusieurs *« handlers »* chacun d'eux prend en compte un aspect non fonctionnel du composant. Il est possible dans iPOJO d'ajouter de nouveaux handlers au conteneur d'un composant pour couvrir d'autres aspects non fonctionnels non pris en compte par les handlers fournis par défaut.

Grâce à ces mécanismes, iPOJO permet de construire plus facilement qu'auparavant des applications à services dynamiques tout en respectant les principes de l'architecture à services dynamiques. La simplification de la gestion du dynamisme est un apport très important qui aide considérablement le développeur. De plus, la possibilité de gérer uniquement les propriétés non fonctionnelles que souhaite l'utilisateur est un atout. Ceci fait que la technologie iPOJO est aujourd'hui utilisée dans de nombreux produits, comme JOnAS [Ow00b] ou Cilia [Garz12].

2.4.5. KEVOREE

Kevoree [BDBM00] est un modèle à composants open source récent développé par l'INRIA-IRSA Rennes au sein de l'équipe Triskell. Ce modèle vise à gérer des applications adaptatives et distribuées par l'utilisation des modèles à l'exécution (Model@Runtime) [MBJF09]. Ce modèle se distingue des modèles à composants traditionnels par la définition du concept de « canaux de communication (channels) ». Ces

canaux permettent de relier des composants distants déployés dans des plates-formes hétérogènes, selon une sémantique définie pour chaque canal.

Pour gérer l'hétérogénéité des plates-formes de reconfiguration et la dissémination des politiques de reconfiguration [FDPB12a][FMFB12], Kevoree définit de nouveaux concepts :

- **Nœud :** permet de modéliser la topologie de l'infrastructure distribuée. Les nœuds sont organisés hiérarchiquement et chaque nœud contient un ensemble de composants et de canaux de communication. Plusieurs type de nœuds sont déjà définis et permettent d'intégrer différentes plates-formes tels que Java, Android, MiniCloud, FreeBSD et Arduino.
- **Groupe :** permet de synchroniser la représentation d'un modèle d'une application en exécution pour un ensemble de nœuds. Chaque groupe définit une sémantique de communication, ce qui lui permet de disséminer le model@Runtime à l'ensemble des nœuds.

Kevoree est capable à la fois de modéliser l'architecture d'un système distribué hétérogène, tels que les systèmes ubiquitaires et de gérer des évolutions au cours de leur fonctionnement. Il vise à faciliter le développement d'applications dans le contexte de l'informatique ubiquitaire distribuée [FDPB12b].

3. ARCHITECTURES LOGICIELLES

Les modèles à composants traditionnels et ceux orientés service utilisent le concept d'architecture logicielle pour décrire la composition des applications. Ces architectures peuvent avoir différents niveaux d'abstraction et jouer différents rôles selon le cycle de vie et l'expertise que l'on souhaite réaliser sur l'application [CGBS02]. Ainsi, une architecture peut être aussi bien utilisée à la conception pour créer (par génération ou interprétation) les instances de composants correspondantes et leurs connecteurs, ou pour créer ou sélectionner à l'exécution les instances de composants et les lier.

3.1. DÉFINITIONS

En général, dans l'ingénierie des logiciels, la conception des systèmes se base sur la notion d'architecture logicielle. Cette notion décrit l'ensemble des entités logicielles qui composent le système ainsi que les différentes interactions existant entre ces entités [GaSh93].

Il existe plusieurs définitions du concept d'architecture logicielle. Il existe même un standard (IEEE 1471-2000) établi par un groupe de travail au sein de la communauté de l'ingénierie logicielle. Ce groupe définit l'architecture comme [JeLe00] :

> ***« 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. »***

La plupart des autres définitions trouvées dans la littérature nous éclaire un peu plus sur ce concept. Garlan et Shaw présentent le concept comme le moyen d'augmenter le niveau d'abstraction d'un système. Ceci facilite la compréhension du système lors de sa conception, de sa réalisation et de son évolution [GaSh93]. Ils définissent l'architecture dans [ShGa96] comme :

> ***« 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. »***

L'architecture logicielle est souvent définie comme la structure de haut niveau d'un système. D'autres définitions enrichissent cette structure par des propriétés permettant de mieux caractériser la structure. Ainsi dans [BaCK12], on nous propose de décrire l'architecture logicielle comme :

> ***« [...] the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. »***

Enfin, comme décrite dans [TaMD09], l'architecture peut aussi être vue comme l'ensemble de décisions capitales prises pendant la conception du système :

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

Afin de mettre en évidence la convergence de ces définitions, *« The Software Engineering Institute »* propose de recenser dans son site web [Soft00] les différentes définitions que l'on retrouve dans la littérature et chez les professionnels. Cette liste atteint plus de 90 définitions de la notion d'architecture logicielle. La multiplication des définitions est due en partie aux différents rôles que l'architecture logicielle peut endosser. Ainsi elle peut être :

- **Prospective :** en représentant l'intention (ou le but) d'un système.

- **Descriptive :** en matérialisant le système en exécution.

L'architecture est donc capable de définir le système mais aussi de le refléter durant son exécution.

3.2. CONCEPTS DE BASE D'UNE ARCHITECTURE LOGICIELLE

Les différents acteurs ne se sont pas mis d'accord sur une définition universelle de l'architecture logicielle. Malgré tout nous retrouvons dans les différentes définitions les mêmes concepts [MeTa00] :

- **Composant :** c'est l'unité d'assemblage de l'application. Chaque composant réalise une fonctionnalité spécifique. Un composant peut prendre différentes formes : type, implémentation ou instance. Les interfaces que décrit le composant (fournies ou requises) sont ses points d'accès et de communication avec l'extérieur.
- **Connecteur :** représente l'interaction entre les composants de l'architecture. Il contient les informations concernant les règles d'interactions entre les composants (exemple : les protocoles de communication entre les composants).
- **Configuration :** modélise la structure du système et décrit l'ensemble des composants et des connecteurs nécessaires pour le fonctionnement de ce système.

La description du composant et la description de l'architecture logicielle sont indépendantes. De nombreux langages de description de l'architecture (Architecture Description Langage : ADL) sont apparus pour faciliter la description, l'évaluation et la conception d'architecture logicielle. Dans [MeTa00], le langage d'architecture est définit 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. »***

3.3. ARCHITECTURES LOGICIELLES DYNAMIQUES

Pour gérer le dynamisme et l'adaptation des applications et des environnements, l'architecture logicielle a évolué pour devenir dynamique. Cette évolution a permis de décrire les degrés de variabilité des applications à l'exécution. Ainsi, une architecture logicielle dynamique peut être sensible au contexte d'exécution et réaliser des opérations telles que l'ajout, le retrait et la modification des composants et des connecteurs de l'architecture.

Dans [BHTV04], l'architecture dynamique est définie comme :

> ***« 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. »***

L'architecture dynamique permet de décrire les deux facettes du logiciel :

- **La partie fixe :** c'est la partie statique du logiciel qui n'a aucun besoin d'adaptation à l'exécution.
- **La partie variable :** c'est la partie qui peut s'adapter dynamiquement durant l'exécution de l'application. Cette partie peut évoluer grâce à la modification de la structure et de la définition de l'application. Nous pouvons par exemple ajouter ou modifier un type de composant lors de l'exécution.

3.4. LES ARCHITECTURES ET LES DÉFIS DE L'UBIQUITAIRE

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 :

Abstraction Les architectures permettent d'exprimer l'application avec de multiples niveaux d'abstraction. En effet, elle peut offrir plusieurs facettes (vues) de l'architecture aux différents acteurs qui participent à la réalisation de l'application. La description de l'architecture est souvent associée à des outils permettant de vérifier la cohérence et de la valider statiquement avant son exécution, ou même de générer le code nécessaire à son exécution.

Évolutivité L'architecture logicielle est capable d'exprimer le caractère évolutif des systèmes pour leur permettre de s'adapter à de nouveaux besoins. Ainsi, dans une architecture dynamique, nous sommes capables d'ajouter de nouveaux composants ou fonctionnalités, de substituer ou modifier un composant existant dans le but de s'adapter à l'environnement d'exécution de l'application [BMDL08][GaSc09]. Il existe plusieurs raisons pour lesquelles l'évolution à l'exécution peut devenir complexe, l'expression même de cette architecture pouvant limiter cette évolution. Ainsi, une architecture où tous les composants sont définis statiquement est plus difficilement manipulable à l'exécution qu'une architecture décrite de manière implicite. La gestion des architectures dynamiques est l'objet de plusieurs travaux de recherche [Brad04], [YuML05], [BPPT06].

Résilience La composition de l'application et les méthodes utilisées pour réaliser l'assemblage de l'application peut contraindre la résilience. Dans l'approche classique, une application est définie d'une manière statique où tous les composants et leurs connexions sont définis explicitement et statiquement. Ceci rend l'application difficile à adapter et donc difficile à rendre résiliente. Par contre, les architectures dynamiques permettent de décrire le comportement d'adaptation à adopter pour faire face à un changement dans le contexte.

Abstraction	Évolutivité	Résilience
●	◑	◐

Tableau 2 : Architecture logicielle et l'ubiquitaire

3.5. TRAVAUX EXISTANTS

Dans cette section, nous présenterons brièvement quelques technologies qui offrent le moyen de décrire des applications complexes à travers des architectures logicielles. Dans la dernière partie de ce chapitre, nous présenterons le résultat de la comparaison de ces différentes technologies selon une liste de caractéristiques obtenues à partir de l'analyse des défis de l'environnent ubiquitaire. Cette comparaison sera résumée dans les tableaux du dernier chapitre.

3.5.1. DARWIN

Darwin est l'un des premiers langages de description d'architecture. En effet, ce langage est né dans les années 90 [KrMa90] dans le but d'offrir un langage de configuration pour le Framework Regis (environnement pour les applications réparties[MaDK94]). Ce langage fournit une sémantique basée sur le π-calcul pour la description des architectures logicielles [MaEK95].

Darwin permet de décrire des applications hiérarchiques contenant à la fois des composants primitifs et des composants composites. Ce langage est utilisé pour décrire les interfaces des composants primitifs, des connecteurs (« bind ») et pour décrire les composites. Ces derniers n'ont pas d'implémentation mais peuvent contenir d'autres composites ou des composants primitifs. Le code métier est donc contenu dans les composants primitifs qui peuvent être implémentés par un langage de programmation tel que Java ou C++.

Darwin permet de réaliser des architectures dynamiques et cela à travers deux mécanismes :

- **Instanciation paresseuse :** un composant (préalablement sélectionné) est instancié seulement lorsqu'il est réclamé par un autre composant.

- **Instanciation dynamique :** un composant est arbitrairement choisi et instancié en fonction de la compatibilité entre les interfaces fournies et requises.

Il existe cependant des inconvénients pour les architectures réalisées avec Darwin. Le départ d'instances n'est pas supporté car il est impossible de distinguer une instance dans un ensemble d'instances créées dynamiquement. Les adaptations réalisées sont alors souvent codées « en dur » dans chaque composant.

3.5.2. DYNAMIC WRIGHT

Dynamic Wright est une extension à Wright, un ADL[5] permettant de représenter la structure d'une application sous la forme d'un graphe où les composants et les connecteurs sont des nœuds. Pour rendre le langage dynamique, cette extension a intégré la description de reconfigurations dynamiques dans la description du système [AlDG98]. Cet ADL décrit la configuration initiale et la reconfiguration dynamique en utilisant une algèbre de processus proche de CSP[6].

Dynamic Wright est surtout utilisé pour l'analyse d'une application ou d'une famille d'applications par rapport aux différentes reconfigurations spécifiées. Par exemple, il supporte la vérification automatisée de contraintes architecturales. Ce langage supporte l'ajout et la destruction d'instances de composants et la création et la destruction de liaisons. Cependant, il ne possède pas d'environnement d'utilisation ou d'exécution : les architectures décrites ne peuvent être que vérifiées ou simulées.

3.5.3. ARCHJAVA

Pour pouvoir réaliser des applications ubiquitaires, certaines approches choisissent de coupler fortement les concepts d'architecture avec le langage de programmation, c'est le cas de ArchJava [AlCN02][Arch00]. Il intègre la spécification de l'architecture dans le langage Java. Ce couplage fort entre les deux langages permet de réaliser des analyses statiques du système réalisé [AlLo03]. À titre d'exemple, il est capable de garantir statiquement l'intégrité des communications dans la mise en œuvre de l'architecture [AlCN06]. Cette propriété est très intéressante pour les environnements ubiquitaires car elle évite la réalisation d'architectures inconsistantes qui peuvent causer des confusions ou des violations de propriétés architecturales. La majorité des ADLs n'est pas capable d'assurer une telle vérification.

Le concept de connecteur dans ArchJava peut être étendu pour supporter des stratégies d'auto-adaptation [ASCN02]. Cependant le dynamisme est limité à l'instanciation et aux liaisons automatiques des composants. Aussi, la gestion du dynamisme est limitée par le fait du couplage des deux langages : le code de gestion du dynamisme d'une architecture est mêlé au code métier des composants. Pour finir, la destruction d'instances de composants et de connexions n'est pas supportée.

[5] Architecture Description Language
[6] Communicating Sequential Processes

3.5.4. WCOMP

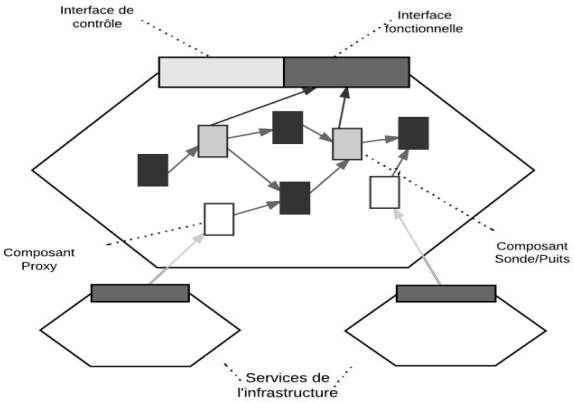


Figure 10 : Composite Wcomp

```
pointcut pcEngine:
  mail ←/sendMail∗/
  cache ←/sendCache∗/
  value ←/primitiveValueEmitter∗/
  battery_low ←/batteryLow∗/
advice ConsoMinimale(mail,cache,battery_low):
     value.^Emit -> ( cache.Send() )
 cache.^Sending -> ( battery_low.ShutDown() )
```

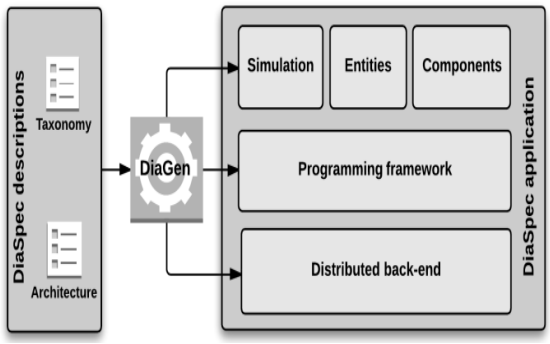
Figure 11 : Exemple d'une description d'AA

WComp s'adresse directement au domaine de l'informatique ubiquitaire et a été initié en 2003 [CFGJ03]. Il vise à répondre aux problèmes de la mobilité et de l'hétérogénéité des équipements dans un environnement d'exécution dynamique tels que l'ubiquitaire. C'est une approche basée sur un modèle à composants dit « légers » pour concevoir des services web composites. L'approche s'appuie sur les technologies à services UPnP [JeWe03] et DPWS [Thel06] pour faciliter la communication entre les services. Elle fournit également un environnement complet pour construire des applications sous forme de graphe de services web.

Les applications ubiquitaires sont définies en terme de compositions structurelles en utilisant le paradigme Service Lightweight Component Architecture (SLCA) [TLRH11]. Ce paradigme est basé sur une approche orientée service (les services web) et sur un paradigme de communication basé événement. Pour réaliser l'adaptation elle s'appuie sur ensemble d'Aspects d'Assemblages (AA). Un aspect d'assemblage [TCLR06] s'appuie sur le principe de la programmation orientée aspect, AOP [KiLM97]. Les aspects d'assemblage sont des pièces d'information décrivant comment un assemblage de composants peut être structurellement modifié. La propriété « boite-noire » des composants est alors conservé. Ces modifications incluent l'ajout de composants et les liaisons entre eux.

Les applications WComp sont construites par opportunisme, c'est-à-dire en fonction de la disponibilité des services UPnP et DPWS conformément à la définition de l'application faite en SLCA. Les mécanismes d'adaptation qui rentrent en jeu ne nécessitent pas la vision globale de l'architecture en cours d'exécution. Par conséquent, l'environnement d'exécution ne modélise pas le système.

3.5.5. DIASUITE




```
device LocatedDevice {
   attribute location as Location;}
device TemperatureSensor extends
LocatedDevice {
  source temperature as Temperature;}
device Heater extends LocatedDevice {
  action Heat { on(); off(); };}
structure Temperature { value as Float; }
structure Location { room as String; }
context AverageTemperature as Temperature {
  source temperature from TemperatureSensor;}
```

Figure 12 : Cycle de développement DiaSuite **Figure 13 : Exemple d'une description DiaSpec**

DiaSuite [CaBC10] est une approche outillée basée sur le paradigme Sense/Compute/Control (SCC) [CBME10][CBCL11][BeBC12] qui permet d'aider le développement et la simulation des applications pervasives. L'approche s'appuie sur un langage de conception dédié appelé DiaSpec [Cass11]. Ce langage permet de décrire :

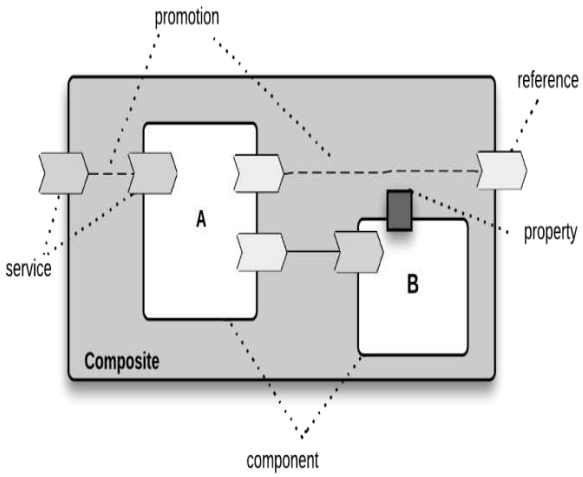
- **La taxonomie,** qui est une spécification d'éléments architecturaux appelés « entités ». Ces entités sont indépendantes et elles sont chargées de capturer des informations de l'environnement pour produire des données (on parle alors de « source »), ou de réaliser des actions sur ce dernier (on parle alors « action »). Elles gèrent l'interaction avec l'environnement.
- **L'architecture**, qui est composée d'opérateurs de contexte (qui dépendent des données produites par les entités) et d'opérateurs de contrôle (qui dépendent des actions réalisées par les entités). Elle est construite au-dessus de la taxonomie et gère la logique applicative de l'application.

À partir de cette description, d'autres outils prennent le relais. À partir de la spécification produite dans DiaSpec, un framework de programmation dédié peut être généré en Java grâce à DiaGen [CBLC09]. Ce framework fournit un support d'implémentation et d'exécution dédié au développement de systèmes d'informatique ubiquitaire [Merc11].

DiaSuite offre un support complet pour les différentes phases pour le développement d'une application pervasive à partir d'une description de haut niveau. Cependant, l'outil ne traite pas des aspects liés à l'évolution et à l'adaptation dynamique durant l'exécution des applications pervasives.

3.5.6. SERVICE COMPONENT ARCHITECTURE

Service Component Architecture (SCA) est une spécification développée par un consortium qui vise à combiner les avantages des approches à composants et des approches à services. L'objectif de SCA [BBCH07] est de simplifier l'écriture d'applications à base de composants sans se soucier du langage d'implantation du composant. Cette approche est issue d'une collaboration OSOA [Oasi00] entre différentes entreprises comme IBM, Oracle ou BEA Systems. Il existe plusieurs implantations de la spécification SCA telles que Websphere proposé par IBM [Ibm00], Tuscany d'Apache Software Foundation [Apac00b], ou encore Frascati [SeMe12]. L'approche est aussi réalisée dans différent langages pour Java, C++, C#, etc.




```
<composite name="MyApp">
  <service name="promoteA" … />
  <component name="A">
     <implementation.java class="…" />
     <service name="mSA">
        <interface.java interface="…" />
     </service>
     <reference name="mRA">
        <interface.java interface="…" />
     </reference>
  </component>
  <component name="B">
        …
  </component>
  <wire source="A/mRA" target="B/mSB" />
</composite>
```

Figure 14 : Un composite SCA **Figure 15 : Exemple d'une description SCA**

Une application est définie comme un ensemble de composants logiciels qui interagissent. Un composant SCA implante la logique métier dans un langage de programmation. Les composants SCA se basent sur plusieurs concepts, parmi lesquels on trouve :

- **Les services**, qui sont les fonctionnalités accessibles aux autres composants. Ce sont l'interface d'invocation d'une implémentation.
- **Les références,** qui sont les dépendances vers d'autres services fournis par d'autres composants, ce sont les fonctionnalités requises pour le bon fonctionnement du composant.
- **Les propriétés** qui sont des informations de configuration nécessaires lors de l'instanciation du composant.

L'unité de déploiement dans le domaine SCA est le composite. Il s'agit d'un assemblage hiérarchique qui suit le modèle de la composition structurelle et qui permet d'obtenir une fonctionnalité plus complexe à partir des services fournis par différents composants. Il contient des composants et décrit les différentes liaisons. Un composite est packagé sous la forme d'un zip, jar, ear, war, etc. Un composite est accompagné d'un descripteur XML qui liste l'ensemble des composants et des dépendances de services des composants. L'installation et l'activation d'un composite impose la présence et l'installation de la totalité des éléments nécessaires aux différentes dépendances des composants décrits dans le fichier XML.

SCA propose un mécanisme appelé « auto-wire ». Ce mécanisme offre la possibilité de provoquer une auto-résolution partielle, c'est-à-dire de résoudre des dépendances non explicitées par le développeur avec les composants décrits dans le composites. Du point de vue dynamisme à l'exécution, l'ajout d'éléments et la mise à jour et suppressions des composites sont des comportements possibles.

3.6. CONCLUSION

On trouve plusieurs comparaisons des architectures logicielles et leur langages dans différentes publications telles que [MeTa00], [Bure06] ou encore dans [Proj02]. Cependant, la comparaison que nous proposons dans la partie suivante tient compte uniquement des nouveaux défis soulevés par l'informatique ubiquitaire. Les technologies présentées précédemment ont été comparées selon les propriétés et caractéristiques extraites des problèmes soulevés par ce domaine.

4. COMPARAISON DE QUELQUES TECHNOLOGIES

Cette section compare les différentes technologies présentées dans les sections précédentes. Bien que la plupart de ces approches visent à supporter l'abstraction, l'évolutivité et la résilience 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 des applications ubiquitaire.

Notre comparaison détermine le champ d'actions de chacun des travaux présentés vis-à-vis des concepts de base des approches à composants et de l'architecture logicielle. Il s'agit des composants et des connexions. Dans la suite de cette section, nous décrirons chacun de ces critères. Nous illustrerons le résultat de notre étude comparative dans différents tableaux.

4.1. ABSTRACTION

L'abstraction permet de se détacher de la technologie d'implémentation pour exprimer l'application. Elle doit offrir des garanties au développement, et doit s'assurer de sa cohérence. L'abstraction doit donc offrir les caractéristiques suivantes :

- **Les niveaux d'abstraction :** les travaux présentés proposent des niveaux d'abstraction pour les composants et pour les connexions (dépendances). Plus l'abstraction est élevée, plus il est facile de substituer des composants ou des connexions à l'application. À chaque niveau d'abstraction, le composant (ou la connexion) peut avoir une représentation différente, on parle de **nature d'abstraction.** Cette nature peut être liée au cycle de développement, comme composant et instance ou elle peut décrire le niveau d'abstraction, comme le type de composant et le composant. Plus on dispose de plusieurs niveaux et de natures d'abstraction, plus l'application disposera de degrés d'adaptations. (voir Tableau 3).
- **La vérification des descriptions :** la séparation entre la logique métier et la technologie d'implémentation peut amener à produire des descriptions d'applications incohérentes. Il faut donc être capable de vérifier les descriptions abstraites des composants et des applications aussi bien à la compilation qu'à l'exécution. (voir Tableau 4).
- **Les contraintes des connexions :** les connexions peuvent exprimer des contraintes pour guider la composition des composants (voir Tableau 4). Ces contraintes sont caractérisées par le :
  - **Langage** : il s'agit du langage qui permet de décrire les contraintes. L'expressivité du langage joue un rôle important dans la description de l'application.
  - **Type :** les contraintes peuvent être décrites de manière déclarative, (une expression qui va être évaluée pour établir la connexion) ou de manière impérative, (les connexions sont établies par un programme tiers qui connaît la composition à réaliser).
  - **Emplacements :** les contraintes de connexions peuvent s'exprimer à différents endroits. Au niveau du composant, on parle de contraintes *intrinsèques* ; au niveau du composite (ou de l'application), on parle de contraintes *applicatives* et au niveau du contexte d'exécution, ce sont des contraintes *contextuelles*.

4.2. ÉVOLUTIVITÉ

L'évolution qui nous intéresse, est une évolution qui intervient à l'exécution de l'application. Il s'agit alors d'une évolution dynamique qui peut toucher aussi bien les composants que les connexions. Les travaux seront donc comparés suivant leur capacité à **ajouter, supprimer ou modifier des composants** (types ou instances). De même pour les **connexions** et la capacité de ces approches à **ajouter, supprimer ou substituer** des connexions à l'exécution (voir Tableau 5).

4.3. RÉSILIENCE

La résilience par rapport au contexte de l'exécution peut être obtenue grâce aux :

- **Emplacements de composants** : la diversité **des emplacements** dans lesquels on peut trouver des composants (voir Tableau 6). Ces composants peuvent être disponibles dans le

système local (un registre local par exemple), dans des dépôts de code ou dans des machines distantes. L'interaction avec chacun de ces emplacements va avoir des effets sur le contexte d'exécution des applications comme l'instanciation à partir d'une fabrique, ou la création automatique d'un proxy pour réaliser un appel distant.

- **Gestion des emplacements** : chaque emplacement peut être géré d'une manière différente selon sa nature. Il est question ici des moyens utilisés pour gérer ces emplacements. Cette gestion peut être globale à toutes les applications en exécution ou elle peut être **contextuelle** par rapport à chaque application. Le potentiel d'extensibilité est aussi important car il décrit le moyen d'intégrer de nouveaux emplacements.
- **Stratégies d'échec :** dans le cas où un composant est indisponible, il est nécessaire de décrire des politiques pour éviter l'échec de l'application. Ces stratégies définissent le comportement à adopter face aux changements du contexte d'exécution (voir Tableau 7).
- **Mode de résolution des connexions :** il s'agit de la politique de composition adoptée pour établir une connexion entre deux composants. Dans un environnement ubiquitaire où le contexte d'exécution intervient dans chaque connexion de composant, le choix de la politique est très important (voir Tableau 6). Il existe différentes politiques de compositions :
    - **Statique :** les connexions sont résolues avant l'exécution. Le contexte d'exécution n'intervient pas dans cette politique de résolution. Il est généralement impossible de modifier ces connexions à l'exécution.
    - **Explicite :** les connexions sont décrites explicitement vers une instance de composant (vers un objet bien précis). Ce genre de connexion supporte qu'un composant soit mis à jour mais il est impossible de le substituer par un autre.
    - **Dynamique :** les connexions sont réalisées pas à pas à l'exécution, ceci évite de faire un assemblage prédéfini de l'application. Elle permet de prendre en considération le contexte et peut se décliner en résolution (i) *immédiate*, c'est-à-dire au lancement de l'exécution, (ii) *paresseuse*, la réalisation de la connexion est effectuée uniquement à la demande, ou (iii) d'une manière *adaptative* où la résolution de la connexion est effectuée suite à un changement dans le contexte d'exécution (exemple : apparition d'un composant.

4.4. LES TABLEAUX DE COMPARAISON

Les résultats de notre étude comparative de l'état de l'art sont listés dans les tableaux suivants.

		Niveaux de l'abstraction		Nature de l'abstraction	
		Composants	**Connexions**	**Composants**	**Connexions**
Modèle à composant	**Fractal (Julia)**	1	1	Component	Wire
	EJB	1	-	Bean	**N/A**
	K-Component	2	1	Component, Instance	Connector
	iPOJO	2	1	Factory, Instance	Binding
	Kevoree	2	1	ComponentType, Instance	Channel
Langage d'architecture	**Darwin**	2	1	Component, Instance	Binding
	Dynamic Wright	1	1	Component	Port
	ArchJava (2002)	3	1	Incomplete component, Component, Instance	Connection
	WComp	2	1	Bean, Instance	Link
	DiaSuite	2	1	Component, Instance	**?**
	SCA (Tuscany)	3	1	ComponentType, Component, Instance	Wire

Tableau 3 : Niveaux d'abstraction et phase d'assemblage

		Langage de contraintes	Type		Emplacement des contraintes			Vérification des descriptions	
			Impératif	Déclaratif	Composant	Application	Contexte	Compilation	Exécution
Modèle à composant	**Fractal (Julia & FScript)**	FPath	◐	●	○	●	○	●	●
	EJB	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**
	K-Component	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**
	iPOJO	LDAP	◐	●	●	●	◐	◐	●
	Kevoree	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	◐	●
Langage d'architecture	**Darwin**	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**
	Dynamic Wright	Algèbre de processus (CSP)	○	●	○	●	○	●	○
	ArchJava (2002)	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	●	○
	WComp	Aspect d'assemblage	●	○	○	●	●	○	◐
	DiaSuite	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	●	○
	SCA (Tuscany)	○	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	●	●

Tableau 4 : Expression des contraintes sur les connexions et vérifications

		Composants			Connexions		
		Ajout	Suppression	Modification	Ajout	Suppression	Substitution
Modèle à composant	Fractal (Julia & FScript)	◐	◐	◐	◐	◐	◐
	EJB	◐	◐	○	◐	◐	○
	K-Component	●	●	●	●	●	●
	iPOJO	●	●	●	●	●	●
	Kevoree	●	●	●	●	●	●
Langage d'architecture	Darwin	●	◐	◐	◐	◐	◐
	Dynamic Wright	●	●	○	●	●	◐
	ArchJava	●	●	○	○	○	○
	WComp	●	●	●	●	●	●
	DiaSuite	◐	◐	○	○	○	○
	SCA (Tuscany)	●	●	●	●	●	◐

Tableau 5 : Évolutivité à l'exécution

		Emplacements			Gestion des emplacements			Effet de la résolution			
		Système local	Systèmes distants	Dépôt de code	Moyen de gestion	Contextuel	Extensibilité	Déploiement	Appel distant	Instanciation	Génération Code
Modèle à composant	Fractal (Julia & FScript)	●	○	○	○	○	○	○	○	◐	◐
	EJB	●	◐	○	○	○	○	○	◐	●	○
	K-Component	●	○	○	○	○	○	○	○	○	○
	iPOJO	●	○	○	○	○	●	○	○	◐	◐
	Kevoree	●	●	◐	◐	◐	●	◐	●	◐	◐
Langage d'architecture	Darwin	●	○	○	○	○	○	○	○	●	○
	Dynamic Wright	●	○	○	○	○	○	○	○	○	◐
	ArchJava (2002)	●	○	○	○	○	○	○	○	○	○
	WComp	●	◐	●	○	○	○	◐	◐	◐	◐
	DiaSuite	●	◐	◐	○	○	○	○	◐	◐	◐
	SCA (Tuscany)	●	●	○	○	○	○	○	◐	●	◐

Tableau 6 : Emplacement des composants pour la résolution

		Mode de résolution					Stratégies d'échec			
		Statique	Dynamique			Explicite	Échec	Attente	Retour arrière	Autre
			Immédiate	Paresseuse	Adaptative					
Modèle à composant	Fractal (Julia & FScript)	●	●	○	○	●	○	○	○	○
	EJB	●	◐	◐	○	●	○	○	○	○
	K-Component	●	●	○	○	●	○	◐	○	○
	iPOJO	●	●	●	●	●	◐	●	○	Nullable, Default-Implementaion
	Kevoree	●	●	◐	●	●	○	○	○	○
Langage d'architecture	Darwin	●	●	●	○	○	○	○	○	○
	Dynamic Wright	●	●	○	○	●	○	○	○	○
	ArchJava	●	○	○	○	○	○	○	○	○
	WComp	●	●	◐	●	●	○	◐	○	○
	DiaSuite	●	○	○	○	●	N/A	N/A	N/A	N/A
	SCA (Tuscany)	●	●	●	○	●	○	○	○	○

Tableau 7 : Mode de résolution & stratégies d'échec

		Abstraction		Évolution		Résilience	
		Niveaux d'abstraction	Contraintes	Composants	Connexions	Emplacements	Stratégies d'échec
Modèle à composant	Fractal (Julia)	◐	○	◐	◐	○	○
	EJB	◐	○	◐	○	◐	○
	K-Component	◐	○	●	●	○	◐
	iPOJO	◐	◐	●	●	○	◐
	Kevoree	◐	○	●	●	◐	○
Langage d'architecture	Darwin	◐	○	◐	◐	○	○
	Dynamic Wright	◐	◐	●	●	○	○
	ArchJava	◐	○	◐	○	○	○
	WComp	◐	◐	◐	●	◐	◐
	DiaSuite	◐	○	◐	◐	○	○
	SCA (Tuscany)	◐	○	●	◐	◐	○

Tableau 8 : Synthèse des technologies

DEUXIÈME PARTIE :
CONTRIBUTIONS

CHAPITRE 4
PROPOSITION

1. VUE GLOBALE

L'objectif de l'approche ApAM est de fournir un langage d'architecture et un modèle à composants à services pour décrire les applications ubiquitaires et une machine d'exécution de ce langage d'architecture. Les mêmes concepts de notre modèle à composants à services sont présents durant le développement et pendant l'exécution. Ainsi l'application sera décrite à l'exécution avec les concepts introduits dans le modèle de développement.

Le langage d'architecture et le modèle à composants à services proposé combinent des concepts provenant des architecture logicielles [GaSh93], de la programmation par composant [Szyp02] et de l'approche à service [Papa03b].

Contrairement à la plupart des approches portant sur les applications dynamiques, c'est la plate-forme d'exécution ApAM qui est responsable de déterminer l'architecture de l'application en fonction du contexte courant. Ce calcul se fait essentiellement à travers la résolution dynamique des dépendances entre composants.

Notre approche est basée sur une architecture dite **architecture de référence**. Elle est décrite à l'aide de notre langage d'architecture. Ce langage permet de définir des applications en intention. Cela permet de décrire l'application sans avoir à définir une architecture particulière, ni même un ensemble explicite (et forcément très limité en nombre) d'architectures concrètes. La plate-forme d'exécution calcule les architectures concrètes à l'exécution et s'assure qu'elles respectent les propriétés de l'architecture de référence.

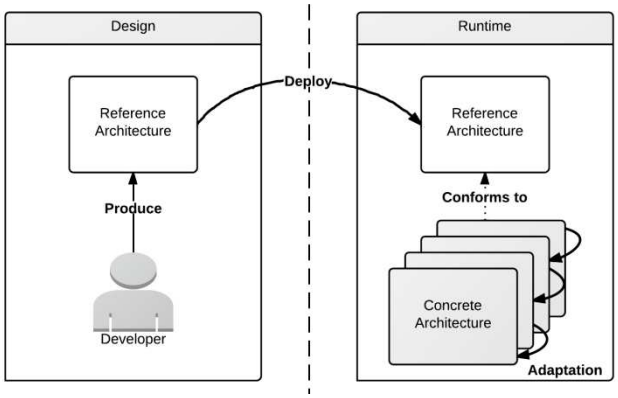


Figure 16 : Vue globale

Notre approche vise à rendre les applications résilientes au contexte, c'est-à-dire de faire en sorte que les applications ubiquitaires puissent s'exécuter sans interruption, en leur permettant de survivre aux changements imprévisible du contexte. Pour cela, nous proposons :

- Un langage d'architecture permettant de décrire des architectures de référence. Il s'agit d'une architecture définie par intention, ce qui correspond à un ensemble vaste et potentiellement non borné d'architectures concrètes.
- Un modèle a composant réflexif qui permet de raisonner sur l'application à l'exécution et de réaliser les adaptations d'architecture nécessaires.
- D'étendre l'espace de résolution des dépendances au-delà du système local. Cela permet aux applications de résoudre leurs dépendances en utilisant les ressources d'autres applications en exécution, les dépôts de code disponibles ou encore les plates-formes distantes.
- D'automatiser l'adaptation en offrant des mécanismes génériques et extensibles pour réagir aux changements du contexte d'exécution, en particulier face à une dépendance manquante ou invalide.

2. PRINCIPES DE L'APPROCHE APAM

L'approche d'ApAM consiste à définir une application au travers d'une architecture de référence. À l'exécution, la plate-forme ApAM interprète l'architecture de référence pour construire et/ou adapter l'architecture concrète de l'application en fonction du contexte d'exécution (voir Figure 16). Cette interprétation produit des architectures concrètes conformes à l'architecture de référence. L'adaptation est réalisée d'une manière automatique par un mécanisme générique qui modifie l'architecture, si besoin en fonction des changements de contexte. L'adaptation de l'architecture concrète permet de passer d'une architecture à une autre.

Selon l'emplacement du code de l'adaptation nous pouvons distinguer trois grandes logiques de l'adaptation dans les systèmes ubiquitaires (voir Figure 17) :

- **Une adaptation contrôlée,** où la logique d'adaptation est mélangée avec l'application. Le développeur décrit le code d'adaptation en même temps que le code de l'application. Ces deux codes sont mélangés.
- **Une adaptation dynamique,** où la logique d'adaptation est exprimée dans un conteneur de l'application géré par le système. Ceci permet d'automatiser un certain nombre d'adaptations qui sont complexes à réaliser à la main.
- **Une adaptation autonomique**, dans laquelle l'adaptation est réalisée par un programme externe. Elle repose sur une plate-forme capable d'observer (via des senseurs) et de modifier (via des effecteurs) les applications en exécution.

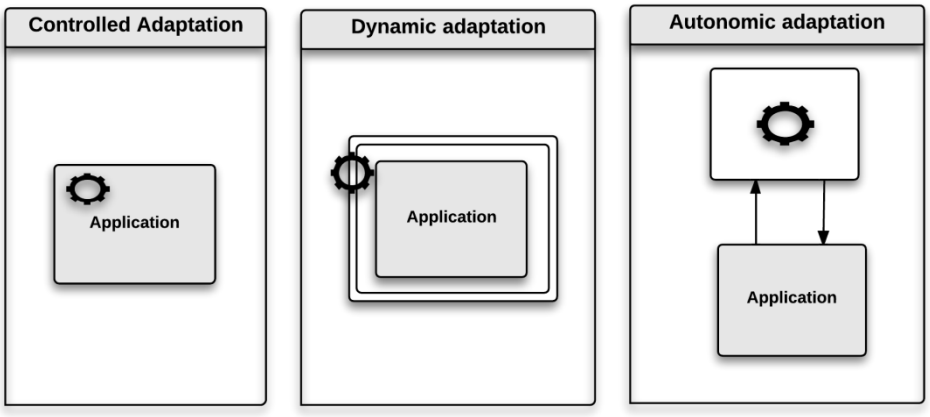


Figure 17 : Logiques de placement de l'adaptation

Dans notre approche, nous nous sommes efforcés à offrir un mélange de ses trois logiques d'adaptation, en tirant les avantages de chaque approche. Notre objectif est de permettre aux applications ubiquitaires d'exprimer la logique d'adaptation souhaitée et de la réaliser dynamiquement. Ainsi, ApAM offre des mécanismes génériques pour réaliser les trois logiques d'adaptation décrites ci-dessus, mais dans tous les cas en offrant ou en imposant le respect de l'architecture de référence (voir Figure 18). Cela assure le respect des invariants de l'application, ce que ne permettent pas les autres méthodes. Ces trois approches sont à la fois différentes et complémentaires et elles permettent aux applications ubiquitaires d'augmenter significativement leur niveau de résilience par rapport au contexte d'exécution.

Notons que dans les approches « traditionnelles » rien ne garantit que l'adaptation réalisée soit correcte. Le programmeur de l'adaptation n'a aucun « garde-fou », alors que dans notre approche et quel que soit le type d'adaptation effectuée, le système vérifie que cette dernière est conforme à l'architecture de référence. L'architecture de référence joue un rôle majeur puisqu'elle garantit la cohérence de l'application quel que soit le contexte et les modifications apportées.

L'architecture de référence est véritablement l'application. Chaque architecture concrète n'est qu'un état de cette application. Nous retrouvons, à un niveau d'abstraction supérieur, la distinction entre un programme et les états de ce programme lors de son exécution. C'est entre autre pour cela que nous parlons de « langage d'architecture exécutable ».

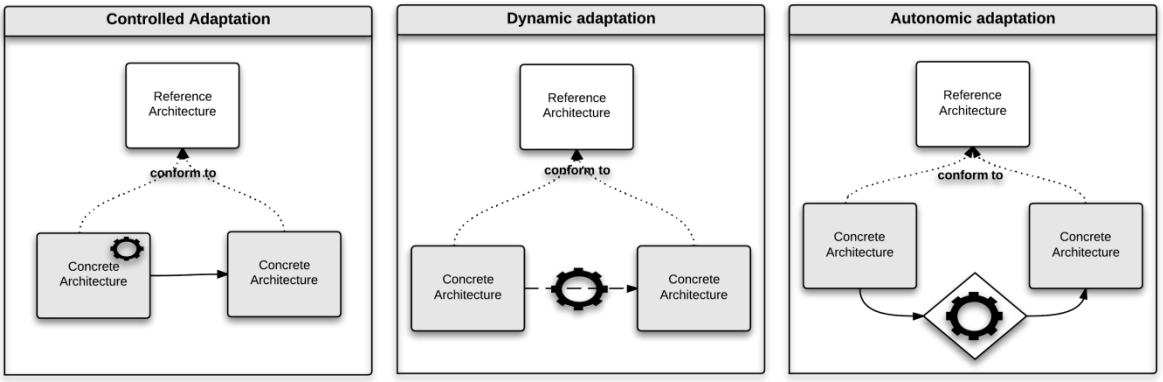


Figure 18 : Logiques d'adaptations de l'architecture concrète

2.1. COHÉRENCE DES CYCLES DE VIE

Le génie logiciel a pour but d'optimiser la prédictibilité, le coût et la qualité des logiciels. Jusqu'à récemment, cet objectif a été poursuivi au travers de méthodes, d'outils et d'ateliers de conception et de développement de logiciel. Traditionnellement, les activités du génie logiciel se situent en amont de l'exécution. Le logiciel est ensuite « packagé » et exécuté sans modification. De ce fait, les applications ainsi réalisées sont auto contenues et considèrent un contexte d'exécution connu et constant, ce qui est souvent vu comme une propriété souhaitable.

Dans les applications ubiquitaires, il n'est plus possible de faire l'hypothèse d'un contexte d'exécution constant. La frontière entre développement et exécution devient floue [BaGh10]. Les tendances actuelles poussent à utiliser à l'exécution les connaissances accumulées pendant le développement de l'application, de manière à pouvoir prendre des décisions et réaliser des adaptations en cohérence avec la description de l'application.

Dans l'approche ApAM, nous avons souhaité éliminer le cloisonnement existant entre les phases de développement, de déploiement et d'exécution. Ainsi, la même information est partagée et propagée durant tout le cycle de vie de l'application. La plate-forme d'exécution implémente des mécanismes lui permettant de réaliser la découverte de composants dans des espaces de résolutions distincts et hétérogènes, de réaliser les vérifications nécessaires pour la cohérence et le choix d'un composant et finalement de réaliser les adaptations nécessaires en réaction à des changements du contexte d'exécution.

2.2. MODÈLE DE DÉVELOPPEMENT

Le canevas à composants ApAM fournit un modèle de développement simple. Il permet de masquer la complexité du SOA dans les composants à service. Basé sur l'idée de Plain Old Java Object (POJO) [Fowl00], ce framework simplifie la réalisation des composants dynamiques pour les développeurs. Il masque la complexité du SOA au travers des métadonnées spécifiées par le développeur qui seront gérées par la machine d'exécution ApAM.

Le langage ApAM combine deux langages :

- **Un langage de programmation** qui permet de décrire le code fonctionnel du composant.
- **Un langage d'architecture** permettant de décrire les composants, leurs dépendances et de décrire les différents assemblages permis.

Cette séparation facilite l'évolution de l'application. Les modifications réalisées au niveau de l'architecture pourront être réalisées indépendamment du code métier de l'application et vice versa.

C'est la combinaison de ces deux langages qui permet d'avoir des applications flexibles et facilement adaptables. Même si les deux langages peuvent évoluer indépendamment, c'est leur complémentarité qui permet de développer « simplement » des applications ubiquitaires complexes. Ce mariage a pour objectif de permettre l'expression de la logique métier dans le langage de programmation « presque » sans tenir compte des aspects dynamiques. Inversement, le langage d'architecture traite les aspects de l'application qui sont liés à la nature du contexte d'exécution (dynamisme, résilience, appel distant, etc.) sans avoir de connaissance sur la logique métier. Cette démarche est classique et commune aux langages d'architecture. Par opposition aux langages d'architecture, nos deux langages sont fortement couplés pour former « un seul langage » afin de fournir des garanties de cohérence et de vérification, tant à la compilation qu'à l'exécution.

2.3. LANGAGE D'ARCHITECTURE PAR INTENTION

La conception d'applications ubiquitaires s'appuie sur le langage d'architecture ApAM. Ce langage se distingue par le fait qu'il n'est pas nécessaire de décrire explicitement l'ensemble des composants de l'application, ni leurs connexions. ApAM encourage les développeurs à donner une description abstraite de l'application ou du composant. Ils peuvent spécifier le minimum nécessaire au démarrage et à l'exécution de l'application, laissant à ApAM le soin de compléter la composition.

À l'exécution, ApAM interprète l'architecture de référence et produit une architecture concrète, soit en remplaçant les composants abstraits par des composants plus concrets, soit en déployant des composants, soit en établissant des relations de dépendance (wires) entre des composants concrets.

Au lieu de décrire explicitement l'ensemble des adaptations possibles, cette approche laisse la liberté à la machine d'exécution de réaliser les adaptations en cohérence avec la description de l'application et en fonction du contexte d'exécution. Lorsque l'ensemble des contextes possibles et le nombre de composants deviennent importants, le nombre d'adaptations possibles devient gigantesque voire infini, impossible à calculer manuellement. Nous estimons que seul l'interpréteur du langage d'architecture peut réaliser cette tâche.

2.4. GARANTIES ET COHÉRENCE DU LANGAGE D'ARCHITECTURE

Dans la plupart des approches dynamiques étudiées, il est difficile de savoir si l'architecture réalisée au développement va pouvoir s'exécuter. Cela est souvent dû à des expressions abstraites dans le langage d'architecture qui ne sont interprétées qu'à l'exécution.

Grâce au mécanisme de groupes d'ApAM, le langage d'architecture est compilé. La description des composants est vérifiée et leurs dépendances sont validées. L'architecture de référence décrite en utilisant le langage d'architecture est compilée en même temps que les composants, dans le même espace de construction (Maven). Les relations entre le langage de programmation d'un côté et les dépendances avec leurs contraintes de l'autre côté, sont fortement couplées. Elles sont compilées et vérifiées par rapport aux autres composants qui entrent dans la définition de l'application ou utilisés par le langage de programmation.

2.5. MACHINE D'EXÉCUTION ET DE RÉSOLUTION DE DÉPENDANCES

La machine ApAM est un interpréteur du langage d'architecture. Elle s'appuie sur :

- La résolution de groupes (voir Chapitre 5 §4.1.1).
- La résolution de dépendances (voir Chapitre 5 §5.1).
- Les espaces de sélection (ou de résolution) (voir Chapitre 5 §5.2.1).

Le premier mécanisme permet de remplacer un composant abstrait par un composant plus concret satisfaisant certaines contraintes. Le deuxième mécanisme permet de calculer les dépendances entre les composants (et les applications) décrites en intention. Enfin, le troisième mécanisme utilise

espaces de résolution de composant permettent à ApAM de sélectionner un composant en fonction du contexte d'exécution. Cette sélection est réalisée à partir de la description abstraite du composant. Ces espaces permettent à ApAM d'inclure les composants appartenant à un ensemble de plates-formes distantes et hétérogènes dans le processus de résolution.

La plate-forme d'exécution d'ApAM repose sur un mécanisme d'interception et d'injection [Fabr76]. La plate-forme intercepte les appels vers les dépendances de service. L'interception est réalisée chaque fois qu'un composant essaie d'appeler l'une de ses dépendances. ApAM réalise une résolution pour chaque appel intercepté et sélectionne le composant qui satisfait la dépendance. Enfin, l'instance du composant sélectionné est injectée dans le champ correspondant. Ce mécanisme permet à ApAM de gérer les dépendances de l'application et de ses composants à chaud et durant leur exécution.

2.6. MÉCANISMES DE RÉSILIENCE

En plus de pouvoir introspecter l'état du système et de le reconfigurer, ApAM apporte des mécanismes génériques et extensibles pour assurer aux applications ubiquitaires un degré de résilience vis-à-vis du contexte d'exécution. Nous proposons ainsi un contexte d'exécution plus vaste et diversifié pour assurer la diversité de l'environnement. Ainsi, ApAM offre la possibilité d'interagir avec des plates-formes hétérogènes, tout en offrant des mécanismes d'introspection homogènes. Ces plates-formes peuvent être des dépôts de code, des machines ou des services distants, ou encore d'autres Framework à base de modèle à composants ou à service.

D'autres mécanismes de résilience permettent à ApAM de gérer les situations d'échec de résolution (voir Chapitre 5§ 5.2.4). ApAM offre des stratégies de résilience qui s'appliquent aussi bien aux composants qu'aux applications. Ces stratégies se déclenchent à l'exécution. Elles permettent de faire face à des changements de contexte d'exécution qui peuvent provoquer un échec de l'application. L'application peut ainsi se mettre en attente de l'arrivée d'un composant qui satisfait sa dépendance, signifier que son architecture est en état d'échec, ou encore remplacer une branche de son architecture invalide par une autre branche. Tout cela en restant en exécution.

CHAPITRE 5
APAM : MODÈLE DE COMPOSANTS A SERVICES

1. INTRODUCTION

Le langage d'architecture d'ApAM repose sur un nouveau modèle à composants à services. Ce modèle combine les concepts de l'approche à composant, de l'approche à service et de l'architecture logicielle. L'assemblage des composants à services, en utilisant notre langage, permet de décrire des applications ubiquitaires. Notre modèle a été conçu pour satisfaire les exigences de notre proposition. Dans ce chapitre, nous présenterons pas-à-pas notre modèle, puis nous détaillerons les mécanismes de groupe, de résolution et de résilience présentés dans la partie précédente.

2. LE COMPOSANT À SERVICE

La notion centrale de notre modèle est le concept de composant à service. Par abus de langage, nous parlerons de composant (voir Figure 19). Un composant est caractérisé par un ensemble de propriétés, par les ressources fournies et requises (interfaces, messages) et par les contraintes et préférences que doivent satisfaire les ressources requises.

D'une façon générale, nous définissons une **ressource** comme tout élément qui peut être fourni ou requis par un composant (voir §2.2 ci-dessous).

$\mathbb{R} \stackrel{\text{def}}{=}$ l'ensemble fini de toutes les ressources distinctes de l'application.

Un composant peut être décrit par un ensemble de **propriétés**. Une propriété est un couple composé d'un nom d'attribut et d'une ou plusieurs valeurs (en fonction de sa multiplicité) qui permettent de configurer le composant (voir §2.1 ci-dessous).

$\mathbb{A} \stackrel{\text{def}}{=}$ l'ensemble fini de tous les identificateurs d'attributs, et
$\mathbb{V} \stackrel{\text{def}}{=}$ l'ensemble (potentiellement infini) de valeurs possibles d'une propriété
Les propriétés du composant sont alors représentées par un sous-ensemble de $\mathbb{A} \times \mathbb{V}$

Les propriétés permettent aussi de définir des prédicats qui peuvent être utilisés pour sélectionner des composants. ApAM propose un **langage** de **contraintes** (voir §2.2.2 ci-dessous) pour exprimer ces prédicats, en utilisant les noms d'attributs comme variables dans des expressions booléennes.

$\mathbb{E} \stackrel{\text{def}}{=}$ l'ensemble des expressions du langage

La fonction **vars** : $\mathbb{E} \rightarrow P(\mathbb{A})$ permet de déterminer les variables libres d'une expression. L'interpréteur du langage permet d'évaluer une expression sur un ensemble des propriétés pour obtenir une valeur de vérité. Pour toute expression $exp \in \mathbb{E}$, on note :

$[\![\text{exp}]\!] : \mathrm{P}(\mathbb{A} \times \mathbb{V}) \rightarrow \mathbb{B}$ le prédicat associé à l'expression par l'interpréteur

Les propriétés des composants sont fortement typées (voir § 4.2 ci-dessous) Une **définition de propriété** permet de spécifier pour un attribut donné le type des valeurs possibles. Pour représenter les types de valeurs, on définit :

$\mathbb{T} \stackrel{\text{def}}{=}$ l'ensemble des types primitifs

Actuellement les types reconnus par ApAM sont les chaînes de caractères, les entiers, les booléens et les énumérations. La fonction type : $\mathbb{V} \rightarrow \mathbb{T}$ permet de déterminer le type d'une valeur. Les définitions de propriétés permettent également de typer les expressions du langage de contraintes.

Les ressources requises par un composant sont déclarées de façon explicite par une **définition de dépendance**. ApAM propose deux mécanismes pour spécifier les ressources requises (voir § 2.2.1 ci-dessous) : donner la référence de la ressource ou spécifier directement le composant la fournissant. Le deuxième mécanisme apparaît rigide à première vue mais l'introduction des composants abstraits (voir § 4.1 ci-dessous) va apporter d'avantage de flexibilité et de contrôle sur la résolution de dépendances.

À l'exécution, plusieurs composants peuvent fournir les ressources requises. Néanmoins seul un sous ensemble peut satisfaire les besoins du composant qui a exprimé la dépendance. Par conséquent, nous pouvons associer à chaque dépendance des contraintes qui expriment les caractéristiques que les composants cibles doivent satisfaire pour être acceptés en tant que fournisseurs. Nous avons :

$\mathbb{D} \stackrel{def}{=}$ l'ensemble fini des identificateurs de dépendance et
$\mathbb{C} \stackrel{def}{=}$ l'ensemble fini des identificateurs de composant
Les définitions de dépendances vont être représentées par une relation fonctionnelle de $\mathbb{D}$ vers $(\mathbb{R} \cup \mathbb{C}) \times P(\mathbb{E})$

Ainsi, nous avons pu caractériser un composant par son identificateur et un 4-uplet avec ses ressources, ses propriétés et ses définitions. Nous désignerons par **Composant** l'ensemble de tous les composants :

$$\textbf{\textit{Composant}} \stackrel{def}{=} \{(id, r, p, pd, dd) \mid id \in \mathbb{C}, r \subseteq \mathbb{R}, p \subseteq \mathbb{A} \times \mathbb{V}, pd \subseteq \mathbb{A} \times \mathbb{T}, dd \subseteq \mathbb{D} \times ((\mathbb{R} \cup \mathbb{C}) \times P(\mathbb{E}))\}$$

Nous avons défini les fonctions suivantes pour faciliter la manipulation des descriptions de composant :

$$\text{id} : \textbf{\textit{Composant}} \to \mathbb{C} \stackrel{\text{def}}{=} (\text{id, r, p, pd , dd}) \mapsto \text{id}$$
$$\text{ressources} : \textbf{\textit{Composant}} \to P(\mathbb{R}) \stackrel{\text{def}}{=} (\text{id, r, p, pd , dd}) \mapsto \text{r}$$
$$\text{propriétés} : \textbf{\textit{Composant}} \to P(\mathbb{A} \times \mathbb{V}) \stackrel{\text{def}}{=} (\text{id, r, p, pd , dd}) \mapsto \text{p}$$
$$\Delta\text{-propriétés} : \textbf{\textit{Composant}} \to P(\mathbb{A} \times \mathbb{T}) \stackrel{\text{def}}{=} (\text{id, r, p, pd , dd}) \mapsto \text{pd}$$
$$\Delta\text{-dépendances} : \textbf{\textit{Composant}} \to P(\mathbb{D} \times ((\mathbb{R} \cup \mathbb{C}) \times P(\mathbb{E}))) \stackrel{\text{def}}{=} (\text{id, r, p, pd , dd}) \mapsto \text{dd}$$

Généralement, une **application** est décrite par un ou plusieurs composants. Dans ApAM, l'architecture d'une application est exprimée par l'ensemble de ses composants et leurs dépendances.

Nous pouvons définir**, à ce stade**, l'architecture A d'une application par :

$$A \in \textbf{\textit{Architecture}} \subseteq \text{Composant}$$

Par la suite, nous détaillerons cette description du composant pour expliquer l'utilité de chacun des éléments qui va composer notre application ApAM.

2.1. PROPRIÉTÉS

Les propriétés sont des couples <attribut, valeur> où chaque attribut est l'identifiant de la propriété. Le type de la propriété indique si la valeur de la propriété doit être unique ou plutôt un ensemble de valeurs. Par exemple, pour un composant qui représente un dispositif physique, les propriétés peuvent être utilisées pour décrire l'état du composant (allumé/éteint), sa localisation (chambre, cuisine, etc.) ou tout simplement le nom du vendeur du dispositif (Siemens, Fagor, etc.).

Les propriétés ont deux rôles dans la description du composant :

- Elles peuvent être utilisées pour configurer le composant. Ainsi pour un même composant on peut avoir des configurations différentes. Les valeurs des propriétés peuvent être spécifiées à n'importe quelle étape du cycle de vie du composant. De plus, ApAM fournit les mécanismes nécessaires pour modifier les valeurs des propriétés dynamiquement et pendant l'exécution du composant.
- Elles sont utilisées pour vérifier les contraintes des dépendances. Comme les propriétés représentent la configuration actuelle d'un composant, ApAM valide les dépendances vers un composant en vérifiant que les contraintes de dépendances exprimées sur le composant source seront satisfaites par les propriétés exprimées sur le composant destination.

2.2. RESSOURCES

ApAM définit le concept de ressource. Ce concept permet de factoriser l'ensemble des éléments qu'un composant peut fournir ou requérir. Dans ApAM, il est possible de déclarer deux types de ressources : les interfaces et les messages. Cependant le concept est extensible et il peut être enrichi par d'autres types de ressource, comme les événements par exemple. Ce mode de communication asynchrone est très répandu pour la gestion des environnements ubiquitaires [GaSS02][RHCR02][CCGL07].

Les interfaces décrivent les fonctionnalités des composants. Un composant qui expose des interfaces est un fournisseur et un composant qui requiert des interfaces est un consommateur. Un composant peut être à la fois fournisseur et consommateur.

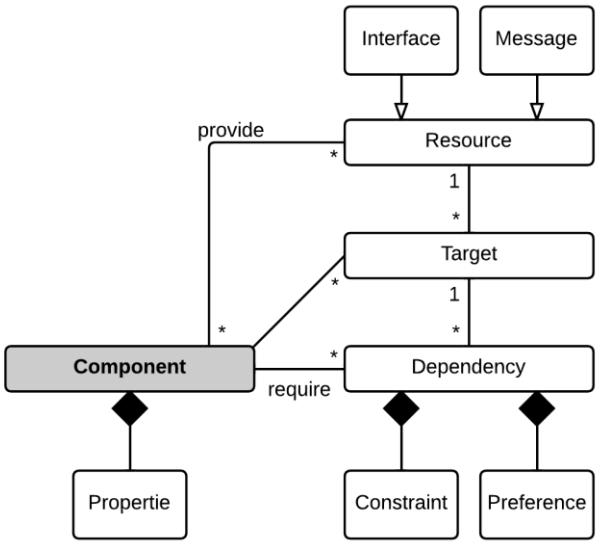


Figure 19 : Composant à service

Les messages sont utilisés dans ApAM pour gérer les flots de données, comme les données provenant d'un équipement de la maison (thermomètre, pluviomètre, etc.). Les interactions avec les messages se basent sur le paradigme producteur-consommateur. Dans ApAM, les producteurs et les consommateurs sont caractérisés par le type de données (ou messages) qu'ils produisent ou consomment.

2.2.1. DÉPENDANCES

ApAM est basée sur l'approche orienté service (Service-Oriented Computing) pour décrire les dépendances entre les composants. Les dépendances sont résolues par rapport aux ressources présentes dans la plate-forme d'exécution ApAM. Les composants ApAM sont faiblement couplés et peuvent réagir à l'apparition et à la disparition d'autres composants.

Les dépendances dans ApAM peuvent être définies soit par des ressources, soit par des composants (voir Figure 19). Dans son implémentation actuelle, ApAM gère deux types de dépendances de ressources :

- Les dépendances basées sur les interfaces sont similaires à la plupart des approches à services dynamiques. Grâce à ce type de dépendances, ApAM offre des interactions synchrones entres les composants en utilisant des appels de méthode. Les composants qui utilisent ces dépendances sont faiblement couplés. L'interaction entre les composants est basée sur les méthodes décrites dans l'interface. Dans ce type de dépendances l'interaction est à l'initiative du consommateur.
- Les dépendances de messages gèrent les flots de données et sont basées sur le paradigme producteur-consommateur. Dans ce patron l'initiative d'échange de message est partagée entre le producteur et le consommateur. Un producteur peut envoyer des messages à tous les consommateurs connectés à lui, on parle alors d'une communication en mode PUSH. À l'inverse, un consommateur peut réclamer des données auprès de ses producteurs : on parle alors de communication en mode PULL.

ApAM définit le concept de dépendance complexe qui représente une dépendance vers un ensemble de ressources qui doivent être offertes par un même composant. 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 (voir Figure 20).

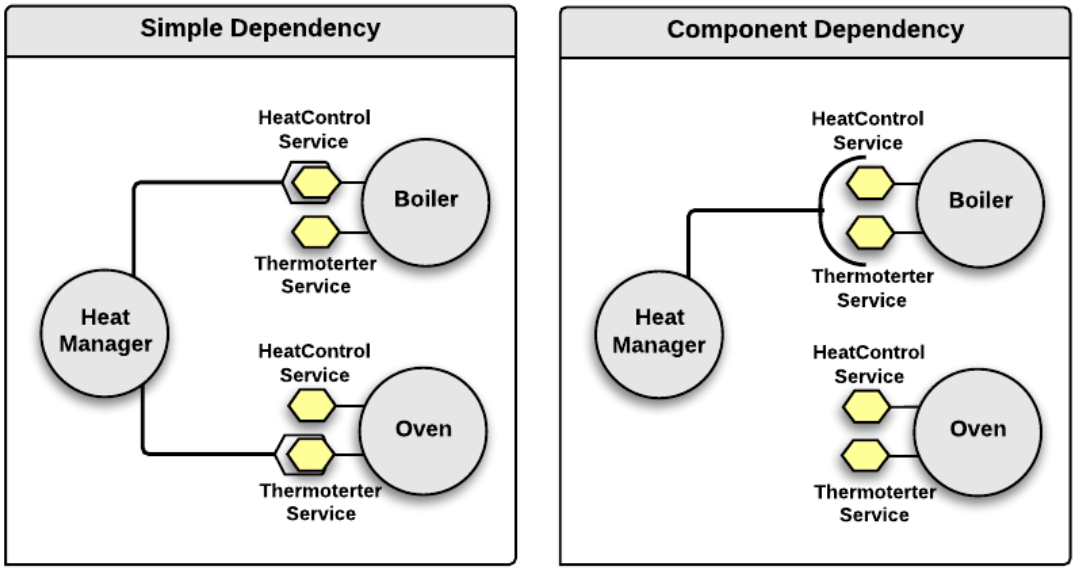


Figure 20 : Dépendances simples et complexes

Une dépendance de ressource ou de composant, possède des attributs de cardinalité (qui expriment l'optionalité et la multiplicité de la connexion) et de filtrage qui définissent les besoins spécifiques (contraintes) et préférables (préférences) que les fournisseurs des ressources requises doivent satisfaire.

2.2.2. CONTRAINTES

À l'exécution, une résolution de dépendances (uniquement basée sur les ressources requises) peut avoir comme solution un vaste ensemble de composants. Cependant, seul un sous ensemble peut satisfaire exactement les besoins spécifiques exprimés par une dépendance. Par conséquent, chaque composant doit être capable de décrire ses besoins. Pour cela, un composant peut associer à chacune de ses dépendances, des filtres qui expriment les caractéristiques qui doivent être satisfaites par la

solution (le composant cible). Par ailleurs, dans ce sous ensemble de composants sélectionnés, des préférences permettent de mieux cibler les candidats possibles. Ces préférences peuvent aussi s'exprimer dans ApAM de la même manière que les besoins spécifiques. Deux types de filtres sont alors gérés dans ApAM pour spécifier les besoins : les contraintes et les préférences.

Les contraintes sont exprimées dans un langage inspiré de LDAP[7], qui vérifie la validité des expressions afin d'assurer la compatibilité entre les composant clients et les composants fournisseurs. Ces expressions définissent des exigences logiques sur les valeurs des propriétés des composants destinataires. Le nombre de contraintes qui peut être exprimé est arbitraire et dépend de chaque composant.

2.3. PRÉFÉRENCES

Malgré les contraintes, le mécanisme de résolution de dépendances peut retourner un grand nombre de composants. Si la dépendance est multiple, tous ces composants sont solutions de cette dépendance. Quand la dépendance est simple (et non-multiple), un seul composant doit être sélectionné. Les préférences permettent de guider ce choix.

Les clauses de préférences spécifient un certain nombre de conseils pour trouver le « meilleur » composant à choisir. Elles permettent de réaliser un deuxième filtrage sur les composants qui ont validé les contraintes. Si plus d'un candidat satisfait le filtre de préférences, un composant sera choisi arbitrairement dans cet ensemble. Les préférences sont ignorées si aucun candidat ne satisfait les filtres de préférences.

[7] Lightweight Directory Access Protocol

3. SCÉNARIO D'APPLICATION

Avant d'aborder les diverses évolutions de notre modèle à composant, nous présenterons un exemple d'application ubiquitaire. Tout au long de ce chapitre nous nous baserons sur ce scénario pour donner des exemples.

Home Light Follow Me

Notre application permet de gérer l'éclairage de la maison. Grâce à cette application, les lumières de la pièce vont s'allumer ou s'éteindre en fonction de la présence de l'utilisateur dans la pièce. Cette application va utiliser différents équipements qui vont remplir différentes tâches : détecter la présence de l'utilisateur, vérifier la luminosité de la pièce, allumer ou éteindre la lumière.

L'application est hébergée sur la passerelle résidentielle. Elle est constituée d'un composant de gestion de lumière qui utilise les dispositifs de la maison. On considère que les différents dispositifs utilisés dans la maison sont dynamiques : les lumières et les détecteurs peuvent changer de pièce et peuvent apparaître ou disparaître à tout moment de la pièce ou de la maison. Quand un équipement est disponible il est automatiquement détecté par la passerelle et il rejoint spontanément l'application de gestion de lumière. De la même manière, quand un dispositif disparaît, il n'est plus disponible par l'application qui va devoir s'adapter à ce changement de contexte.

L'application **« Home Light Follow Me »** est composée d'un ensemble de composants qui interagit avec les différents types de dispositifs disponibles dans la maison. Aussi, l'application devra sélectionner les composants les mieux adaptés pour un contexte donné (emplacement de l'utilisateur, état des lumières, etc.).

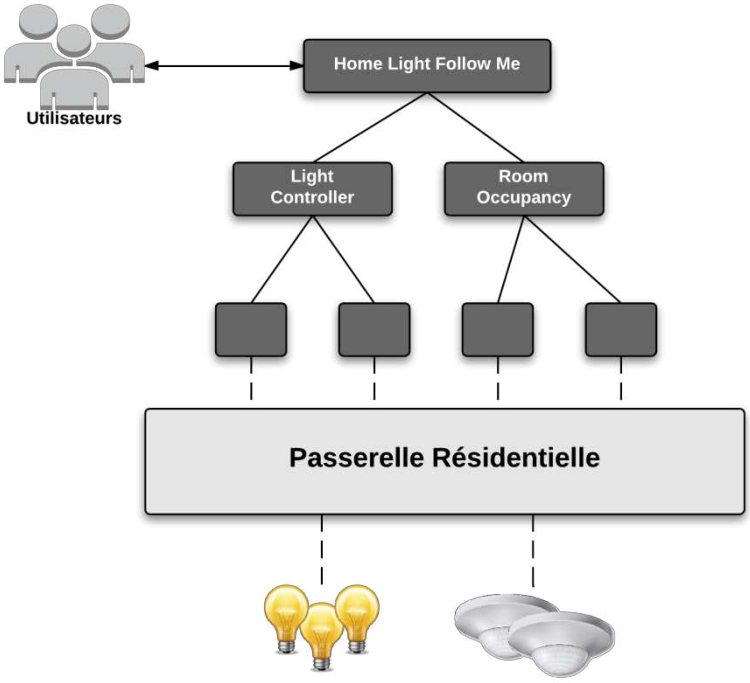


Figure 21 : Exemple d'applications

Notre objectif, à travers cet exemple, est de montrer comment il est possible de définir une application ubiquitaire dans ApAM qui réagit dynamiquement à l'apparition ou à la disparition des composants. Cette application doit continuer à fonctionner même si des parties de l'application ne sont pas disponibles. Par exemple : si une lampe d'une pièce disparaît (ou a changé de pièce), l'application doit continuer à éclairer la pièce en utilisant les autres lampes disponibles.

4. DÉFINIR ET CONTRÔLER L'ESPACE DE SÉLECTION

Les applications ubiquitaires ont la particularité d'être confrontées à un environnement dynamique. Les architectures qui décrivent ces applications doivent alors permettre aux applications en exécution de :

- Sélectionner les composants nécessaires à l'exécution. Cette sélection doit se faire au bon moment pour que ce choix respecte les contraintes du contexte. Cette adaptation vient compléter l'architecture concrète de l'application et ne met pas en question les choix réalisés précédemment.
- Substituer des composants de l'application pour faire face à un changement de contexte d'exécution qui met en question des choix réalisés précédemment. La description de l'architecture doit alors permettre de décrire l'ensemble des composants qui peuvent se substituer les uns aux autres.

Dans cette partie nous détaillerons les mécanismes que nous avons mis en place pour permettre la substitution et la sélection dynamique de composants.

4.1. LES NIVEAUX D'ABSTRACTION

Notre approche vise à rendre l'architecture plus flexible à l'exécution en exprimant l'architecture à plusieurs niveaux d'abstraction. Cette abstraction concerne aussi bien l'application que les différentes entités qui la composent.

Nous avons analysé différentes approches pour la description d'architecture. La plupart de ces approches décrivent l'application par un assemblage statique des composants de l'application et ce à différent niveaux d'abstractions. Chaque niveau d'abstraction correspond à une structure architecturale. Chacune représentant un point de vue (préoccupation) du système, nous avons identifié les assemblages suivants :

- S'il n'y a qu'un seul niveau, l'architecture est décrite en termes d'instances, ce qui ne permet que des adaptations très limitées, car les instances et les liaisons d'assemblage sont nommées explicitement.
- S'il y a deux niveaux d'abstraction, l'architecture est décrite en utilisant le type des composants. Ces assemblages sont plus flexibles car ils permettent de choisir les instances qui vont composer l'application. Les liaisons sont exprimées entre types de composant, ce qui permet de créer/détruire des liens entre instances. Cependant, les types de composants sont déclarés explicitement dans l'architecture ce qui est une forte limitation dans un monde ouvert.

Pour rendre les architectures plus flexibles ApAM propose d'utiliser un niveau d'abstraction supplémentaire. L'architecture est alors décrite en terme de « groupes » de composants ce qui permet à la plate-forme de choisir le (type de) composant le plus adapté à un contexte d'exécution donné, même si ce composant n'est pas décrit ou connu lors du développement.

4.1.1. GROUPES D'ÉQUIVALENCE

Il s'agit d'un des concepts essentiel sur lequel se base notre approche. Un groupe d'équivalence est défini comme un ensemble d'éléments qui sont indiscernables pour un certain point de vue. Un point de vue est défini comme un ensemble de propriétés et de relations que possèdent tous les membres d'un groupe d'équivalence : c'est la connaissance que nous avons de ces composants vus depuis un niveau d'abstraction supérieur.

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 : c'est le seul élément visible depuis le niveau d'abstraction supérieur. Il contient 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.

L'utilisation de ce concept est née de la volonté de classifier les éléments, de propager des attributs à différents niveaux d'abstraction et de permettre de typer et spécialiser les éléments de l'application. Il existe différents travaux et approches qui vont dans ce sens. Ainsi, nous retrouvons le concept de groupe d'équivalence sous différentes formes dans différentes technologies et langages :

- En UML[8], sous le terme de « power type », un groupe d'équivalence est formé par un type et l'ensemble de ses sous-types, dans lequel chaque sous-type hérite de toutes les propriétés définies par son type [GoHe06][Odel98].
- En OOP[9], une classe et ses instances forment un groupe d'équivalence. Les instances d'une classe décrivent les mêmes attributs définis par leur classe. Elles donnent aussi une valeur à chaque attribut variable déclaré par celle-ci.
- Dans le MDA[10], le groupe d'équivalence est représenté par la définition de l'application et les différents raffinements réalisés par les ingénieurs.
- Dans la matérialisation [DaPZ02], [ZiPY94], des facettes disjointes (instances et types) sont associées à chaque matérialisation. Les attributs des facettes sont hérités et propagés à chaque niveau.
- Dans l'approche d'instanciation profonde [AtKü01], tout élément est potentiellement un type qui peut être instancié. Cette approche définit le concept de « potency ». Il s'agit d'un nombre qui indique le niveau d'instanciation nécessaire pour chaque attribut. Chaque type et ses instances représentent alors un groupe d'équivalence.

Nous avons étendu le modèle de composant présenté précédemment avec le concept de groupes. Cette extension a pour but, entre autre, de permettre la substitution d'un composant par un autre. Le concept de groupe introduit trois matérialisations : **spécification, implémentation et instance**, chacune correspondant à un niveau d'abstraction différent. Chaque composant A à un niveau d'abstraction donné Ai définit un ensemble de composants Ai+1, appelé les membres de Ai, qui sont substituables les uns par les autres. Ainsi l'espace de choix pour réaliser la substitution d'un composant B, au niveau d'abstraction j (noté Bj) est identifié comme l'ensemble des membres du composant Bj. Cet ensemble est donc directement identifiable : il a été validé et certifié au préalable, assurant une substitution rapide et sure des composants.

Grâce au concept de groupe, un composant définit à un niveau i est une abstraction de l'ensemble des composants des niveaux d'abstraction i + n. Nous proposons la formalisation suivante pour décrire l'abstraction :

Pour représenter les notions de composant abstrait et de niveaux d'abstraction, nous étendons la notion d'architecture d'une application par la définition d'une hiérarchie de composants. L'architecture A d'une application ApAM est définie par l'ensemble de ses composants et par une relation hiérarchique :

$$A \in \textit{Architecture} \stackrel{def}{=} \{ (\boldsymbol{C}, \textit{matérialise}) \mid \boldsymbol{C} \subseteq \textit{Composant}, \textit{matérialise} \subseteq \boldsymbol{C} \times \boldsymbol{C} \}$$

Pour tout composant *a,c* ∈***C*** tels que *« c »* **matérialise** *« a »*, on dit que *« a »* est une abstraction de *« c »* et réciproquement que « *c* » est une matérialisation ou une concrétisation de « *a* ». Pour s'assurer que le graphe A corresponde effectivement à une hiérarchie, la relation binaire **matérialise** doit être antiréflexive, antisymétrique et sa fermeture transitive antiréflexive.

De plus, dans ApAM, nous imposons la condition suivante $\forall c \in \mathbf{C} : |\{a \mid c \text{ matérialise } a\}| \leq 1$. La hiérarchie d'abstraction dans ApAM est donc effectivement une forêt. Cette contrainte additionnelle

[8] Unified Modeling Language
[9] Object Oriented programming
[10] Model Driven Architecture

vient du fait que nous utilisons cette hiérarchie pour définir un système de typage (voir §4.2 ci-dessous) que nous voulons garder relativement simple.

À partir de cette définition de base, nous pouvons formaliser la notion de groupe d'équivalence. Pour toute **architecture** ***A*** nous définissons une nouvelle relation binaire sur ***C***, que l'on nomme **substituable** :

$$(c1,c2) \in \textbf{\textit{substituable}} \Leftrightarrow \exists a \in \textbf{\textit{C}} : c1 \textbf{\textit{ matérialise }} a \wedge c2 \textbf{\textit{ matérialise }} a$$

Nous pouvons montrer (à partir des propriétés énoncées de la relation **matérialise**) que la fermeture réflexive de la relation **substituable** est une relation d'équivalence et elle définit un ensemble de classes d'équivalence **C/substituable**$^{=}$. Chaque classe d'équivalence représente un groupe d'équivalence dans ApAM. Pour formaliser ce rapport, nous définissons les fonctions **groupe** et **membres** :

$$\textbf{\textit{groupe}} : \textbf{\textit{C}} \rightarrow \textbf{\textit{C}} \stackrel{\text{def}}{=} c \mapsto \begin{cases} a, & si\ \exists! a \in \textbf{\textit{C}} : (c\ \textbf{\textit{matérialise}}\ a) \\ c\ , & sinon \end{cases}$$
$$\textbf{\textit{membres}} : \textbf{\textit{C}} \rightarrow P(\textbf{\textit{C}}) \stackrel{\text{def}}{=} a \mapsto \{c \mid c \in \textbf{\textit{C}} \wedge c \textbf{\textit{ matérialise }} a\}$$

Nous pouvons montrer que deux composants appartiennent à la même classe d'équivalence si et seulement s'ils ont le même groupe. Nous pouvons également montrer que tous les membres d'un groupe appartiennent à la même classe d'équivalence. Autrement dit :

$$\forall\ c1,c2 \in \mathrm{C} : (\textbf{\textit{groupe}}(c1) = \textbf{\textit{groupe}}(c2) \Leftrightarrow (\exists g \in \textbf{\textit{C/substituable}}^{=} : c1 \in g \wedge c2 \in g))$$
$$\forall a \in \textbf{\textit{C}} : \exists!\ g \in \textbf{\textit{C/substituable}}^{=} : \forall\ c1,c2 \in \textbf{\textit{membres}}(\mathrm{a}) : c1 \in g \wedge c1 \in g$$

Notez que dans cette formalisation un groupe est toujours défini par un composant qui abstrait tous les membres du groupe et que l'on nomme informellement le représentant du groupe. Nous pouvons clarifier cette notion informelle en notant que :

$$\forall\ g \in \textbf{\textit{C/substituable}}^{=} : \exists! a \in \textbf{\textit{C}} : \forall c \in g : \textbf{\textit{groupe}}(c) = a$$

Par abus de langage dans le texte, quand nous parlons de groupe nous pouvons faire référence (en fonction du contexte) soit à l'ensemble de ses membres, soit à son représentant, soit à la classe d'équivalence associée.

Une spécification est l'abstraction d'un ensemble d'implémentations. Elle définit le groupe de spécifications, où la spécification est le représentant du groupe et les implémentations représentent les membres du groupe. Une implémentation est à son tour l'abstraction d'un ensemble d'instances : on parle alors de groupe d'implémentation. Dans ce groupe, l'implémentation est le représentant du groupe et l'ensemble des instances est les membres de ce groupe.

Le concept de groupe est assez générique pour pouvoir définir diverses taxonomies de composants. Dans ApAM nous proposons additionnellement une structuration plus spécifique et adaptée aux applications visées par une hiérarchie à trois niveaux d'abstraction : spécification, implémentation et instance.

Pour formaliser ces concepts nous définissons une partition de l'ensemble de composants :

$$\textit{Composant} = \textit{Spécification} \cup \textit{Implémentation} \cup \textit{Instance}$$

Toute architecture A=*(*C,***matérialise***)* d'une application ApAM doit respecter les contraintes suivantes du graphe A :

$$\forall s \in \boldsymbol{C} : s \in \textit{Spécification} \Rightarrow \textit{membres}(s) \subseteq \textit{Implémentation}$$
$$\forall i \in \boldsymbol{C} : i \in \textit{Implémentation} \Rightarrow \textit{membres}(i) \subseteq \textit{Instance}$$
$$\forall \iota \in \boldsymbol{C} : \iota \in \textit{Instance} \Rightarrow \textit{membres}(\iota) = \emptyset$$

$$\forall \iota \in C : \iota \in \textit{Instance} \Rightarrow \textit{groupe}(\iota) \in \textit{Implémentation}$$
$$\forall i \in C : i \in \textit{Implémentation} \Rightarrow \textit{groupe}(i) \in \textit{Spécification}$$
$$\forall s \in C : s \in \textit{Spécification} \Rightarrow \textit{groupe}(s) = s$$

Avec ces restrictions, nous pouvons définir les fonctions suivantes :

$$\textit{spécification-de} : \textit{Implémentation} \rightarrow \textit{Spécification} \stackrel{\text{def}}{=} i \mapsto \textit{groupe}(i)$$
$$\textit{implémentation-de} : \textit{Instance} \rightarrow \textit{Implémentation} \stackrel{\text{def}}{=} \iota \mapsto \textit{groupe}(\iota)$$

Le représentant du groupe est considéré comme le type des membres du groupe. Un membre est considéré comme une instance du représentant, il hérite alors de toutes les propriétés de son représentant. Un membre peut être le représentant d'un groupe, il devient alors une instance et un type en même temps.

Pour résumer :

- Une spécification est à la fois une instance et le type d'un ensemble d'implémentations.
- Une implémentation est à la fois une instance du type spécification et le type d'un ensemble d'instances (c'est une instance et un type en même temps).
- Finalement, une instance est une instance d'implémentation.

Ceci a pour conséquence que les dépendances et les propriétés définies au niveau du représentant seront communes à tous les membres des groupes. Cependant chaque niveau d'abstraction peut déclarer de nouvelles dépendances et de nouvelles propriétés qui lui seront spécifiques ainsi qu'aux membres du groupe qu'il représente.

La relation entre un représentant et les membres de son groupe est validée par le mécanisme de conformité. C'est ce mécanisme qui, dès la compilation de l'application, permet de vérifier statiquement que l'héritage est bien respecté entre les niveaux d'abstractions. Cette tâche est confiée au compilateur ApAM. Ce dernier est chargé de maintenir la conformité (voir §4.2 ci-dessous) entre les niveaux d'abstractions pendant les phases précédant l'exécution de l'application. La vérification est réalisée aussi lors de l'exécution par ApAM.

L'architecture de référence des applications ubiquitaires définie dans ApAM s'exprime dans ces trois niveaux d'abstraction qui seront détaillés par la suite.

4.1.1. SPÉCIFICATION

C'est le plus haut niveau d'abstraction dans notre modèle à composants. L'architecture de référence de l'application peut être décrite dès ce niveau d'abstraction. Ce niveau permet de décrire l'application indépendamment des technologies d'implémentation.

Dans des approches classiques, où une spécification est représentée par une interface, les spécifications ne peuvent pas exprimer de dépendances, elles ne sont pas composabls e[Perr96]. Les applications conçues via ces approches ne peuvent donc ni décrire leurs besoins à ce niveau d'abstraction, ni définir d'architectures abstraites.

Notre approche permet de définir des dépendances à partir de la spécification de services. Les dépendances décrites au niveau de la spécification permettent de définir des compositions structurelles de l'application constituées de spécifications. Comme le montre la

Figure 22, un composant peut définir une dépendance vers une spécification. Comme les spécifications sont le niveau d'abstraction le plus haut, l'utilisation de la spécification pour décrire la dépendance va permettre de sélectionner à l'exécution de l'application des composants dans l'ensemble des membres du groupe d'équivalence représenté par cette spécification. Ensuite, le composant sélectionné pourra être lui-même substitué par n'importe quel membre de son groupe d'équivalence qui satisfait les contraintes et les préférences décrites dans la dépendance. L'architecture de référence exprimée à ce niveau gagne en flexibilité car l'espace de choix pour sélectionner des composants est vaste et maitrisé. Le processus de concrétisation des composants est réalisé par le mécanisme de résolution (voir §5.1 ci-dessous) qui va remplacer (résoudre) le composant abstrait en un composant exécutable (plus concret) en passant au travers de deux niveaux d'abstraction (implémentation puis instance).

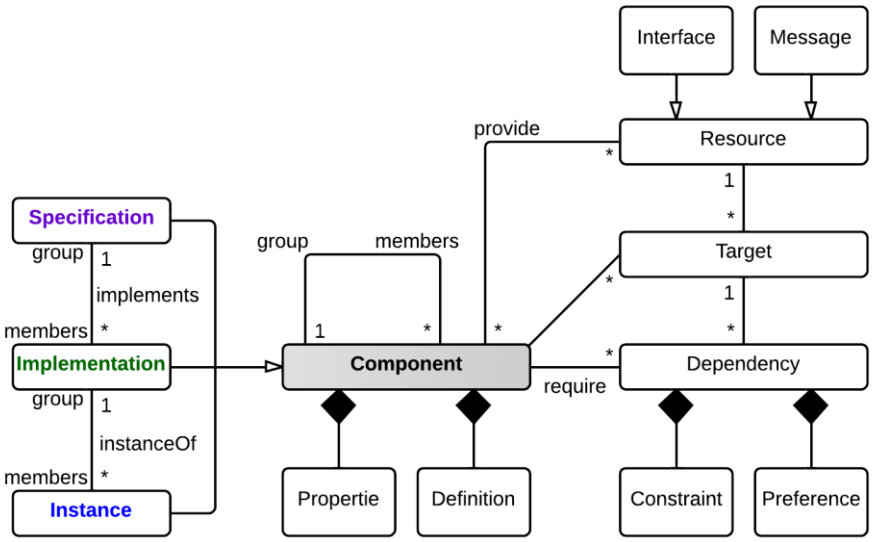


Figure 22 : Les niveaux d'abstraction

Une spécification peut décrire des contraintes et des préférences pour ces dépendances. Ces contraintes et préférence sont propagées vers les niveaux d'abstraction plus bas. Grâce à notre mécanisme de conformité (voir §4.2 ci-dessous), tous les niveaux d'abstraction plus bas vont hériter de ces dépendances et de leurs contraintes. Chaque niveau peut alors enrichir la dépendance par des contraintes supplémentaires pour préciser la dépendance.

Les dépendances de spécification sont aussi utilisées pour définir des dépendances complexes entre deux composants pour éviter les dépendances ambiguës. ApAM utilise le concept de spécification pour grouper les ressources fournies et requises. Ceci permet de voir un ensemble de dépendances comme une dépendance atomique, ce qui facilite la gestion des liaisons. Les dépendances de spécification forcent les dépendances de ressources à être satisfaites par le même fournisseur, sans forcément mentionner nominativement l'implémentation ou l'instance de composant.

Pour illustrer le concept de spécification, nous reprenons le scénario défini précédemment. Dans l'application « *Home Light Follow Me* », nous pouvons définir la spécification qui correspond au gestionnaire de luminaires, ce gestionnaire peut avoir différentes implémentations. Ainsi, cette spécification :

- Définit les ressources qu'elle fournit (une interface *LightControllerService*).
- Spécifie des définitions de propriétés avec les attributs *provider, location.*
- Déclare les ressources requises à travers la dépendance *LightDependency*.

En pratique, la spécification est décrite dans un fichier XML, telle que :

```
<specification name="LightController"
interfaces="home.services.light.LightControllerService">
     <definition name ="provider" type="string" />
     <definition name ="location" type="{kitchen, bathroom, livingroom, bedroom}" />
     <dependency specification="Light" id="LightDependency" />
</specification>
```

Une spécification est une description abstraite complète (elle décrit les caractéristiques, les ressources fournies et requises) du contrat de service. La spécification peut être fournie par aucune ou plusieurs implémentations. Ces implémentations doivent satisfaire le contrat de service.

L'architecture de référence dans ApAM peut être décrite à partir de ce niveau d'abstraction. Elle peut aussi être décrite en réalisant un assemblage de composant qui mélange les trois niveaux d'abstraction (spécification, implémentation, instance). Ainsi, il est possible de définir des dépendances entre des composants qui ne sont pas dans le même niveau d'abstraction. Plus l'architecture de référence est exprimée dans un niveau d'abstraction bas, plus l'espace de choix pour la sélection de composant concret sera contraint et limité (à la limite, il n'y a aucun choix si toutes les instances sont spécifiées statiquement). Inversement, plus l'architecture de référence de l'application est décrite dans un niveau d'abstraction élevé, plus l'application sera flexible et résiliente au contexte d'exécution.

La décomposition du modèle à composants sur trois niveaux d'abstraction et la vérification statique de l'architecture permettent de couvrir l'ensemble du cycle de vie de l'application (conception, développement, compilation, packaging, exécution). En effet, chaque niveau d'abstraction permet de décrire les besoins associés à chaque niveau du cycle de vie. La compilation de l'architecture achève ce processus en vérifiant la cohérence des composants et de l'assemblage de l'application et réalise le packaging de l'application. Grâce à notre modèle à composants et au niveau d'abstraction, l'architecture de référence regroupe tous les besoins décrits durant l'ensemble du cycle de vie de l'application. Ces informations sont ramenées à la plate-forme d'exécution ApAM. Ils vont permettre à ApAM de construire, d'exécuter et d'adapter l'application.

4.1.2. IMPLÉMENTATION

Ce concept existe dans la plupart des approches à composants orienté services. Une implémentation représente l'unité exécutable (souvent une classe Java). L'implémentation a des propriétés et peut fournir et/ou requérir des ressources. Dans ApAM, une implémentation doit définir une classe Java qui implémente (dans le sens Java) les interfaces ou doit déclarer des méthodes ou des attributs pour échanger les messages décrits dans le contrat de service (sa spécification). Fournir toutes les ressources de sa spécification est l'une des contraintes qu'une l'implémentation doit satisfaire pour être valide.

L'aspect original de l'approche ApAM est que les implémentations dépendent des composants ou des ressources décrits dans la spécification, mais elles peuvent également dépendre des composants ou des ressources propres à l'implémentation. Dans ApAM, les dépendances décrites par la spécification sont précisées par les implémentations. ApAM utilise un mécanisme d'injection pour gérer les dépendances. Chaque dépendance de l'implémentation est référencée (par un attribut ou une méthode) dans le code source (la classe Java). À l'exécution, les champs sont injectés avec des objets obtenus à partir de la résolution (voir §5.1.2 ci-dessous).

Pour illustrer le concept d'implémentation, nous reprenons le scénario de l'application « *Home Light Follow Me* ». Différentes implémentations de la spécification LightController peuvent être réalisées. Par exemple, le LIG définit sa propre implémentation de LightController. Cette implémentation est liée à la classe Java *LIGLightService*, elle :

- Définit une nouvelle ressource qui va être fournie par le composant *LightAlarmService*, en plus de celle décrite dans la spécification *LightControllerService*. Il n'est pas nécessaire de répéter toutes les interfaces.
- Donne la valeur « LIG » à la propriété « *provider* ».
- Attache la propriété « *location* » au champ « *myLocation* » de la classe d'implémentation.

En pratique, l'implémentation est décrite dans un fichier XML, telle que :

```
<implementation name="LIGLightController"
                              classname="fr.liglab.services.light.LIGLightService"
                              interfaces="home.services.alarm.LightAlarmService"
                              specification="LightController" >
      <property name ="provider" value="LIG" />
      <definition name ="location" field ="myLocation" type="{kitchen, bathroom,
livingroom, bedroom}" />
      <dependency specification="Light" id="LightDependency">
            <interface field="lights"/>
            <message push="notifyLightStatus"/>
           <constraints>
               <instance filter="($this.location=$.LightDependency.$location)">
           </constraints>
         <preferences>
               <implementation filter="(provider=LIG)">
         </preferences>
 <dependency/>
</implementation>
```

Comme le montre la Figure 22, les implémentations sont liées aux spécifications à travers la relation « *implements* ». Ce lien décrit la conformité entre la spécification et l'implémentation. Ce lien permet à l'implémentation de raffiner la spécification ainsi que les dépendances en rajoutant des contraintes ou des préférences additionnelles de sélection. Grâce au mécanisme de groupes, la validité d'une implémentation (i.e., sa conformité et sa cohérence vis-à-vis de sa spécification) peut être assurée statiquement, lors de la compilation du composant (voir §4.2 ci-dessous).

L'implémentation représente le lien entre l'architecture la plus abstraite (les spécifications) et l'architecture la plus concrète (les instances). En effet, si le passage entre la spécification et l'implémentation se fait seulement d'une manière statique et déclarative, la nature de l'implémentation (classe Java) permet à ApAM de créer des instances automatiquement à partir de l'implémentation et cela dynamiquement à l'exécution, mais aussi de manière statique en décrivant l'instance à créer explicitement.

4.1.3. INSTANCE

Les instances sont les entités les plus concrètes des composants. Elles représentent les entités en exécution. Une instance est reliée à une et une seule implémentation. Elle hérite de toutes les propriétés et les dépendances de son implémentation associée et par conséquent de ceux de la spécification.

Dans ApAM, les instances peuvent être créées dynamiquement. Suite à une demande de résolution, il n'est donc pas nécessaire de les déclarer. Cependant, pour préciser une configuration ou pour raffiner le composant et ses dépendances, il est possible de spécifier statiquement l'instance que nous souhaitons rendre disponible dans la plate-forme ApAM. Cette instance est alors créée automatiquement et rendue disponible dès le démarrage de l'application.

Pour illustrer le concept d'instance, nous reprenons le scénario *« Home Light Follow Me »*, dans lequel nous souhaitons rendre disponible une instance de l'implémentation *LIGLightService*. Cette instance peut :

- Donner des valeurs aux définitions de propriétés spécifiées dans son implémentation et sa spécification.
- Rajouter des nouvelles contraintes et préférence pour les différentes dépendances déclarées.

```
<instance name="LIG-Kitchen-Light-Controller" implementaion="LIGLightController" >
     <property name ="location" value="kitchen" />
     <dependency specification="Light" id="LightDependency">
         <constraints>
               <instance filter="(provider=LIG)">
         </constraints>
      </dependency>
  </instance>
```

Dans notre approche, l'exécution d'une l'application est représentée par les instances et leurs connexions (wires). ApAM connecte une instance cliente à une instance fournisseur si, suite à une demande de résolution de l'instance cliente, l'instance fournisseur d'une part satisfait la dépendance et d'autre part est visible par l'instance client (voir §7 ci-dessous). Cette connexion, que nous appelons **wire,** est créée en réponse à une résolution de la dépendance réussie. La connexion entre instances est maintenue tant qu'aucun changement de contexte d'exécution n'affecte les deux instances ou leurs propriétés. En effet, ApAM détruit les connexions entre les composants si les contraintes de dépendances ne sont plus satisfaites, puis elle attend une nouvelle demande de résolution pour sélectionner le composant adéquat.

Un wire n'est pas une déclaration de dépendance, mais une connexion effective entre deux instances à un instant donné de l'exécution de l'application. Il s'agit d'une des concrétisations possibles de la dépendance :

Nous désignons par wires l'ensemble de tous les wires :

$$\textbf{\textit{wire}} = \{(source, id, cible) \mid source \in \textbf{\textit{Instance}} \wedge\ id \in \mathbb{D}\ \times cible \in \textbf{\textit{Instance}}\}$$

Un wire est valide s'il est défini par une dépendance et s'il satisfait les contraintes de la dépendance.

Pour formaliser cette notion nous définissons une fonction qui évalue l'ensemble de propositions qu'un wire doit satisfaire pour être valide par rapport à une définition de dépendance :

$$\omega\text{-}\textbf{\textit{valide}} : (\mathbb{D} \times ((\mathbb{R} \cup \mathbb{C}) \times \mathrm{P}(\mathbb{E}))) \times \textbf{\textit{wire}} \rightarrow \mathbb{B}\ ^{11}$$

$$((idd, requis, \text{contraintes}), (source, idw, cible)) \mapsto$$
$$\text{idd} = idw \wedge$$
$$requis \in \mathbb{R} \Rightarrow requis \in \textbf{\textit{ressources}}(cible) \wedge$$
$$requis \in \mathbb{C} \Rightarrow \exists gc \in \textbf{\textit{Composant}} : \textbf{\textit{id}}(gc) = requis \wedge cible\ \textbf{\textit{matérialise*}}\ gc \wedge$$
$$\forall c \in contraintes : [\![c]\!](\textbf{\textit{propriétés}}(cible))$$

L'état courant de l'exécution d'une application est décrit par son architecture concrète. Une architecture concrète est un graphe dont les nœuds sont des instances et les arcs des relations wire. L'architecture concrète dans ApAM représente une concrétisation de l'architecture de référence en exécution sur la plate-forme. Nous définissons l'état de l'exécution E comme :

[11] On dénote par $\mathbb{B}$ l'ensemble booléen {true,false}

$$\textit{Etat} \stackrel{\text{def}}{=} \{ (I, \omega) \mid I \subseteq \textit{Instance} \wedge \omega \subseteq \textit{wire} \}$$

Un état d'exécution est dit valide par rapport à son architecture si toutes ses instances sont des concrétisations d'un composant de l'architecture et tous les wires sont valides par rapport à une définition de dépendance de l'architecture. En appliquant la définition de membre et de conformité, nous pouvons déduire qu'il suffit à un wire d'être valide par rapport à la définition de dépendance la plus proche (ou la plus raffiné) pour satisfaire l'ensemble des définitions imbriquées.

Pour formaliser cette notion nous définissons une fonction qui évalue l'ensemble de propositions qu'un état E doit satisfaire pour être valide par rapport à son architecture A :

$$\textbf{valide} : (\textit{Architecture} \times \textit{Etat}) \rightarrow \mathbb{B}$$

$$((C, \textit{matérialise}), (I, \omega)) \mapsto$$
$$\forall (\textit{source}, \textit{id}, \textit{cible}) \in \omega : \textit{source} \in I \wedge \text{cible} \in I \wedge$$
$$\exists \textit{gs}, \textit{gc} \in C : \textit{source}\ \textit{matérialise}^*\ \textit{gs} \wedge \textit{cible}\ \textit{matérialise}^*\ \textit{gc} \wedge$$
$$\exists d \in \Delta\text{-dépendances}(\textit{gs}) : \omega\text{-valide}(d, (\textit{source}, \textit{id}, \textit{cible}))$$

Chaque dépendance peut être une abstraction d'un grand nombre de dépendances et de wires. Une architecture de référence peut être l'abstraction d'un grand nombre d'architectures concrètes et par conséquent d'un grand nombre d'exécutions valides. Nous avons alors les propriétés suivantes :

- L'architecture la plus abstraire est celle qui englobe le plus grand nombre d'architectures valides et par conséquence d'architectures concrètes valides.
- L'architecture peut être définie et vérifiée à la compilation même si les composants et les contraintes des niveaux d'abstraction plus bas sont inconnues.
- Quel que soit le niveau d'abstraction (spécification, implémentation, instance), il est possible d'exécuter une application et de prouver qu'elle est valide par rapport à l'architecture de référence.

Pour nous, contrôler l'exécution d'une application signifie résoudre les wires, au moins lorsqu'ils sont requis.

À l'exécution, nous retrouvons les concepts de notre modèle à composants (spécification, implémentation, instance). ApAM offre un modèle réflexif qui matérialise l'application en exécution, ce qui permet d'adapter l'architecture concrète de l'application.

Les instances ApAM ont un cycle de vie très simple. Une fois démarrée dans ApAM une instance est toujours valide et fournit ses ressources indépendamment de la résolution de ses dépendances. De plus, pour ne pas surcharger le système avec des demandes de résolution de dépendances, les dépendances ne sont résolues que lorsqu'elles sont requises par le composant et non à chaque changement de contexte. ApAM adopte le mécanisme de la résolution paresseuse (voir §5.1.1 ci-dessous). Grâce à ce mécanisme, les demandes de résolutions sont moins fréquentes et des modifications de dépendance inutiles sont évitées. L'état courant de l'application ApAM est presque toujours un état « partiel » ; partiel car les composants et les dépendances ne sont pas résolus immédiatement et peuvent rester toujours non résolus. L'architecture concrète de l'application représentée par l'état courant évolue selon le déroulement de l'exécution. Ainsi, des dépendances peuvent être résolues et d'autre peuvent rester toujours abstraites car elles ne sont jamais utilisées (principe de Pareto, ou principe des 80/20 [Koch98]).

Pour adapter une architecture abstraite, ApAM obéit aux règles de conformités. Ces règles permettent à ApAM de construire une partie de l'application, de substituer des composants ou d'adapter l'architecture tout en restant conforme aux besoins et à la description de l'architecture de référence. Ce mécanisme de conformité sera décri dans le paragraphe suivant.

4.2. DÉFINITION DE LA CONFORMITÉ ENTRE LES NIVEAUX D'ABSTRACTION

En introduisant des niveaux d'abstraction dans le composant, il faut définir un lien cohérent entre les niveaux d'abstraction. Ce lien permet de vérifier la cohérence de l'application et assure qu'il n'y ait pas de discontinuité, de perte d'information, ou de modification sémantique dans le processus de définition de l'application. Le concept de groupe d'équivalence nous permet de maintenir la conformité dans notre modèle à composant.

Il existe trois niveaux d'abstraction de composant dans ApAM : spécification, implémentation et instance. En pratique, les spécifications sont des composants qui décrivent les ressources communes, comme les interfaces ou les messages que doivent fournir tous les membres. Les implémentations sont des composants. Il s'agit de classes annotées qui implémentent les ressources fournies. Finalement, les instances représentent l'instanciation d'une implémentation associée.

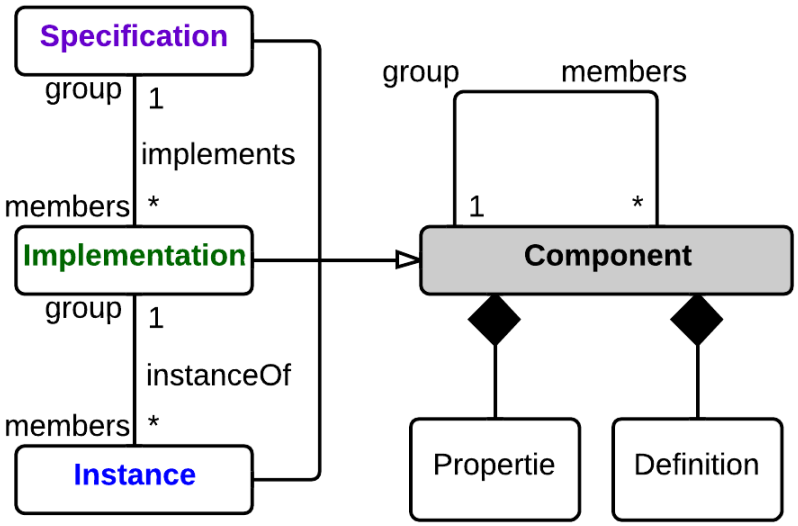


Figure 23 : Les définitions et les propriétés du groupe

La conformité est portée dans notre modèle à composants par les relations « implements » et « instanceOf » (voir Figure 23). Elles décrivent le lien entre un niveau d'abstraction et un autre, ou plutôt entre un représentant de groupe et ses membres. La relation « implements » relie une implémentation à sa spécification. Elle permet de s'assurer de la cohérence entre les deux concepts. De la même manière, la relation « instanceOf » décrit le lien de conformité entre l'instance et son implémentation. Les relations de conformité définies par les groupes d'équivalence, imposent aux membres d'un groupe d'être conformes avec la partie commune définie par leur représentant. Cette partie commune est définie par les ressources fournies, les ressources requises et les propriétés.

Lors du passage d'un niveau d'abstraction à un autre, il est possible de raffiner les dépendances. Par exemple, les implémentations peuvent raffiner les dépendances des spécifications. Le raffinement des dépendances permet de préciser l'ensemble des composants qui peuvent satisfaire la dépendance.

Pour formaliser ce concept nous définissons une fonction qui évalue les conditions qu'une définition de dépendance doit satisfaire pour être un raffinement d'une deuxième définition.

$$\textbf{\textit{Δ-raffine}} : (\mathbb{D} \times ((\mathbb{R} \cup \mathbb{C}) \times P(\mathbb{E}))) \times (\mathbb{D} \times ((\mathbb{R} \cup \mathbb{C}) \times P(\mathbb{E}))) \rightarrow \mathbb{B}$$

$$((\textit{id-raffinement}, \textit{requis-raffiné}, \textit{contraintes-raffinées}) , (\textit{id}, \textit{requis}, \textit{contraintes})) \mapsto$$

$$\textit{id} = \textit{id-raffinement}$$
$$\wedge$$
$$\textit{contraintes} \subseteq \textit{contraintes-raffinées}$$
$$\wedge$$
$$\textit{requis} \in \mathbb{R} \wedge \textit{requis-raffiné} \in \mathbb{R} \Rightarrow \textit{requis} = \textit{requis-raffiné}$$
$$\wedge$$
$$\textit{requis} \in \mathbb{C} \wedge \textit{requis-raffiné} \in \mathbb{C} \Rightarrow \exists g, gr \in \textbf{\textit{Composant}} : \textbf{\textit{id}}(g) = \textit{requis} \wedge \textbf{\textit{id}}(gr) = \textit{requis-raffiné} \wedge gr\ \textbf{\textit{matérialise}}^*\ g$$
$$\wedge$$
$$\textit{requis} \in \mathbb{R} \wedge \textit{requis-raffiné} \in \mathbb{C} \Rightarrow \exists gr \in \textbf{\textit{Composant}} : \textbf{\textit{id}}(gr) = \textit{requis-raffiné} \wedge \textit{requis} \in \textbf{\textit{ressources}}(gr)$$
$$\wedge$$
$$\textit{requis} \in \mathbb{C} \wedge \textit{requis-raffiné} \in \mathbb{R} \Rightarrow \textit{false}$$

Les membres d'un groupe doivent fournir au minimum les ressources et les propriétés fournies par le représentant et doivent avoir des dépendances conformes. Les membres peuvent instancier des propriétés définies par le groupe, fournir de nouvelles ressources et avoir de nouvelles dépendances. Nous pouvons formaliser la notion de conformité à partir des définitions précédentes :

$$\textbf{\textit{conforme}} : (\textbf{\textit{Composant}} \times \textbf{\textit{Composant}}) \rightarrow \mathbb{B}$$

$$(c, a) \mapsto$$
$$\textbf{\textit{ressources}}(a) \subseteq \textbf{\textit{ressources}}(c)$$
$$\wedge$$
$$\textbf{\textit{propriétés}}(a) \subseteq \textbf{\textit{propriétés}}(c)$$
$$\wedge$$
$$\textbf{\textit{Δ -propriétés}}(a) \subseteq \textbf{\textit{Δ -propriétés}}\ (c)$$
$$\wedge$$
$$\forall da \in \textbf{\textit{Δ -dépendances}}(a) : \exists dc \in \textbf{\textit{Δ-dépendances}}(c) : dc\ \textbf{\textit{Δ-raffine}}\ da$$
$$\wedge$$
$$\forall (att, value) \in \textbf{\textit{propriétés}}(c) \setminus \textbf{\textit{propriétés}}(a) :$$
$$\exists type \in \mathbb{T} : \textbf{\textit{type}}(value) = type \wedge (att, type) \in \textbf{\textit{Δ-propriétés}}\ (c) \wedge$$
$$\neg \exists value2 \in \mathbb{V} : value \neq value2 \wedge (att, value2) \in \textbf{\textit{propriétés}}(a)$$

À tour de rôle, chaque membre peut être un groupe et avoir ses propres membres, jusqu'à atteindre le niveau le plus bas : les instances des composants.

Finalement, nous pouvons synthétiser tous les critères de cohérence d'une architecture (qui sont mis en œuvre par le compilateur ApAM) :

$$\textbf{\textit{cohérente}} : (\textbf{\textit{Architecture}}) \rightarrow \mathbb{B}$$

$$\textbf{\textit{(C, matérialise)}} \mapsto$$

$$\forall (c,a) \in \textbf{\textit{matérialise}} : c \in \textbf{\textit{C}} \wedge a \in \textbf{\textit{C}}$$
$$\wedge$$
$$\forall c \in \textbf{\textit{C}} : (c, c) \notin \textbf{\textit{matérialise}}^+$$

$$\wedge$$
$$\forall c \in \boldsymbol{C}: |\{a \mid c \ \boldsymbol{matérialise}\ a\}| \leq 1$$
$$\wedge$$
$$\forall s \in \boldsymbol{C} \cap \boldsymbol{Spécification}: \boldsymbol{membres}(s) \subseteq \boldsymbol{Implémentation} \wedge \boldsymbol{groupe}(s) = s$$
$$\wedge$$
$$\forall i \in \boldsymbol{C} \cap \boldsymbol{Implémentation}: \boldsymbol{membres}(i) \subseteq \boldsymbol{Instance} \wedge \boldsymbol{groupe}(i) \in \boldsymbol{Spécification}$$
$$\wedge$$
$$\forall \iota \in \boldsymbol{C} \cap \boldsymbol{Instance}: \boldsymbol{membres}(\iota) = \emptyset \ \wedge \boldsymbol{groupe}(\iota) \in \boldsymbol{Implémentation}$$
$$\wedge$$
$$\forall (c,a) \in \boldsymbol{matérialise}^* \Rightarrow c \ \boldsymbol{conforme}\ a$$

L'introduction du concept de groupe renforce le typage des différentes abstractions de composants. Ainsi, les composants peuvent spécifier des définitions de propriétés pour les niveaux d'abstraction suivants. Les définitions sont des couples ***<attribut, type>***. La conformité s'assure que les niveaux d'abstractions suivants déclarent une propriété pour chaque définition déclarée avec une valeur correspondant au bon type. Pour illustrer la propagation des propriétés, nous proposons de reprendre l'exemple du scénario précédemment présentés « *Home Light Follow Me* ». Dans cet exemple, les dispositifs de lumières sont représentés par la spécification « Light ». Cette spécification peut avoir différentes implémentations (selon le constructeur par exemple) et aussi différentes instances qui réifient la présence réelle des dispositifs de type lumière dans la maison.

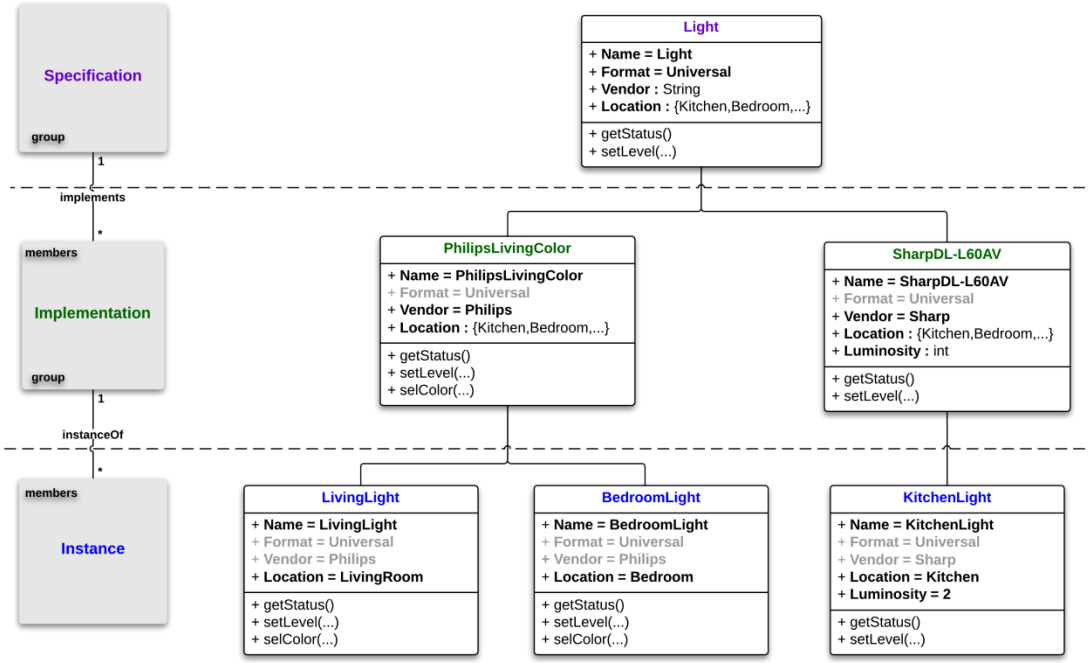


Figure 24 : Exemple de conformité

Ainsi, le représentant « *Light* » peut définir des attributs communs à tous ses membres du groupe et aussi des propriétés variables par lesquelles les membres peuvent être comparés et distingués :

- **Les propriétés communes** sont propagées du représentant à chacun de ses membres. Tous les éléments du groupes partagent alors les propriétés commune définies par le représentant (Exemple : Format = Universal » de la
- Figure 24).
- **Les propriétés variables :** il s'agit de définitions de propriétés que les membres doivent obligatoirement avoir et dont la valeur est fixée par le membre. Exemple : dans la

- **Figure** 24, les propriétés « Vendor » et « Location » sont spécifiées dans le représentant et chaque membre attribue une valeur spécifique à chaque propriété.

Les implémentations « *PhilipsLivingColor* » et « *SharpDL-L60AV* » sont les membres du groupe « *Light* ». Elles héritent des propriétés communes et variables définies par la spécification « *Light* ». Les deux implémentations sont équivalentes si on ne considère que les propriétés communes. Ce sont les propriétés variables qui permettent de les différentier. L'implémentation « *SharpDL-L60AV* » est aussi un représentant de groupe. Elle définit les propriétés communes (« *Vendor* ») mais aussi des propriétés variables (« *Luminosity* ») pour ces instances. Les instances héritent des propriétés de leur implémentation et par transitivité, de celles de leur spécification. Les instances « *LivingLight* » et « *BedroomLight* » sont équivalentes si on ne considère que les propriétés communes définies par leur représentant (« *PhilipsLivingColor* »), elles sont membres du même groupe « *PhilipsLivingColor* ».

Le concept de groupes d'équivalences et le mécanisme de conformité nous permettent de définir les relations entre les différents niveaux d'abstraction (voir Figure 23). Ce lien définit un mécanisme qui permet de passer d'un niveau d'abstraction à un autre ou de substituer un composant par un autre composant tout en restant conforme avec la description : c'est le mécanisme de résolution. La partie suivant détaillera ce mécanisme.

5. EXÉCUTER UNE APPLICATION APAM

L'architecture de référence d'une application est définie en utilisant le modèle à composants ApAM (voir Figure 22). L'objectif de la plate-forme est d'interpréter la description abstraite de l'application afin de l'exécuter. ApAM concrétise l'architecture définie dans les trois niveaux d'abstraction. Pour passer d'une architecture de référence à une architecture concrète ApAM utilise le mécanisme de résolution. Ce mécanisme permet le passage d'un composant abstrait vers un composant plus concret. Tout au long de cette partie nous détaillerons ce mécanisme afin d'expliquer la concrétisation de l'architecture.

Le rôle de la plate-forme d'exécution ApAM est de contrôler l'exécution d'applications. L'architecture concrète d'une application peut être vue comme un ensemble de composants connectés entre eux. Dans notre approche, contrôler l'exécution de l'application consiste à sélectionner les composants à utiliser ou encore à créer ou à détruire les connexions entre ces composants.

5.1. LA RÉSOLUTION

La résolution est le mécanisme qui permet de sélectionner ou de créer les membres d'un groupe. Dans ApAM, la résolution est une fonction qui prend en entrée le représentant d'un groupe (spécification ou implémentation) et qui retourne un ou plusieurs membres de ce groupe. Les membres retournés peuvent avoir été sélectionnés parmi les membres existants, avoir été déployés ou avoir été créés. La création des membres d'un groupe se produit lorsque le représentant du groupe est une implémentation. Ce représentant est une fabrique qui permet la création des membres de son groupe (instances), on parle alors d'instanciation des implémentations. Le mécanisme de résolution dans ApAM se produit en réponse à deux principaux événements :

- **L'activation d'une application**. Pour activer une application, il suffit de donner son nom (de la spécification, de l'implémentation ou de l'instance) qui est considéré comme une demande de résolution. Si un nom d'instance est donné, le système se contente d'activer l'instance en question Une activation à partir de la spécification (ou de l'implémentation) provoque la sélection ou la création de l'instance qui tient compte des contraintes du contexte d'exécution.
- **L'utilisation d'une dépendance de l'application**. Par défaut, ApAM adopte un comportement paresseux (qui est configurable), c'est-à-dire que les dépendances ne sont calculées que lorsqu'elles sont utilisées par le code métier de l'application : l'utilisation d'une dépendance provoque une demande de résolution. Comme pour l'activation d'une application, plus une dépendance est exprimée à haut niveau d'abstraction, plus le mécanisme de résolution aura le choix et pourra tenir compte du contexte d'exécution. Si les dépendances sont décrites en termes de ressources (des interfaces de service par exemple), le système recherche les implémentations qui fournissent ces ressources, puis il les résout pour obtenir des instances.

La résolution de dépendances s'exprime surtout au niveau des implémentations (les composants de type implémentation). Dans ce niveau d'abstraction des champs sont déclarés dans le code source. Ces champs vont contenir l'adresse du composant destination (voir §5.1.2 ci-dessous). À l'instanciation de l'implémentation, les champs ne sont pas forcément initialisés, on dit alors que le wire n'est pas résolu. Une demande de réalisation de wire est déclenchée si au cours de l'exécution l'un des champs liés à une déclaration de dépendances est sollicité. Si le wire n'est pas résolu, la plate-forme ApAM tente de résoudre la dépendance et résout le wire de **façon transparente**.

Le processus de **résolution de dépendances** est le mécanisme de base pour expliquer l'exécution d'une application ApAM en termes de l'évolution de son architecture. Une résolution a deux effets principaux sur l'application. D'une part, certains composants abstraits vont être concrétisés (possible par le déploiement ou par l'instanciation de nouveaux membres du groupe d'équivalence). D'autre part, l'état de l'application change pour satisfaire la demande de réalisation de wire.

Pour modéliser ce concept, nous commençons par formaliser les critères qui nous permettent de déterminer dans quels cas une architecture (dite concrète) **raffine** une autre architecture (dite abstraite) :

$$\textit{raffine} : (\textit{Architecture} \times \textit{Architecture}) \rightarrow \mathbb{B}$$

$$((C, \textit{matérialise}), (C^a, \textit{matérialise}^a)) \mapsto$$
$$C^a \subseteq C \wedge \textit{matérialise}^a \subseteq \textit{matérialise} \wedge \textit{cohérente}(C^a, \textit{matérialise}^a) \Rightarrow \textit{cohérente}(C, \textit{matérialise})$$

Ensuite, nous formalisons une demande de réalisation de wire par l'évènement déclencheur :

$$\textit{?-résolution} = \{ (\textit{source}, \textit{dépendance}) \mid \textit{source} \in \textit{Instance} \wedge \textit{dépendance} \in \mathbb{D} \}$$

Finalement, pour formaliser la résolution d'une dépendance nous définissons les critères qui nous permettent de déterminer si un nouvel état, avec son architecture correspondante, satisfait une demande de résolution :

$$\textit{résolution-valide} : (\textit{Architecture} \times \textit{Etat}) \times \textit{?-résolution} \times (\textit{Architecture} \times \textit{Etat}) \rightarrow \mathbb{B}$$

$$((A, (I, \omega)), (\textit{source}, \textit{id}), (A', (I', \omega'))) \mapsto$$

$$\textit{valide}(A,(I, \omega)) \wedge \textit{source} \in I$$
$$\wedge$$
$$\textit{raffine}(A', A) \wedge \textit{valide}(A', (I', \omega'))$$
$$\wedge$$
$$I \subseteq I' \wedge \omega \subseteq \omega' \wedge \exists \textit{cible} \in I' : (\textit{source}, \textit{id}, \textit{cible}) \in \omega'$$

À noter que cette formalisation n'exprime que la condition de validité du processus de résolution. Les mécanismes, pour choisir une architecture concrète parmi toutes les alternatives possibles, sont complexes et doivent tenir compte du contexte d'exécution de l'application (voir §5.2.1 ci-dessous). Aussi, l'échec de la résolution peut entrainer une reconfiguration plus complexe de l'architecture (voir §5.2.4 ci-dessous. La résolution de dépendances n'est pas le seul facteur déclencheur d'une reconfiguration, l'architecture peut changer en réponse à d'autres évolutions du contexte (exemples : apparition d'un dispositif ou indisponibilité d'un service).

Résoudre une dépendance consiste donc à créer un wire ***w*** à partir de la définition de la dépendance la plus concrète ***d=<id, T, C>***. Si ***T*** est un composant, créer ***w*** consiste à trouver l'instance descendante de ***T*** qui satisfait les contraintes ***C***. En effet, la résolution peut être restreinte en spécifiant les propriétés que les membres sélectionnés ou créés doivent satisfaire. Quand les contraintes sont exprimées sur les propriétés variables définies par le représentant, ApAM assure leur validité statiquement. Pour cela, la résolution est déclenchée dès la phase de compilation. Une vérification est alors réalisée par rapport aux propriétés définies par le représentant du groupe. La résolution peut être contrainte à tous les niveaux d'abstraction. Chaque contrainte définie dans un niveau d'abstraction peut être raffinée et précisée par les niveaux d'abstractions suivant. Ces contraintes sont séparées en deux catégories dans ApAM (voir §2.3 ci-dessus). Les contraintes doivent impérativement être respectées alors que les préférences sont des contraintes moins discriminantes. Les membres qui ne satisfont pas les préférences sont placés en dernier dans la liste des solutions. ApAM fait alors le choix de les utiliser dans le cas où aucun membre ne satisfait les préférences.

Une dépendance est définie par son nom, ses contraintes et ses préférences (voir Figure 22). Ainsi, quand une dépendance est définie vers :

- Une **implémentation** : la résolution consiste à trouver en premier l'implémentation, puis à sélectionner une de ses instances qui satisfont les contraintes et les préférences.

- Une **spécification** : les implémentations de cette spécification qui satisfont les contraintes sont d'abord sélectionnées. Le processus de résolution est ensuite réitéré sur cet ensemble d'implémentations pour trouver l'instance la mieux adaptée.
- Une **ressource** : la résolution consiste à trouver les composants qui fournissent cette ressource et qui satisfont les contraintes et les préférences. Le processus se répète alors récursivement jusqu'à trouver l'instance.

Si aucune instance n'est trouvée pendant la phase de recherche, une instance est alors créée à partir d'une implémentation sélectionnée. Plus loin dans le chapitre (voir §5.2) nous verrons qu'il est possible d'aller au-delà des implémentations disponibles dans le contexte d'exécution pendant la résolution et nous expliquerons les comportements en cas d'échec de la résolution.

5.1.1. LES TYPES DE RÉSOLUTION

À la compilation, la résolution se déclenche automatiquement afin de vérifier la cohérence des différents niveaux d'abstraction d'un composant. À l'exécution, ce déclenchement est configurable. Le développeur peut ainsi choisir le type de résolution qu'il veut pour chaque composant et/ou dépendance.

Dans la plate-forme ApAM, il existe trois moyens pour déclencher la résolution à l'exécution :

- **La résolution immédiate (Eager)** qui se déclenche lors de l'activation d'une application. Elle peut être spécifiée sur n'importe quelle dépendance de composant. Le mécanisme de résolution se produit alors automatiquement lors du déploiement du composant, pour toutes les dépendances annotées par cette propriété.
- **La résolution paresseuse (Lazy)** qui est le comportement par défaut adopté par notre plate-forme d'exécution. Contrairement à la résolution immédiate, elle ne se déclenche que pour les dépendances sollicitées au cours de l'exécution de l'application.
- **La résolution dynamique ou adaptative (Dynamic)** qui se déclenche en réaction à l'apparition d'un composant. Dans ce type de résolution l'initiative de la demande de résolution est inversée : la résolution est réalisée si un composant apparaît dynamiquement dans la plate-forme et s'il peut satisfaire une dépendance non résolue.

5.1.2. INJECTION DE DÉPENDANCES

Notre modèle repose sur la liaison entre la description du composant de type implémentation et la classe d'implémentation Java de cette implémentation. Pour permettre au composant de créer des liaisons et pour pouvoir superviser cette création, notre composant est placé dans un conteneur. Ce dernier permet à notre plate-forme d'exécution : (i) d'injecter du code ou des références dans le composant, (ii) d'intercepter les appels de méthodes (iii) et d'appeler des méthodes du composant. Pour pouvoir réaliser toutes ces opérations, les implémentations sont manipulées lors de la compilation du composant pour générer une classe dite instrumentée, permettant l'injection des valeurs dans les champs. Cette classe instrumentée est ensuite utilisée à l'exécution à la place de la classe d'origine.

En pratique, notre algorithme de résolution de dépendances est déclenché quand l'exécution courante (le thread Java) de l'application utilise un champ (un field java) associé à une dépendance qui a été au préalable instrumentée par ApAM. Si ce champ n'est pas résolu (il est « *null* »), le processus de résolution est lancé. Si la résolution réussit, l'adresse de l'instance destination est injectée dans le champ de l'instance cliente et une relation wire est créée dans ApAM pour modéliser le succès de la résolution de la dépendance (voir Figure 25). Par défaut, les wires sont créés à la demande par la résolution paresseuse.

Pour spécifier plus précisément le concept d'**injection de dépendances** nous allons étendre l'architecture de l'application par une projection vers le langage de programmation Java. Dans l'implémentation d'ApAM, cette projection se fait par instrumentation de code et par les mécanismes réflexifs de Java.

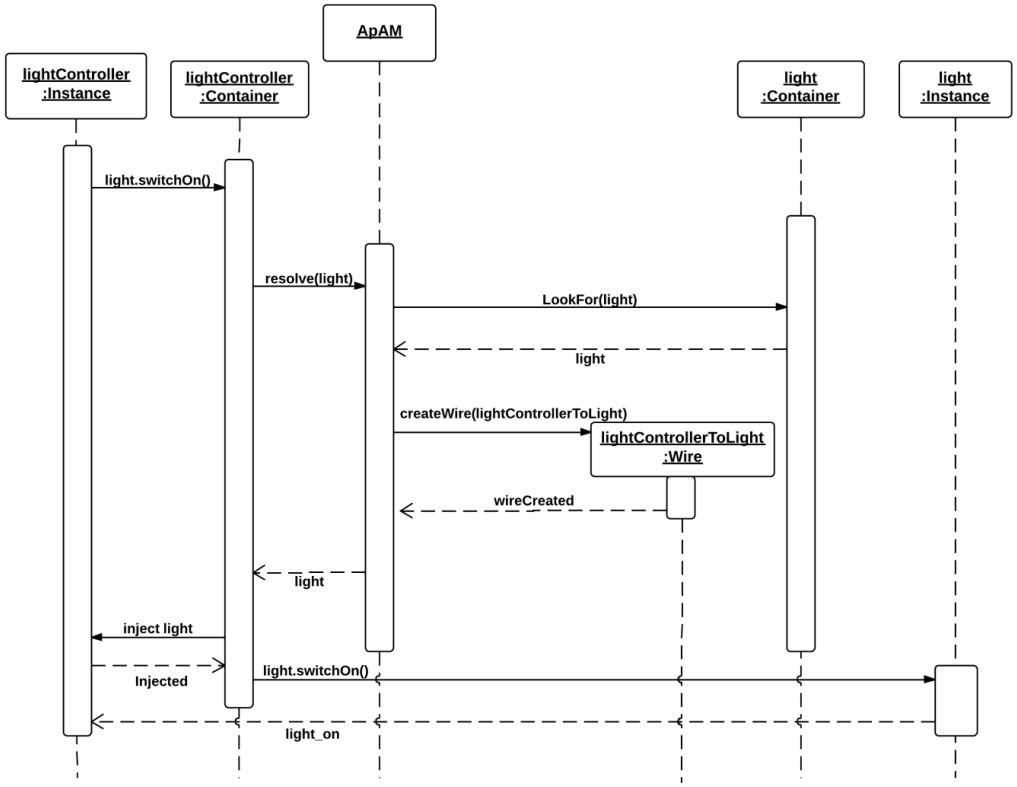


Figure 25 : Mécanisme de résolution (cas standard)

Pour les besoins de cette spécification, nous allons utiliser une modélisation largement simplifiée de certains concepts de la machine d'exécution de Java (classe, objet et champ) à l'aide des fonctions suivantes qu'on suppose prédéfinies :

$$\begin{gathered} \textit{interfaces} : \textit{Classe} \rightarrow \textit{P(Classe)} \\ \textit{champs} : \textit{Classe} \rightarrow \textit{P(Champ)} \\ \textit{type} : \textit{Champ} \rightarrow \textit{Classe} \\ \textit{classe} : \textit{Objet} \rightarrow \textit{Classe} \\ \textit{référence} : \textit{Objet} \times \textit{Champ} \rightarrow \textit{Objet} \end{gathered}$$

L'injection de dépendances est basée sur trois correspondances entre éléments de l'architecture et concepts Java : une implémentation est projetée sur une classe Java qui la réalise, les ressources fournies sont projetées vers des interfaces Java qui représentent le contrat de services et les dépendances sont associées à des champs. Nous définissons une projection d'architecture de la façon suivante :

$$\begin{gathered} \Pi^{A} \stackrel{\text{def}}{=} \{ (\pi^{R}, \pi^{I}, \pi^{D}) \mid \pi^{R} \in \mathbb{R} \rightarrow \textit{Classe} \wedge \pi^{I} \in \textit{Implémentation} \rightarrow \textit{Classe} \wedge \\ \pi^{D} \in (\textit{Implémentation} \times \mathbb{D}) \rightarrow \textit{Champ} \} \end{gathered}$$

Nous pouvons ainsi augmenter la définition d'architecture :

$$A^{\pi} \in \textit{Architecture}^{\pi} \stackrel{\text{def}}{=} \{ (A, \pi) \mid A \in \textit{Architecture} \wedge \pi \in \Pi^{A} \}$$

Et nous pouvons donner une version étendue de la fonction de validation de cohérence :

$$\pi\text{-}\mathit{cohérente} : (\mathit{Architecture}^{\pi}) \rightarrow \mathbb{B}$$

$$((\mathbf{C}, \mathbf{matérialise}), (\pi^{R}, \pi^{I}, \pi^{D})) \mapsto \mathbf{cohérente}((\mathbf{C}, \mathbf{matérialise}))$$
$$\wedge$$
$$\forall i \in \mathbf{C} \cap \mathbf{Implémentation} : \forall rf \in \mathbf{ressources(i)} : \forall\ (d, requis, contraintes) \in \mathbf{\Delta\text{-}dépendances(i)} :$$
$$\pi^{R}(rf) \in \mathbf{interfaces}(\pi^{I}(i)) \wedge \pi^{D}(i, d) \in \mathbf{champs}(\pi^{I}(i)) \wedge$$
$$requis \in \mathbb{R} \Rightarrow \mathbf{type}(\pi^{D}(i, id)) = \pi^{R}(requis) \wedge$$
$$requis \in \mathbb{C} \Rightarrow \exists c \in \mathbf{C} : \mathbf{id}(c) = requis \wedge \exists rr \in \mathbf{ressources(c)} : \mathbf{type}(\pi^{D}(i, id)) = \pi^{R}(rr)$$

À l'exécution, les instances de l'application correspondent à des objets Java et la création d'un **wire** a pour effet d'injecter la référence de l'objet cible dans les champs instrumentés de l'objet source. Nous pouvons modéliser ce comportement par une extension de la définition d'état de l'exécution :

$$E^{\pi} \in \mathit{Etat}^{\pi} \stackrel{\text{def}}{=} \{\ (\mathbf{E}, \phi) \mid \mathbf{E} \in \mathbf{Etat} \wedge \phi \in \mathbf{Instance} \rightarrow \mathbf{Objet}\}$$

Ainsi que l'extension correspondante de la fonction de validation d'un état par rapport à son architecture :

$$\pi\text{-}\mathit{valide} : (\mathit{Architecture}^{\pi} \times \mathit{Etat}^{\pi}) \rightarrow \mathbb{B}$$

$$(((\mathbf{C}, \mathbf{matérialise}), (\pi^{R}, \pi^{I}, \pi^{D})), ((\mathbf{I}, \omega), \phi)) \mapsto \mathbf{valide}\ ((\mathbf{C}, \mathbf{matérialise}), (\mathbf{I}, \omega)) \wedge$$
$$\forall (source, id, cible) \in \omega :$$
$$\mathbf{classe}(\phi(source)) = \pi^{I}(\mathbf{implémentation\text{-}de}(source)) \wedge$$
$$\mathbf{classe}(\phi(cible)) = \pi^{I}(\mathbf{implémentation\text{-}de}(cible)) \wedge$$
$$\mathbf{référence}(\phi(source), \pi^{D}(\mathbf{implémentation\text{-}de}(source), id)) = \phi(cible)$$

La résolution est le mécanisme qui nous permet de passer d'une architecture de référence vers des architectures concrètes. Grâce à ce mécanisme nous sommes capables de spécifier des architectures flexibles tout en assurant la conformité des architectures concrètes vis-à-vis de l'architecture de référence. Cependant, la résolution fait face à deux problèmes :

- Si elle permet de trouver des composants, elle reste limitée par les composants qui sont dans son environnement d'exécution.
- Si la résolution échoue, l'application ne peut pas continuer son exécution.

Pour faire face à ces deux problèmes, nous avons mis en place des mécanismes de résilience. Ces mécanisme sont configurables et permettent à l'application de guider sa résolution et de faire face à l'indisponibilité d'un composant.

5.2. RÉSILIENCE DES APPLICATIONS

La résolution permet de passer d'une architecture de référence (abstraite) à une architecture concrète lors de l'exécution. Or, il est possible que ce passage ne puisse aboutir. On parle alors d'échec de la résolution. Le but principal d'ApAM est d'adapter l'architecture invalide ou incomplète courante jusqu'à ce qu'elle redevienne valide, tout en s'assurant que l'architecture concrète est conforme à l'architecture de référence. L'objectif n'étant pas de fournir une solution optimale mais de s'assurer que l'application continue à s'exécuter et puisse survivre aux changements du contexte d'exécution. La résilience apporte à la résolution le moyen d'atteindre un ensemble plus large de composants pour favoriser la réussite de la résolution. Elle permet également de définir le comportement à adopter face à un échec de résolution à l'exécution.

Les systèmes modulaires traditionnels disposent d'un seul espace de résolution. Celui-ci est généralement représenté par un registre local contenant les composants définis au packaging (plate-forme à composants) ou ceux en cours d'exécution (plate-forme à services). En effet, dans ces systèmes l'espace de résolution est confondu avec l'état courant du système local en exécution. Pour offrir une

plus vaste possibilité de solutions, il faut agrandir l'espace de résolution au-delà du système local en exécution. Les chercheurs écologistes James Lovelock [Love00], C.S. Holling [Holl73] ou D.TILMAN [Tilm96] ont montré l'importance de la biodiversité pour la résilience [Wiki00a] :

> ***« La diversité présente dans un milieu est le gage d'un meilleur auto entretien de l'écosystème. »***

Cette même observation peut être faite sur les systèmes informatiques : plus le système est capable de trouver des ressources alternatives (au-delà de son propre écosystème), plus il sera capable d'offrir de nouvelles solutions pour satisfaire des besoins imprévus. Dans le domaine informatique, la résilience est définie dans [Lapr08] comme :

> ***« The persistence of service delivery that can justifiably be trusted, when facing changes. »***

L'auteur décrit la résilience comme la capacité d'une application à continuer à délivrer son service sans s'interrompre, malgré les changements qu'elle peut subir. Dans notre approche ces changements sont essentiellement dus à la nature du contexte d'exécution de l'application. Augmenter la résilience de l'application vis-à-vis du contexte d'exécution est l'un des points difficiles auquel nous essayons de répondre. Nous proposons alors des mécanismes automatiques d'adaptation afin éviter l'échec de l'application, cette adaptation devant être réalisée sans l'intervention d'un administrateur.

Contrairement aux approches dans lesquelles un programme ad hoc réalise les opérations d'adaptations, dans notre vision, l'adaptation représente l'exécution normale de l'application. Nous considérons qu'une adaptation est nécessaire quand l'architecture concrète n'est plus conforme à l'architecture de référence, ou si une demande de résolution est déclenchée par l'exécution de l'architecture concrète. Cela se produit en cas d'indisponibilité d'un composant requis pour une résolution de groupe ou de dépendance à l'exécution.

Dans la suite de cette partie, nous détaillerons les deux mécanismes de résilience que nous avons mis en place. Le premier consiste à maîtriser les emplacements dans lesquels les composants de l'application peuvent être trouvés et sélectionnés (résolus) : nous appelons ces emplacements des **espaces de résolution**. Puis, si aucune solution n'est disponible dans les différents espaces de résolution, notre second mécanisme consiste à offrir des **stratégies** pour faire face à **l'échec de la résolution**.

5.2.1. LES ESPACES DE RÉSOLUTION

Dans les plates-formes à service traditionnelles, l'espace de résolution est constitué exclusivement des services en exécution présents dans le registre du système local. Dans notre approche, nous permettons aux dépendances de composants de se résoudre dans plusieurs plates-formes hétérogènes et distribuées. Pour cela, nous avons mis en place un mécanisme extensible permettant d'augmenter la possibilité de trouver le(s) composant(s) nécessaire(s) à l'exécution d'une application. Ce mécanisme est basé sur des gestionnaires (ou managers) qui automatisent le provisionnement des composants dans la plate-forme ApAM. Le but des managers est de synchroniser ApAM avec d'autres plates-formes hétérogènes. Selon leurs types et le choix du développeur, le manager (ou le gestionnaire) de chaque plate-forme aura la charge de rapatrier les composants nécessaires dans la machine d'exécution. Il assure la communication entre plates-formes.

Notre approche utilise le paradigme orienté services afin de faire profiter chaque application des ressources fournies par les autres applications disponibles dans le système. Il nous offre une première extension de l'espace de résolution de l'application. Pour élargir cet espace, nous proposons aux applications de profiter des dépôts de code, des machines et des services distants. Ceci constitue une réelle avancée. En effet, l'adaptation n'est plus limitée aux composants existants dans le système local en cours d'exécution et d'autres composants peuvent être déployés à la demande, sur des machines

locales ou distantes. Ces composants peuvent également être prêtés par un tiers à travers des services locaux ou distants.

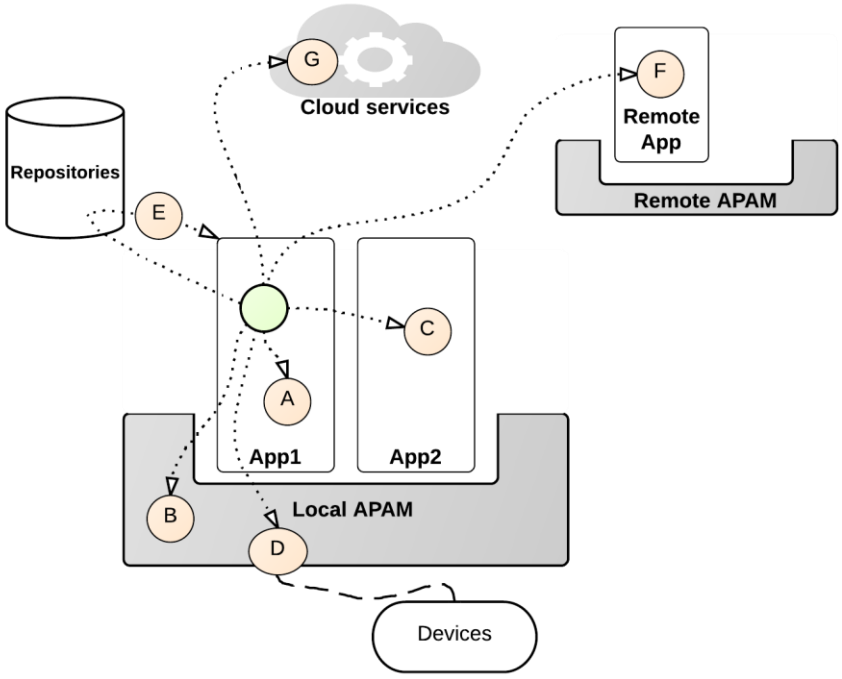


Figure 26: Les espaces de résolution

Grâce à notre approche, les mécanismes de résiliences d'ApAM peuvent tirer parti du contexte d'exécution des applications ubiquitaires pour trouver des solutions d'adaptation plus larges et diversifiées. Comme présenté dans la Figure 26, une application ubiquitaire qui s'exécute dans ApAM peut sélectionner des composants à partir :

A. De son propre contenu. Celui-ci est l'ensemble des composants déjà contenu dans l'application (comme dans les plates-formes à composants).
B. De la plate-forme locale, qui peut fournir des services techniques ou des ressources prédéfinies. Ces services ne font pas partie de l'application, mais elles sont nécessaires pour assurer son exécution (comme dans les plates-formes à services).
C. Du contenu des autres applications en cours d'exécution, si elles acceptent de partager leur contenu (voir §7 ci-dessous). Les applications qui s'exécutent dans ApAM sont capables de prêter et d'emprunter des composants et des ressources à d'autres applications de la plate-forme.
D. De dispositifs physiques. La plate-forme ApAM réifie en termes de composant, les équipements disponibles et découvrables ce qui permet aux applications de se connecter et de communiquer avec ces équipements de manière transparente.
E. Des dépôts de composants, qui peuvent être utilisés comme solution à une demande de résolution. ApAM permet ainsi le déploiement automatique et à chaud de composants dans la plate-forme. Ces dépôts peuvent être publiques ou privés à certaines applications. Les dépôts privés appartiennent à l'application qui les déclare et ils sont considérés comme faisant partie de l'application. Le déploiement des composants est automatiquement déclenché par le mécanisme de résolution d'ApAM.
F. Des plates-formes distantes qui partagent leurs ressources (applications et services techniques).
G. D'autres types de dépôts sont imaginables, puisque le système est extensible.

Selon la nature de chaque application ubiquitaire, un développeur spécifie les espaces de résolution qu'il souhaite considérer. Par exemple, il peut considérer uniquement ses dépôts privés de code ainsi que des plates-formes distantes dans lesquelles il a confiance (dans le but de limiter les risques liés à la sécurité de son application ubiquitaire). Grâce aux espaces de résolution, l'exécution d'une application ApAM peut être influencée par son contexte. Le contexte représente l'ensemble des espaces de résolution considérés par l'application (plates-formes locales, distantes, équipements découvert, dépôts de code, ressources partagées), pour réaliser des opérations d'adaptation d'architecture.

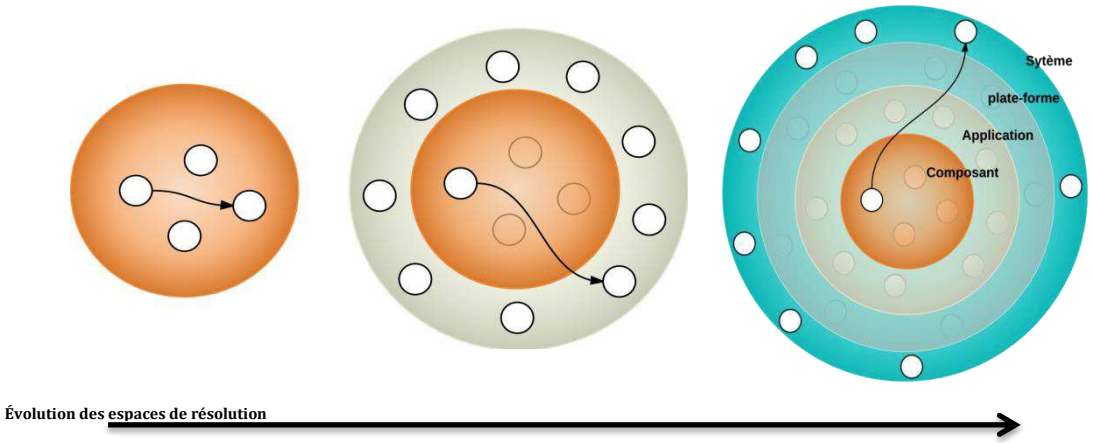


Figure 27 : Un large contexte pour la résolution des dépendances

Tous les espaces de résolution utilisés par une application peuvent influencer la résolution des dépendances des composants de cette application. En effet, chaque espace peut définir des contraintes supplémentaires en lien avec le contexte d'exécution. Ces contraintes viennent s'ajouter (i) aux contraintes décrites par le composant lui-même, (ii) aux contraintes de l'application, mais aussi (iii) aux contraintes des autres espaces. La résolution d'une dépendance dépend ainsi de l'application, du composant, du contexte d'exécution et des espaces de résolution. La Figure 27 montre comment une résolution de dépendances peut aboutir à des résultats différents selon les espaces de résolution pris en compte.

En pratique, les managers doivent implémenter l'interface « *DependencyManager* » décrit ci-dessous. Les managers qui implémentent cette interface sont alors appelés par ApAM durant le processus de résolution de dépendance.

Cette interface rassemble les fonctionnalités nécessaires pour :

- Trouver un composant en utilisant son nom/identifiant.
- Trouver un composant grâce à la description de la dépendance.
- Être notifié du résultat de la résolution ainsi que du choix des composants sélectionnés.

Chaque manager est appelé pendant le processus de résolution d'une dépendance avec en paramètre la dépendance à résoudre et ses contraintes. Le manager peut alors choisir s'il souhaite ou non participer à cette résolution. S'il participe il peut choisir l'ordre dans lequel il souhaite être appelé et il peut ajouter des contraintes.

```
/**
 * Interface that each dependency manager MUST implement.
 * Used by APAM to resolve the dependencies and manage the application.
*/

public interface DependencyManager extends Manager{

    /**
      * Provided that a dependency resolution is required by client,
      * each manager is asked if it want to be involved.
      */
      public void getSelectionPath(Component source, DependencyDeclaration dependency,
List<DependencyManager> selPath);

   /**
     * Performs a complete resolution of the dependency.
     *The manager is asked to find the "right" implementations and instances for the
provided dependency.
     */
    public Resolved<?> resolveDependency(Component source, Dependency dependency);

  /**
    * Once the resolution terminated, either successful or not, the managers are
notified of the current selection.
    */
    public void notifySelection(Component client, ResolvableReference resName, String
depName, Implementation impl, Instance inst, Set<Instance> insts);

}
```

5.2.2. EXEMPLES D'ESPACES DE RÉSOLUTIONS

Pour illustrer le rôle des managers dans le rapatriement de composants, nous allons présenter deux managers différents. Ceux-ci ont été réalisés de manière générique pour que toutes les applications ubiquitaires qui s'exécutent sur ApAM puissent profiter de leurs fonctionnalités.

Le manager OBR

Grâce à ce manager, ApAM est capable d'explorer un ou plusieurs dépôts (de type OBR [Apac00c]) de composants réutilisables afin de sélectionner le composant souhaité. Ensuite il est déployé et instancié si nécessaire. Ainsi, la plate-forme ApAM peut déclarer les dépôts avec lesquels elle souhaite être liée. Les dépôts déclarés à ce niveau sont alors disponibles pour toutes les applications de la plate-forme. Le manager OBR n'est pas le seul espace de résolution de type dépôt de code qu'il est possible d'implémenter dans ApAM. En effet, tous les types de dépôt de code peuvent être ajoutés dans ApAM, tel que P2 [Ecli00] par exemple.

Pour ce manager, le rapatriement des composants depuis les différents dépôts de codes se fait uniquement à la demande. Ceci nous permet de déployer les composants seulement pour répondre à une demande de résolution. Les composants déployés peuvent par la suite être instanciés avec les bonnes propriétés.

La Figure 28 décrit le processus de la résolution avec l'intervention du manager OBRMAN. Le processus est le suivant :

- La machine ApAM intercepte les accès vers les champs qui sont liés à des déclarations de dépendance.

- Le mécanisme de résolution cherche un composant qui satisfait la demande. Pendant cette phase, la demande est transmise à OBRMAN.
- OBRMAN consulte la liste des ressources disponibles dans les différents dépôts déclarés.
- Le composant choisi est déployé dans la machine ApAM.
- Le champ est injecté avec l'adresse du composant.
- L'exécution de l'application peut alors continuer.

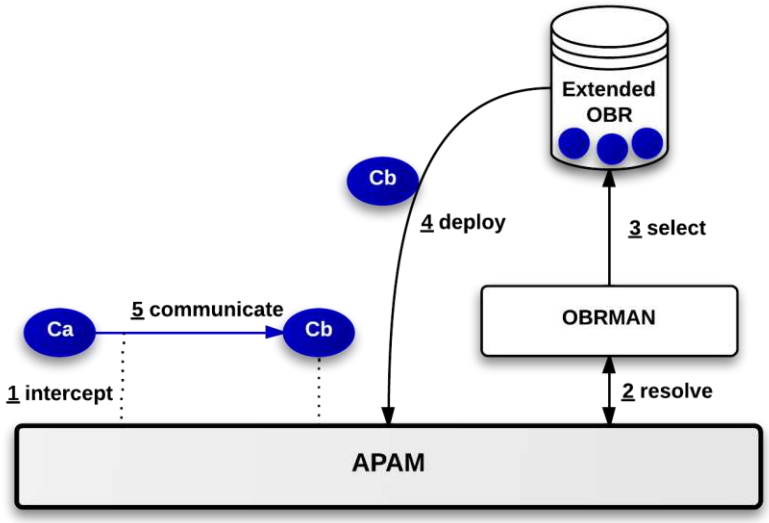


Figure 28 : La résolution par OBRMAN

En pratique, dans le cas spécial d'OBRMAN, les dépôts que le développeur souhaite utiliser sont définis comme suit :

```
LocalMavenRepository = [true | false]
DefaultOSGiRepositories = [true | false]
Repositories= http:/…….../repository.xml \
                      file:/F:/…… \
                      https:/…
```

Ainsi, l'application peut utiliser le dépôt de code « Maven » spécifié dans l'environnement où elle s'exécute (« *LocalMavenRepository* »), profiter des dépôts de code décris dans la plate-forme OSGi™ qui héberge ApAM (« *DefaultOSGiRepositories* ») et enfin spécifier une liste ordonnée d'adresses vers les dépôts OBR dans lesquels nous souhaitons réaliser la résolution (« *Repositories* »).

ApAM s'appuie sur OBR pour la résolution dynamique des packages pendant le déploiement des bundles. Les dépôts de code OBR ont été enrichis par les concepts de notre modèle à composants. Ainsi, on retrouve dans le dépôt des composants de type spécification, implémentation, et instance ayant les mêmes propriétés et les mêmes définitions de propriété. Cela permet de vérifier les contraintes de résolution d'un composant avant son déploiement. La vérification des contraintes se fait alors de façon homogène, indépendamment de la nature de la plate-forme.

Le manager DISTRIMAN

Le but de DISTRIMAN (le manager de distribution) est d'essayer de résoudre les dépendances en cherchant dans les plates-formes ApAM distantes. Pendant la résolution, DISTRIMAN peut réaliser des demandes de résolution vers les plates-formes qu'il a découvertes. Si la plate-forme ApAM distante réussit à résoudre la dépendance, un proxy est créé dans la machine locale. Ce proxy est connecté à un

point de terminaison (« *EndPoint* ») de la machine distante. Ces points fournissent les fonctionnalités nécessaires aux clients pour établir une connexion distante. Une fois cette connexion établie, la machine distante peut répondre à des demandes de résolutions de dépendances en communiquant les informations nécessaires pour la création de proxy vers ses composants.

Le processus de la résolution utilisant DISTRIMAN est décrit dans la Figure 29. Le manager de distribution génère des proxys pour représenter les composants dans la machine d'exécution locale. Ce comportement est complètement transparent à l'exécution et aucune intervention humaine n'est nécessaire. Il est à noter que pour éviter de créer des proxys pour tous les composants distants disponibles, ce manager ne créé les proxys que pour une demande de résolution qu'il a réussi à réaliser. Ainsi, seuls les composants distants qui répondent à une demande de résolution sont représentés en local.

DISTRIMAN surveille l'arrivée et le départ des machines ApAM. Ainsi, pour faire face à un départ de machine, il détruit les différents proxys locaux liés à celle-ci. Les composants qui ont disparu sont remplacés, s'ils sont nécessaires, par le déclenchement de nouvelles demandes de résolution. Cette gestion du dynamisme (apparition ou disparition des machines) est aussi appliquée aux composants distants. Ainsi, le comportement de chaque composant est surveillé par DISTRIMAN et il est répercuté sur les proxys du composant.

Une demande de résolution sur une plate-forme distante va invoquer les différents managers présents sur cette machine distante. Cela peut avoir comme effet le déploiement puis l'instanciation d'un composant à travers OBRMAN distant. Cependant, la résolution distante n'est pas transitive, les autres DISTRIMAN présents ne sont pas appelés pour une demande de résolution distante.

Pour que la résolution de dépendance à distance fonctionne, il y a certaines règles de distribution à respecter. Ces règles s'appliquent aux bundles, qui sont les unités de déploiement gérées par ApAM. Il faut par exemple bien décomposer les bundles et surtout séparer les spécifications et les implémentations (ne pas hésiter à mettre le composant spécification dans un bundle différent de son composant implémentation). Ceci permettra d'utiliser la spécification indépendamment de l'implémentation et de pouvoir la déployer dans plusieurs machines.

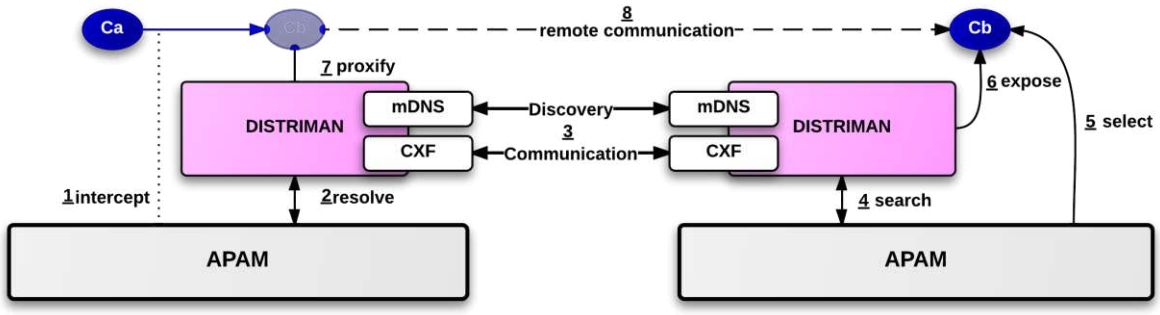


Figure 29 : La résolution par DISTRIMAN

5.2.3. PRIORITÉ DES MANAGERS

La plate-forme ApAM a été conçue pour être extensible avec des managers tiers. Ces managers ont la charge des tâches spécifiques et sophistiquées nécessaires à la résolution des dépendances des composants de l'application en exécution sur ApAM. Il est important de spécifier l'ordre dans lequel les différents managers vont intervenir dans le mécanisme de résolution.

Nous avons mis en place un mécanisme de priorité qui permet aux développeurs de spécifier l'ordre dans lequel les managers doivent intervenir dans le processus de résolution. La priorité d'un gestionnaire peut être spécifiée statiquement ou dynamiquement pendant l'exécution de l'application. L'ordre spécifié statiquement dans le manager est considéré comme étant l'ordre par défaut. Par défaut, le manager OBRMAN a une priorité plus élevée que DISTRIMAN, cela signifie qu'ApAM va

d'abord essayer de trouver un composant à déployer avant de réaliser une demande de résolution distante vers les machines découvertes. Cet ordre est configurable : le développeur peut modifier tous les ordres établis par défaut dans la plate-forme ApAM.

L'ordre d'interrogation des managers est important. En effet, ce sont les composants présents dans l'espace de résolution géré par le manager le plus prioritaire qui sont d'abord proposés comme solution pour une demande de résolution. Cependant, pour valider une solution donnée par un manager, le composant est soumis à l'approbation des autres managers. Ceci nous assure qu'un composant choisi dans un espace de résolution respecte bien toutes les contraintes imposées par tous les managers qui interviennent dans la résolution.

5.2.4. ÉCHEC DE LA RÉSOLUTION

Les espaces de résolution nous permettent d'atteindre plus de composants et ainsi augmenter les chances de maintenir les applications en exécution. Or, il est toujours possible de ne pas obtenir de composant satisfaisant pour une demande de résolution donné. La multiplication des espaces ne garantit pas la réussite de toutes les demandes de résolution. Les composants requis peuvent ne pas exister dans les espaces de résolution disponibles : c'est le cas en particulier pour un dispositif physique non présent ou hors de portée. C'est pourquoi nous avons mis en place des stratégies permettant au développeur de définir le comportement à adopter face à un échec de la résolution.

Les stratégies de résiliences ne sont pas totalement transparentes. Le développeur de l'application doit être conscient des adaptations que va subir son application et il doit alors guider cette adaptation en fonction de ses besoins. Il est aussi le mieux placé pour distinguer les parties critiques de l'application ainsi que les adaptations adéquates. Plus il fournit d'informations, plus l'adaptation sera précise. ApAM encourage les développeurs à détailler la politique d'adaptation à utiliser. Ces politiques sont interprétées à l'exécution et prises en compte pour guider le choix à réaliser par ApAM. Elles interviennent lors de l'échec d'une résolution de dépendance.

Ces stratégies vont permettre aux applications d'obtenir une certaine résilience par rapport à leur contexte d'exécution. Elles permettent de maîtriser l'échec de résolution de façon localisée. Ainsi, pour chaque dépendance de l'application on peut spécifier la réaction adéquate quand le phénomène d'échec de résolution se produit. Dans notre approche, chaque déclaration de dépendance est traitée séparément. Cela nous permet de gérer le comportement des demandes de résolution indépendamment les unes des autres pour un même composant.

Nous avons défini trois stratégies de résilience dans ApAM : l'échec, l'attente et le retour arrière. Cette liste de mécanismes n'est pas exhaustive : d'autres politiques peuvent être imaginées selon le domaine de l'application et ses spécificités. Néanmoins, ces mécanismes sont génériques, implantés dans le noyau et peuvent être utilisés par toutes les applications ApAM. Il est donc tout à fait possible de réaliser des adaptations contrôlées, dynamiques et autonomiques dans ApAM. Dans la suite de cette partie nous détaillerons les trois stratégies que nous avons mises en place.

L'échec

Dans ApAM, le comportement par défaut de toutes les demandes de résolution qui ne réussissent pas à obtenir une solution pour leur dépendance est l' « échec ». L'échec de la résolution est accepté par notre plate-forme et ne provoque pas pour autant l'interruption complète de l'exécution de l'application.

Pour cette stratégie, quand une demande de résolution échoue, l'erreur est remontée vers le code de l'application. Il est donc à la charge de l'utilisateur de spécifier le comportement spécifique, de manière programmatique, pour l'échec de la résolution. La Figure 30 illustre le cas d'exception lors de l'absence de l'équipement de détection de présence.

En pratique, quand une demande de résolution échoue, le champ lié à la demande de résolution n'est pas injecté. Il est toutefois possible pour le développeur de l'application de spécifier les

exceptions qu'il souhaite lever pour un échec de résolution. Ces exceptions peuvent être décrites sur n'importe quelle dépendance, et peuvent être différentes d'une dépendance à l'autre.

Pour illustrer ce comportement, nous avons repris notre scénario (voir §3 ci-dessus). Dans cet exemple, nous supposons que l'implémentation du LIG pour le composant « *PresenceDetector* », que nous allons appeler *« LIGPresenceSensorDriver »*, va adopter une stratégie d'échec. Elle se décrit comme suit :

```
<implementation name="LIGPresenceSensorDriver" ... >
     …
    <dependency id= "presence-device" specification="PresenceSensor" fail= "exception"
                            exception="fr.imag. ….failedException" >
        <interface field="presenceSensors"/>
        <message push="notifyNewPresence"/>
         …
    </dependency>
</implementation>
```

L'ensemble des instances de l'implémentation *« LIGPresenceSensorDriver »* qui seront créées par ApAM adopteront le comportement défini dans cette implémentation. L'attribut *fail= « exception »* signifie que si la résolution de dépendance échoue, une exception sera levée. Cette exception a été spécifiée à travers le champ *exception="fr.imag. ….failedException".* Les conséquences de cet échec sont que le champ *"presenceSensors"* et la méthode *"notifyNewPresence"* ne seront pas injectés par les valeurs du composant instance dont la spécification est *"PresenceSensor"*. Si aucune exception n'est définie, ApAM lèvera l'exception *"…ResolutionException"* par défaut. Tout cela supposant que le développeur ait mis en place le code nécessaire pour capturer l'exception.

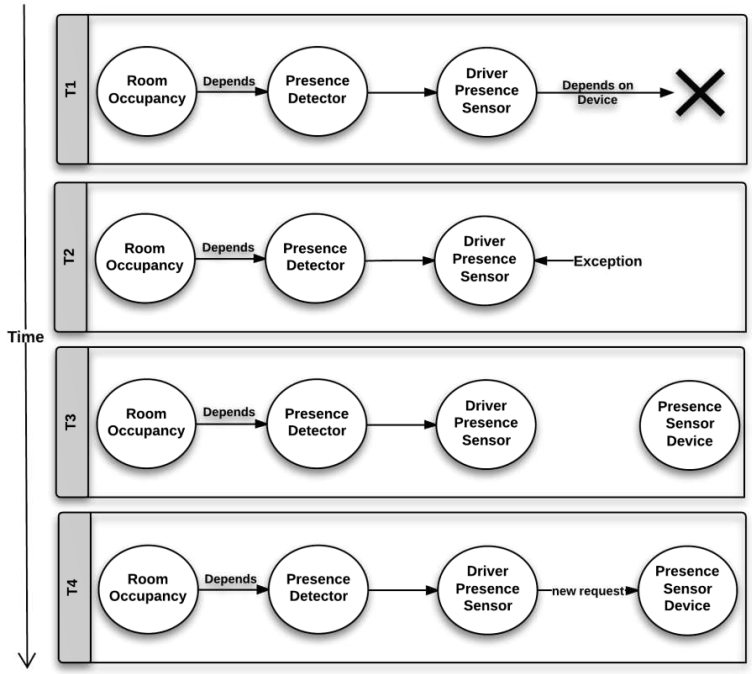


Figure 30 : Stratégies d'échec

Cet échec ne signifie pas l'arrêt de l'application. Une nouvelle demande de résolution sera déclenchée lors de la prochaine tentative de l'application pour accéder au champ *"presenceSensors"*. En fonction des composants disponibles au moment de la nouvelle demande, la résolution réussira ou échouera.

L'attente

Dans un environnent ubiquitaire, les applications doivent faire face à un comportement dynamique de leurs composants. Certains composants de l'application peuvent être indisponibles à un certain moment, mais finissent par réapparaître à d'autres moments.

Dans ApAM, l'attente de composant est un comportement dynamique qui peut être spécifié sur les dépendances de l'application. Ainsi, face à un échec de la résolution, ApAM peut bloquer l'exécution courante de l'application. L'application n'est pas entièrement mise en attente. Seule l'exécution (le thread Java) qui a déclenché la demande de résolution est concernée par cette attente. L'apparition du composant nécessaire ne nécessite pas, dans ce cas de figure, de nouvelle demande de résolution. ApAM surveille l'apparition des composants dans le contexte d'exécution. Dès qu'un composant qui satisfait la dépendance apparaît, la connexion est réalisé et l'application continue son exécution (le *thread* est relâché, voir Figure 31). Afin d'éviter que l'application ne reste indéfiniment en attente d'un composant manquant, il est possible de paramétrer le temps d'attente souhaité.

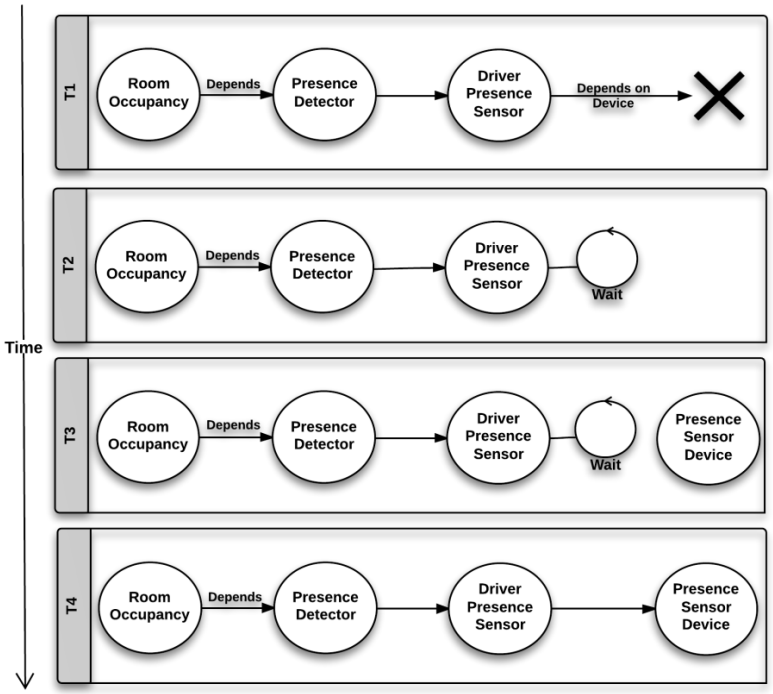


Figure 31 : Stratégies d'attente

En pratique, l'exécution courante correspond au thread Java qui est responsable du déclenchement de la résolution. C'est ce thread qui est mis en attente du composant. Les autres threads de l'application qui ne dépendent pas de ce dernier continuent à s'exécuter. Pour illustrer ce comportement, toujours à partir de l'implémentation *« LIGPresenceSensorDriver »*, la définition d'une stratégie d'attente est spécifiée comme suit :

```
<implementation name="LIGPresenceSensorDriver " ...>
      …
       <dependency id= "presence-device"  specification="PresenceSensor" wait="10000" >
            <interface field="presenceSensors"/>
            <message push="notifyNewPresence"/>
       …
      </dependency>
 </implementation>
```

L'ensemble des instances de l'implémentation *« LIGPresenceSensorDriver »* qui seront créées par ApAM adopteront le comportement définit dans cette implémentation. L'attribut *wait* signifie que si la résolution de dépendance échoue, les threads qui passent par cette dépendance seront mis en attente. Lorsque le composant instance requis par cette dépendance apparaît, l'exécution des threads est alors autorisée à continuer normalement. Cependant, l'attribut *wait="10000"* définit une attente maximale de 10 secondes. Après l'expiration de ce temps, la résolution échoue et les autres stratégies définies peuvent alors prendre la main (exception et/ou retour arrière).

Retour arrière

Le retour arrière est une stratégie qui fait face à l'échec de l'application par une reconfiguration automatique de l'architecture. En effet, dans notre approche, pour faire face à la nature imprévisible des environnements ubiquitaires, l'application s'exécute d'une manière incrémentale. Ainsi, le mécanisme de résolution a été conçu pour fonctionner pas à pas. La conséquence du choix d'un composant sur la suite de l'exécution et sur les futures demandes de résolutions n'est pas prise en compte. Le raisonnement et la prise de décision pour le choix d'un composant se font pas à pas pendant l'exécution de l'application. Il se peut alors que le choix d'un composant nous conduise à construire une architecture qui va échouer dans le futur. Pour répondre à ce problème nous avons mis en place la stratégie de retour arrière.

Cette stratégie permet de revenir sur les choix de composants antérieurs. Le développeur de l'application peut décider des dépendances qui peuvent provoquer un retour arrière. En cas d'échec de la résolution de ces dépendances, leurs composants seront « cachés » par ApAM, ce qui produit la destruction de toutes les connexions (wires) vers ce composant. Le composant sera ensuite « blacklisté » (voir Figure 32). Les futures demandes de résolutions ne se verront plus proposer ce composant. Il redeviendra disponible quand la dépendance qui a déclenché la stratégie sera résolue.

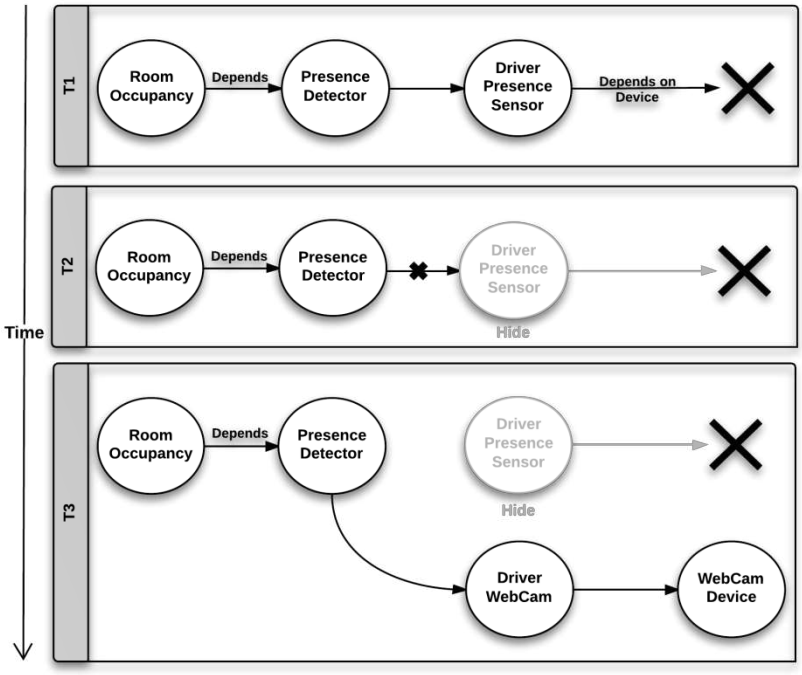


Figure 32 : Stratégies de retour arrière

Pour illustrer ce comportement, nous reprenons l'implémentation *« LIGPresenceSensorDriver »*. Nous avons défini sur cette implémentation la stratégie de retour arrière comme suit :

```
<implementation name=" LIGPresenceSensorDriver " ...
        …
 <dependency id= "presence-device" specification="PresenceSensor" hide= "true" >
      <interface field="presenceSensors"/>
      <message push="notifyNewPresence"/>
             …
     </dependency>
</implementation>
```

L'ensemble des instances de l'implémentation *« LIGPresenceSensorDriver »* qui seront créées par ApAM adopteront le comportement défini dans cette implémentation. L'attribut *« hide »* signifie qu'en cas d'échec de la résolution, l'implémentation *« LIGPresenceSensorDriver »* ainsi que ses instances seront cachées pour les futures demandes de résolution. La demande de résolution sera alors déclenchée par les composants qui dépendent de ce composant. Ils se verront alors proposer d'autres solutions pour leur demande de résolution.

Cette stratégie n'affecte pas le code de l'application. Elle a pour avantage de gérer la reconfiguration au niveau de l'architecture de l'application. Grâce à cette stratégie, l'application peut trouver des solutions alternatives pour une partie de son architecture. En effet, même si notre stratégie est spécifiée localement (pour une dépendance donnée), elle permet à ApAM de remplacer des branches entières de l'architecture de l'application en réalisant des adaptations en cascade (voir Figure 33). Ce comportement permet de revenir sur un ensemble de décisions prises dans un contexte d'exécution antérieur et qui empêche l'exécution courante de l'application. En pratique, au cours de l'exécution d'un thread qui traverse plusieurs dépendances ayant adopté cette même politique, la résolution peut reculer de plusieurs pas, jusqu'à trouver une alternative qui ne déclenche pas l'échec de la résolution. Ceci provoque une reconfiguration automatique d'une partie de l'architecture.

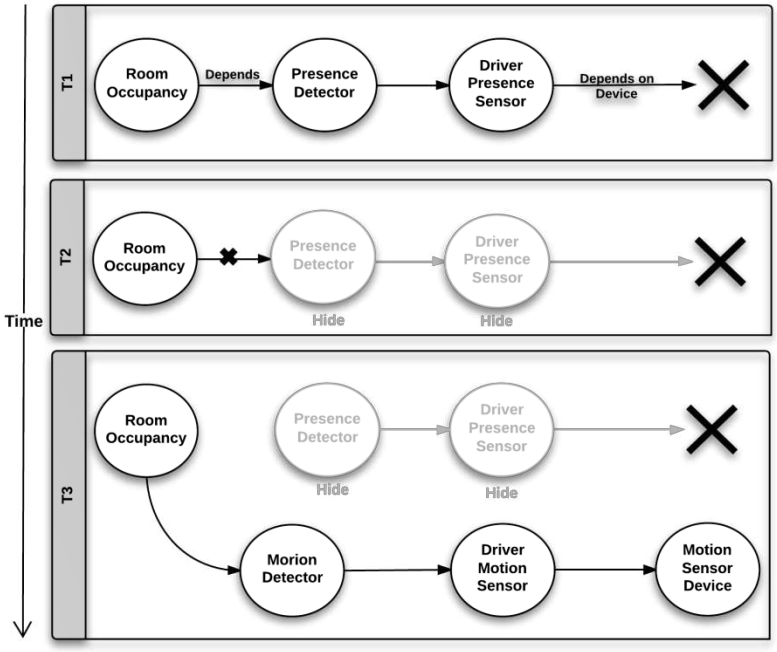


Figure 33 : Retours arrière en cascade

La stratégie *« wait »* peut être combinée avec l'une des deux autres : en cas d'échec d'une résolution, on attend d'abord l'expiration du délai d'attente. Par la suite, on déclenche l'exception et finalement on applique la stratégie du retour arrière.

5.3. SYNTHÈSE

Généralement, dans les approches existantes, les ressources dont dépend une application sont préfinies et disponibles au démarrage de l'application. Cependant, dans notre contexte, la disponibilité des ressources ne peut pas toujours être prédite. Les ressources peuvent être volatiles et donc apparaître et disparaître à tout moment. Parti de ce constat, nous proposons que les demandes de résolution ne se déclenchent qu'en cas de besoin. ApAM a mis en place le mécanisme de résolution **paresseuse** (Lazy), dans le but d'adopter le comportement décrit précédemment. De plus, même si ApAM encourage la résolution **paresseuse**, il est possible pour une application de réaliser des résolutions **immédiates** (Eager) afin de répondre aux besoins spécifiques de certaines applications.

Il est toujours possible (même avec une résolution paresseuse) que l'application n'arrive pas à résoudre ces dépendances. Cependant, combiné avec les espaces de résolution, ApAM peut offrir de multiples alternatives d'adaptations à la volée :

- **Déploiement à la demande :** les composants requis peuvent être déployés à partir de dépôts de codes spécifiques, locaux ou distants.
- **Instanciation à la demande :** à partir des composants disponibles dans ApAM, une instance qui satisfait les dépendances et les contraintes peut être créée à la volée.
- **Liaison à la demande :** les liens entre les composants ne sont établis que si les dépendances déclarées sont réellement appelées par le code métier du composant.

Finalement, l'échec de la résolution provoque le déclenchement automatique des stratégies de résiliences spécifiées par le développeur.

6. MATÉRIALISATION D'UNE APPLICATION APAM

Dans la plate-forme ApAM, tout composant s'exécute à l'intérieur d'un conteneur chargé de gérer l'exécution de ses composants. Ces conteneurs sont eux-mêmes des composants appelés des composants composites ou tout simplement des composites. Notre modèle à composants permet la composition hiérarchique : une application ApAM est réalisée en assemblant différents composants primitifs (définis précédemment) ou composites (une application n'est qu'un cas particulier de composite).

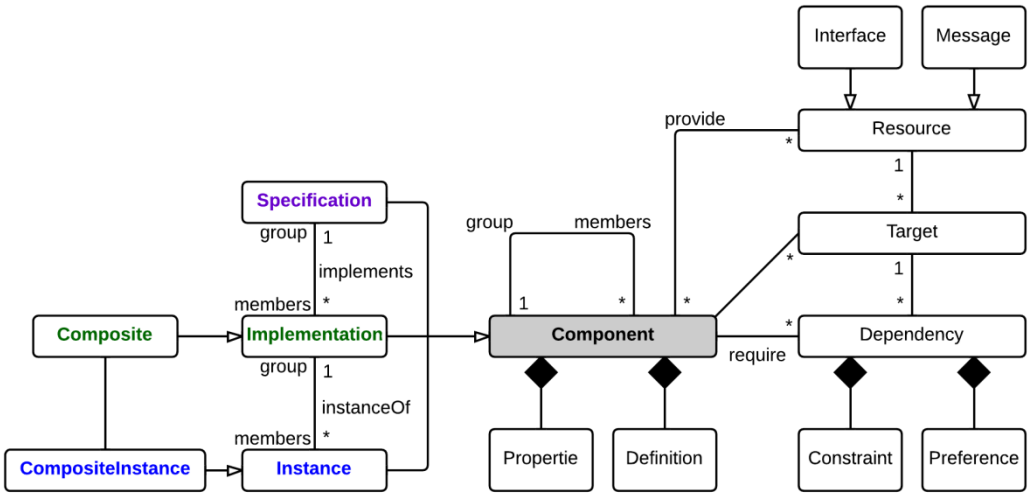


Figure 34 : Le modèle ApAM étendu par les composites

Le composite est constitué de la composition d'un ensemble de composants et de leurs connexions. Cet ensemble n'est pas visible en dehors du composite et permet à ce dernier d'être perçu de l'extérieur comme un composant simple ou une boîte noire. Cette encapsulation est une abstraction qui permet au composite de cacher et de protéger ses composants et de gérer les effets d'échelle. Les composants évoluent dans le contexte du composite et indépendamment du contexte d'exécution externe.

La description « traditionnelle » d'un composite consistant à fournir une liste exhaustive de tous ses éléments et leurs connexions est contraignante, inadéquate, voir même impossible pour les applications dynamiques. Les composants entrant dans la composition peuvent ne pas être connus au développement, ne pas exister à un instant donné ou doivent être ceux disponibles dans le contexte d'exécution courant. Ce contexte est inconnu au développement et à la composition pour les applications ubiquitaires.

De plus, même s'il est théoriquement possible de décrire et de réaliser avec les technologies traditionnelles des applications ubiquitaires qui répondent aux besoins et hypothèses identifiées ci-dessus, les concepteurs et les développeurs doivent faire face à une telle complexité que l'entreprise devient rapidement infaisable en pratique. L'ensemble des propriétés mentionnées fait que la sélection des éléments d'un composite et la gestion des composites en général sont ardues et quasi impossibles à faire à la main pour des composites complexes. Dans la plupart des langages d'architecture existants, l'ensemble des composants et des connexions d'un composite est défini statiquement, ce qui rend difficile la modification de la composition du composite à l'exécution.

Dans notre approche, **toute application est représentée par un composite**. Un composite dans ApAM contient l'architecture de référence d'une application. Pour modéliser la notion de composite

dans notre spécification[12], nous introduisons une nouvelle partition des types de composants avec une contrainte supplémentaire :

$$\textit{Implémentation} = \textit{Implémentation}_{\textit{primitive}} \cup \textit{Composite}$$
$$\textit{Instance} = \textit{Instance}_{\textit{primitive}} \cup \textit{Instance}_{\textit{composite}}$$

$$\forall p \in \textit{Instance}_{\textit{primitive}} : \textit{implémentation-de}(p) \in \textit{Implémentation}_{\textit{primitive}} \wedge$$
$$\forall c \in \textit{Instance}_{\textit{composite}} : \textit{implémentation-de}(c) \in \textit{Composite}$$

Notre application ubiquitaire est un composite construit à partir d'un ensemble de buts qui expriment les propriétés et les contraintes à satisfaire par les éléments participant à la composition.

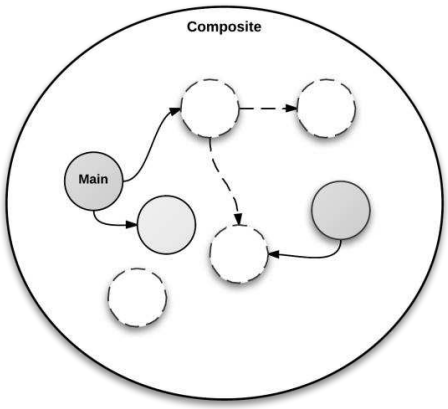


Figure 35 : Assemblage en intention

Les composites ApAM n'indiquent pas explicitement leur contenu. Nous pouvons ne spécifier que les fonctionnalités (interfaces) que cette application va fournir (voir §6.1 ci-dessous). ApAM a alors à charge de sélectionner, déployer et instancier dynamiquement, à partir de dépôts de code et en fonction du contexte de l'exécution, les composants capables de fournir le service correspondant. L'encapsulation peut être réalisée sans forcément lister l'ensemble des composants entrant dans sa composition. Cependant, comme dans les compositions statiques, il est possible de déclarer dans ApAM l'ensemble ou un sous-ensemble des composants qui font partie du composite.

À l'exécution, ApAM construit une architecture concrète de l'application. **L'architecture concrète de l'application est aussi appelée l'état courant de l'exécution** de l'application. En effet, pour une même architecture de référence, la plate-forme ApAM peut générer différentes architectures concrètes. Cela dépend de l'abstraction de l'architecture de référence et du contexte d'exécution. Les architectures concrètes construites sont valides par rapport à l'architecture de référence.

Notre approche permet de construire à l'exécution le contenu des applications. La construction à l'exécution s'appuie sur le mécanisme de résolution qui permet la sélection automatique des éléments nécessaires à la composition. Le calcul permettant de choisir les différents éléments participant à la composition est réalisé en suivant l'exécution du composant principal et en se basant sur la description

[12] L'introduction du concept de composite entraine une modification de la modélisation de l'injection de dépendances, car les composites et leurs instances ne sont pas mappés directement vers des objets Java. En principe, il faudrait reprendre la formalisation présentée dans la section 5.1.2. Néanmoins, les modifications sont relativement intuitives et nous préférons ne pas les présenter pour privilégier la compréhension de la spécification.

en intention du composite, sur la description des composants déjà sélectionnés et sur l'état courant du système. Les composants sont sélectionnés parmi les éléments qui sont disponibles et qui correspondent aux caractéristiques désirées. Cependant, la description peut aussi spécifier statiquement des composants qui doivent être contenus dans le composite à l'exécution.

Les composites dans ApAM (de la même manière que pour les composants primitifs) s'appuient sur le concept de groupes d'équivalence : on parle d'**implémentation composite** et **d'instance composite**. Ces deux niveaux sont dérivés des concepts d'implémentation et d'instance. L'abstraction du composite nous permet de réduire la complexité de la conception des applications en fournissant des représentations de haut niveau réutilisables et compréhensibles. Dans la suite de cette partie nous détaillerons les deux niveaux d'abstraction du composite.

6.1. COMPOSITION EN INTENTION

6.1.1. IMPLÉMENTATION COMPOSITE

Une implémentation composite est une implémentation. Elle hérite donc de toutes les propriétés du concept d'implémentation. Une application dans ApAM peut fournir et requérir des ressources comme n'importe quelle implémentation. Grâce à sa propriété d'encapsulation, un composite permet de définir des architectures hiérarchiques

Lors de la conception, une implémentation composite spécifie les ressources qu'elle fournit. Un composant dit *« main »* fournissant les même ressources que le composite doit alors être déclaré dans la description de l'implémentation composite. Il est le seul composant qui doit obligatoirement être spécifié dans l'implémentation composite. Par défaut, les composites ne spécifient pas statiquement leur contenu, il est dynamiquement construit grâce aux mécanismes de résolution.

Notre mécanisme ne fait pas de distinction entre les composants primitifs et les composites. Ainsi, un composant spécification peut être aussi bien une abstraction d'un composant ou d'un composite. Cette particularité nous permet d'avoir un haut niveau d'abstraction commun entre les composants primitifs et les composites. La complexité du contenu des composites est donc masquée au moment de la concrétisation pour les composants clients. De plus, les composants qui sont contenus dans un composite peuvent être atomiques (c'est-à-dire des composants primitifs) ou composites, ce qui offre la possibilité de construire des composites hiérarchiques.

En pratique, en se basant sur notre scénario, nous avons défini l'implémentation composite « *RoomOccupencyComposite* ». Ce composite spécifie :

- La spécification qu'il fournit « *RoomOccupency* », les ressources décrites dans cette spécification seront fournies par le composite.
- Son composant principal, « *RoomOccupencyImpl* ».
- Ses propriétés (« *vendor* ») et ses définitions (« *location* »).

La définition de l'implémentation composite est spécifiée dans un fichier XML comme suit :

```
<composite name="RoomOccupencyComposite" main="RoomOccupencyImpl"
        specification="RoomOccupency">
     <definition name ="location" value="{kitchen,bedroom,bathroom}" />
     <property name ="vendor" value="LIG" />
     …
</composite>
```

Le composite « *RoomOccupencyComposite*» déclare le composant « *RoomOccupencyImpl* » comme étant son composant principal. Dans ce cas, il s'agit d'un composant de type implémentation. Cependant, dans la déclaration du composite, il est possible d'utiliser tous les niveaux d'abstraction de composant.

Au niveau de l'architecture, un composite doit pouvoir spécifier quels sont les implémentations qu'il possède. Cependant, nous voulons aussi permettre le partage de code entre applications. Il est donc possible de déployer la même implémentation dans plusieurs composites. Nous proposons alors une extension de la définition d'architecture pour représenter ces **déploiements hiérarchiques :**

$$A \in \textit{Architecture} \stackrel{\text{def}}{=} \{ (C, \textit{matérialise}, \textit{déployé-dans}) \mid C \subseteq \textit{Composant} \wedge \textit{matérialise} \subseteq C \times C \wedge \textit{déployé-dans} \subseteq \textit{Implémentation} \times \textit{Composite} \}$$

De façon analogue, l'état de l'exécution est constitué d'une **hiérarchie d'instances** (atomiques et composites). Dans ce cas, l'état de l'application doit être encapsulé, nous avons alors une composition stricte. Nous proposons donc de modéliser la hiérarchie par une fonction qui dénote le parent de chaque instance (qui peut être nulle pour les instances du premier niveau) :

$$E \in \textit{Etat} \stackrel{\text{def}}{=} \{(I, \omega, \textit{parent}) \mid I \subseteq \textit{Instance} \wedge \omega \subseteq \textit{wire} \wedge \textit{parent} \in \textit{Instance} \rightarrow \textit{Instance}_{\textit{composite}} \cup \{\emptyset\}\}$$

Évidemment, ces deux hiérarchies sont liées. Nous allons donc étendre la fonction de validation pour tenir compte de ces nouvelles conditions de validité :

$$\textit{valide}_{\textit{composite}} : (\textit{Architecture} \times \textit{Etat}) \rightarrow \mathbb{B}$$

$$((C, \textit{matérialise}, \textit{déployé-dans}), (I, \omega, \textit{parent})) \mapsto \textit{valide}\,((C, \textit{matérialise}), (I, \omega)) \wedge$$
$$\forall \iota \in I : \textit{parent}(\iota) \neq \emptyset \Rightarrow \textit{implémentation-de}(\iota)\ \textit{déployé-dans}\ \textit{implémentation-de}(\textit{parent}(\iota))$$

6.1.2. LES CONTRAINTES CONTEXTUELLES

L'un des rôles principal du composite est de gérer la composition des composants qu'il englobe. La gestion du contenu des composites est réalisée à travers l'expression d'un nouvel ensemble de contraintes et de préférences (pour les dépendances) au niveau des composites. Cet ensemble viendra contraindre les dépendances spécifiées au niveau des composants. On appelle alors contraintes contextuelles, l'ensemble des contraintes qui vont permettre de maîtriser la composition. Les contraintes contextuelles permettent à un composite de raffiner les contraintes de résolution des dépendances pour les composants qu'il encapsule.

Un composant peut être utilisé par différents composites simultanément. En suivant les principes de réutilisation et de séparation des préoccupations, les composants doivent être conçus, autant que possible, indépendamment de leur contexte d'exécution. Les contraintes de dépendances spécifiées au niveau des composants sont appelées « intrinsèque » car elles devront être satisfaites dans tous les contextes d'utilisation. Ces contraintes intrinsèques peuvent limiter la réutilisation d'un composant. C'est pour cela que nous encourageons la réalisation de composants avec le minimum de contraintes et que nous préconisons de définir dans les composites les contraintes adéquates pour chaque composant.

Un composite est aussi en charge de définir les stratégies de résolution et de résilience pour guider la résolution des dépendances des composants qu'il englobe. Ces stratégies sont exprimées par les contraintes contextuelles du composite. Pour formaliser cette notion de **contraintes contextuelles** dans l'architecture nous associons à chaque composite une fonction qui donne les contraintes additionnelles à ajouter à chaque résolution de dépendance.

$$A \in \textit{Architecture} \stackrel{\text{def}}{=} \{ (C, \textit{matérialise}, \textit{déployé-dans}, \textit{contraintes-contextuelles}) \mid C \subseteq \textit{Composant} \wedge \textit{matérialise} \subseteq C \times C \wedge \textit{déployé-dans} \subseteq \textit{Implémentation} \times \textit{Composite} \wedge \textit{contraintes-contextuelles} \in \textit{Composite} \rightarrow (\textit{Composant} \times \mathbb{D} \times (\mathbb{R} \cup \mathbb{C}) \rightarrow P(\mathbb{E})) \}$$

Et nous adaptons la fonction de validité d'un état E par rapport à son architecture A :

$$\textbf{\textit{valide}}_{\textbf{\textit{contextuel}}} : (\textbf{\textit{Architecture}} \times \textbf{\textit{Etat}}) \rightarrow \mathbb{B}$$

$$\textbf{\textit{((C, matérialise, déployé-dans, contraintes-contextuelles), (I, ω, parent))}} \mapsto$$
$$\forall (source, id, cible) \in \boldsymbol{\omega} : \textbf{\textit{parent}}(source) \neq \emptyset \Rightarrow$$
$$\exists gs \in \boldsymbol{C} : source\ \textbf{\textit{matérialise}}^*\ gs \wedge \exists (idd, requis, contraintes) \in \boldsymbol{\Delta}\textbf{\textit{-dépendances}}(gs) :$$
$$\boldsymbol{\omega}\textbf{\textit{-valide}}((idd, requis, contraintes),(source, id, cible)) \wedge$$
$$\forall cc \in \textbf{\textit{contraintes-contextuelles}}(\textbf{\textit{implementation-de}}(\textbf{\textit{parent}}(source))) (source, idd, requis) :$$
$$[\![cc]\!](\textbf{\textit{propriétés}}(cible))$$

Le composite permet au développeur d'avoir une vue globale du contexte dans lequel s'exécutent un ensemble de composants et de l'usage réel de chacun d'eux. Grâce au composite il peut modifier et raffiner les stratégies définies dans les composants qu'il englobe et plus particulièrement contraindre le comportement dynamique de la résolution.

Par exemple, le composite *« RoomOccupencyComposite »* souhaite adapter le comportement dynamique de toutes les dépendances vers des drivers pour faire face à l'échec d'une demande de résolution. En utilisant la convention de nom selon laquelle toute spécification de dispositif à pour suffixe « driver », on obtient alors la définition suivante :

```
<composite name=" RoomOccupencyComposite "  …>
   …
    <contentMngt>
        <dependency specification="*Driver" eager=" true" fail="exception"
            exception="….DeviceDriverNotFound"/>
        </dependency>
    …
    </contentMngt>
</composite>
```

Supposant que le composant *«LIGPresenceDetector »* est placé à l'intérieur du composite *« RoomOccupencyComposite »* et qu'il a définit la dépendance suivante :

```
<implementation name="LIGPresenceDetector" ...>
   ….
   <dependency specification="PresenceSensorDriver" fail="exception"
                                    exception="….PresenseSensorNotFound"/>
   …
</implementation>
```

Comme la dépendance *« PresenceSensorDriver »* vérifie le patron *« *Driver »*, la dépendance contextuelle définie au niveau du composite va surcharger la stratégie pour faire face à l'échec de la résolution de la dépendance (exception) et rajoute un nouveau comportement (« *eager* ») :

- *« eager=true »* signifie que la dépendance doit être résolue tout de suite après la création de l'instance de *« LIGPresenceDetector »*. Par défaut, *« eager = false »*, les dépendances sont alors résolues uniquement quand les champs associés à la dépendance sont sollicités par l'exécution courante de l'application.
- *« Exception=" …DeviceDriverNotFound " »* signifie qu'en cas d'échec de la résolution de la dépendance, ApAM va lancer l'exception mentionnée dans la dépendance contextuelle. Cette valeur surcharge la valeur de l'exception définie dans le composant (*…PresenseSensorNotFound*).

Les contraintes contextuelles peuvent exprimer aussi des contraintes contextuelles :

```
<composite name=" RoomOccupencyComposite " … >
   …
     <contentMngt>
       <dependency specification=" Driver*" …. >
     <constraints>
      <instance filter="(OS=Linux)" />
     </constraints>
    </dependency>
  </contentMngt>
</composite>
```

Dans cet exemple, toutes les demandes de résolutions de dépendances vers une instance dont la spécification contient le suffixe *« *Driver »*, sont contraintes par *(OS=Linux)* en plus de leur contraintes intrinsèques (internes). Les contraintes qui sont spécifiées sont ajoutées à la liste des contraintes de la dépendance. Ils s'appliqueront pour toutes les demandes de résolutions vers l'ensemble des composants qui vérifie le patron de la dépendance contextuelle.

Les composites ApAM permettent d'installer une frontière pour délimiter des ensembles de composants. Cette frontière permet de mieux structurer le rôle de chacun des composants, de leurs dépendances et de leurs propriétés dynamiques. En plus de fournir des ressources et de contraindre les dépendances des composants qu'ils encapsulent, les composites peuvent déclarer des dépendances externe et permettre à leur composants de profiter des composants externes. Nous détaillerons ces dépendances dans les paragraphes suivants.

6.1.3. LA PROMOTION DES DÉPENDANCES

Le concept de composite pour les applications ApAM permet la création de composites classiques dits auto-contenus. Cette propriété garantit que les dépendances des composants encapsulées dans le composites seront résolues uniquement par des composants inclus dans ce même composite. Le concepteur du composite est ainsi assuré du contenu de l'assemblage de l'application. Cependant, son application ne pourra pas profiter des composants présents dans la plate-forme.

Pour faire face à cette limitation, ApAM définit le mécanisme de « *promotion* ». Grâce à ce mécanisme, les composants internes du composite peuvent utiliser des composants externes pour résoudre leurs dépendances. Pour cela, les composants utilisent les dépendances définies par leur composite pour traverser la frontière. C'est donc à la charge du concepteur du composite de déclarer les dépendances du composite (voir Figure 36). Pour illustrer le mécanisme de promotion, nous allons enrichir notre définition du composite « *RoomOccupencyComposite* » par une dépendance externe vers le composant dont la spécification est « *PresenceSenor* », nous proposons la description suivante :

```
<composite name="RoomOccupencyComposite" main="RoomOccupencyImpl"
            specification="RoomOccupency">
    <definition name ="location" value="{kitchen,bedroom,bathroom}" />
    <property name ="vendor" value="LIG" />
    <dependency specification="PresenceSensor" id="device-dependency">
       <constraints>
          <instance filter="(vendor=lig)"/>
       </constraints>
   </dependency>
<composite>
```

Grâce à cette déclaration, toutes les dépendances dont la destination est un composant qui a pour spécification « *PresenceSensor* » seront promues automatiquement. Ainsi, ces dépendances vont pouvoir être résolues à l'extérieur du composite, dans le contexte du composite englobant. Il faut noter

que le composite peut définir des contraintes supplémentaires pour les dépendances qui vont être promues. Ces contraintes viendront s'ajouter au contraintes définies en intrinsèque dans la dépendance du composant.

La déclaration statique de la promotion permet de décrire des stratégies d'échec spécifiques pour chacune des dépendances promues. Cependant la déclaration statique des promotions nécessite une connaissance statique des composants du composite. À titre d'exemple, supposons que le composant « *LIGPresenceSensorDriver* » défini dans la section §5.2.4 ci-dessus, est à l'intérieur de notre composite « *RoomOccupencyComposite* », la promotion de sa dépendance « *presence-device* » doit être déclarée de la manière suivante dans le composite :

```
<composite name="RoomOccupencyComposite" main="RoomOccupencyImpl"
                  specification="RoomOccupency">
    …
 <dependency specification="PresenceSensor" id="device-dependency">
      <constraints>
          <instance filter="(vendor=lig)"/>
        </constraints>
  </dependency>
  <contentMngt>
    <promote implemetation="LIGPresenceSensorDriver"  dependency="presence-device"
       to=" device-dependency" fail="exception" exception="fr.liglab….DeviceNotFound"/>
  </contentMngt>
<composite>
```

Dans notre exemple, la dépendance *« presence-device »* va déclencher une exception de type « *fr.liglab….DeviceNotFound* » en cas d'échec de la demande de résolution à travers sa promotion *«device-dependency»*.

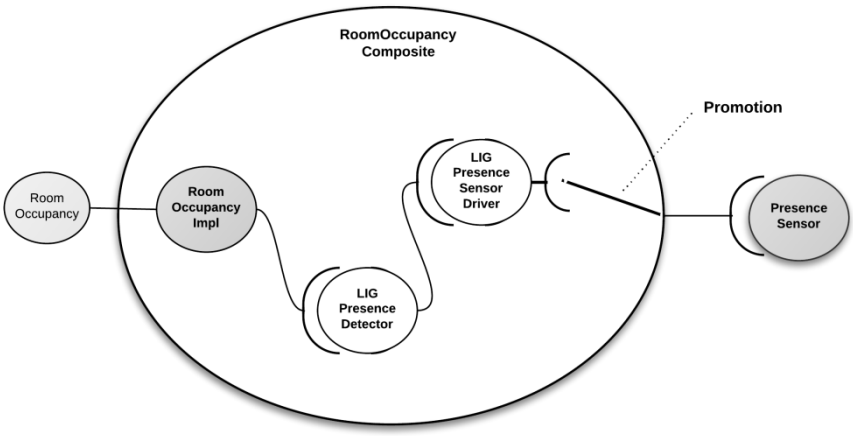


Figure 36 : Exemple de promotion de dépendances

La définition de l'application exprimée dans l'implémentation composite est validée statiquement et cela dès la compilation des différents composants et composites de l'application. Les propriétés, les contraintes et les préférences de l'application portant sur des composants formellement définis, la validité de la définition d'une application peut être vérifiée statiquement.

Notre approche permet la composition dynamique des applications au cours de leur exécution. La composition dynamique peut diminuer les probabilités d'échec d'une application. 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).

L'implémentation composite est donc interprétée par notre plate-forme ApAM qui construit et exécute l'application pas à pas. L'exécution de l'application est représentée par un composite définit en terme d'instance appelé instance composite.

6.2. CONSTRUCTION ET EXÉCUTION DE L'APPLICATION

À l'exécution, l'encapsulation va permettre d'isoler l'assemblage de chaque sous-système de l'application. La description de l'implémentation composite va être conservée par la plate-forme à l'exécution. Elle servira d'architecture référence pour les demandes de résolution. L'architecture concrète de l'application est construite à partir de l'interprétation de cette description. Cette architecture concrète est conceptualisée à l'exécution par un composite appelé **instance composite**. Elle représente une application ou un sous-système pendant son exécution. Par la suite nous détaillerons le rôle de l'implémentation composite et le contenu des composites à l'exécution.

6.2.1. IMPLÉMENTATION COMPOSITE À L'EXÉCUTION

L'implémentation composite a deux rôles dans l'exécution des applications ApAM. (i) Il contient la description originale de l'architecture de référence. (ii) Il permet de réaliser des assemblages en terme d'implémentation.

En effet, l'implémentation composite permet de regrouper l'ensemble des implémentations qui participent à une exécution de cette implémentation composite. Dans cet assemblage, nous retrouvons l'implémentation principale correspondant au composant principal décrit par l'implémentation composite. Il contient aussi un graphe interconnecté de toutes les implémentations, construit au fur et à mesure par les différentes demandes de résolution.

L'implémentation composite est perçue à l'exécution comme une simple implémentation, ce qui nous permet de l'utiliser pour répondre à des demandes de résolution. C'est à partir des implémentations contenues dans ce composite que nous choisirons ou créerons les instances des composants qui feront partie de la concrétisation de l'architecture de référence. L'ensemble de ces instances interconnectées est contenu dans l'instance composite.

6.2.2. INSTANCE COMPOSITE

Une instance composite est une instance dont l'implémentation est une implémentation composite. Elle encapsule l'architecture concrète de l'une des exécutions de l'implémentation composite correspondant. Dans cette instance composite nous retrouvons une instance principale qui est une instance du composant *« main »*. À l'exécution, les instances contenues dans une instance composite peuvent être des instances simples ou composites. Le fait qu'un composant soit atomique ou composite est totalement masqué. Les instances créées à l'exécution sont placées à l'intérieur de l'instance composite à partir de laquelle la demande de résolution a été émise. Nous avons alors une exécution hiérarchique des composites dans ApAM.

Les instances composites permettent d'avoir une vue d'ensemble sur les interactions entre les composants au sein d'un même composite. Une instance composite racine appelé *« root »* est toujours créée au démarrage de notre plate-forme. Ce composite est la racine de tous les composites qui s'exécutent dans la plate-forme et constitue le contexte commun à toutes les applications. Les applications patrimoines sont automatiquement placées dans un composite créé à la volée pour les contenir. Elles peuvent ainsi s'exécuter sur la plate-forme sans qu'aucune modification ne leur soit apportée.

Une instance composite représente l'architecture concrète de l'application à l'exécution, la validité de la composition de cette architecture est garantie par nos mécanismes de conformité. Les résolutions dynamiques assurent la conformité d'une instance composite vis-à-vis de son implémentation composite et de sa spécification.

Dans ApAM, les composites fournissent les fonctionnalités nécessaires pour (i) contrôler le comportement dynamique de la résolution de dépendance et pour (ii) gérer les différents espaces de

résolution d'une manière contextuelle et spécifique à chaque application. Jusque-là nous avons surtout détaillé le premier point. Dans la section suivante, nous allons détailler le rôle des composites dans la gestion des espaces de résolution.

6.2.3. GESTION DES ESPACES DE RÉSOLUTION

Le processus de résolution dans ApAM dépend des espaces de résolution disponibles. Ces espaces représentent les emplacements dans lesquels ApAM va essayer de trouver une solution pour les demandes de résolution. Dans notre approche la liste des espaces de résolution est définie par implémentation composite.

Chaque espace de résolution est géré par un manager spécifique à cet espace. Un même manager peut intervenir dans la demande de résolution de plusieurs composites, mais le comportement que doit adopter ce manager peut être très différent d'une implémentation composite à une autre. C'est pour cette raison que dans ApAM, les managers sont pilotés par des modèles qui décrivent le comportement de chaque manager vis-à-vis d'un composite en particulier.

Dans ApAM, chaque implémentation composite peut spécifier un modèle pour chaque manager qui intervient dans la résolution de ses dépendances. Ce modèle fournit au manager auquel il est associé les informations nécessaires pour le guider dans la recherche et la sélection de composants.

Chaque modèle est spécifique à un type d'espace de résolution. L'interface « Manager » décrite ci-dessous est implémentée par tous les managers. Elle permet de récupérer automatiquement :

- Le nom du manager.
- Sa priorité dans le processus de résolution par rapport aux autres managers.
- Le modèle destiné à guider la résolution de ce manager par implémentation composite.

```
public interface Manager {

    /**
     * the name of that manager.
     */
       public String getName();

     /**
       * returns the relative priority of that manager, for the resolution algorithm
       */
        public int getPriority();

    /**
       * A new composite, holding a model managed by this manager, has been
       * created. The manager is supposed to read and interpret that model.
       */
       public void newComposite(ManagerModel model, Composite composite);
}
```

Pour illustrer le comportement des managers, nous allons reprendre les deux exemples de manager donnés précédemment : OBRMAN et DISTRIMAN.

6.2.4. EXEMPLE 1 : GESTION D'OBRMAN PAR COMPOSITE

Dans le cas d'OBRMAN, le composite spécifie dans le modèle la liste des dépôts de code où les composants pertinents pour ce composite sont stockés. Ainsi, chaque implémentation composite peut définir où sont ses dépôts de code.

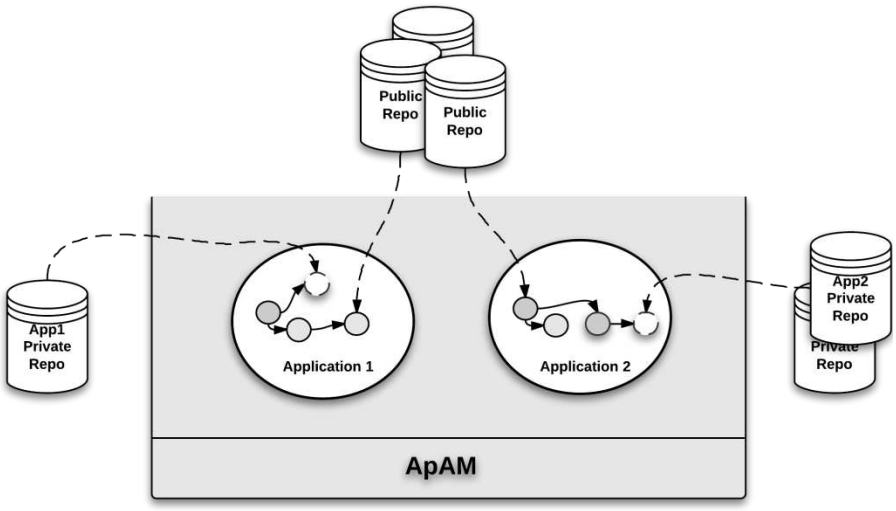


Figure 37 : Multiples dépôts de code

Nous faisons la distinction entre deux types de déploiements dans ApAM : le déploiement logique et le déploiement physique. Le déploiement physique implique le chargement d'un bundle OSGi™ dans la plate-forme. Le déploiement logique indique qu'une implémentation requise existe déjà dans la plate-forme mais qu'elle est contenue dans une autre implémentation composite. OBRMAN réalise alors une copie de l'implémentation puis la place dans le composite. Ainsi, contrairement aux instances, les implémentations peuvent appartenir à plusieurs composites.

Le modèle du manager *« M »* associé à un composite *« C »* est un fichier de nom *« C.M.cfg »* qui doit se trouver dans le répertoire *« ressource »* de l'environnement de développement. Par exemple, pour le manager OBRMAN, le fichier *« C.OBRMAN.cfg »*, peut contenir l'information suivante :

```
LocalMavenRepository = [true | false]
DefaultOSGiRepositories = [true | false]
Repositories= http:/……../repository.xml \
                        file:/F:/…… \
                        https:/…
Composites=S1CompoMain CompoXY …
```

Les dépôts décrits par les attributs « *DefaultOSGiRepositories »* et « *LocalMavenRepository »* sont considérés comme des dépôts publiques, c'est-à-dire qu'ils sont partagés par tous les composites qui s'exécutent sur la même plate-forme ApAM. Ainsi, chaque composite peut décider s'il veut utiliser ces dépôts. Les dépôts qui sont décrits par l'attribut « *Repositories »* sont privés. Chaque composite peut alors donner une liste de dépôts qui seront utilisés exclusivement pour ses demandes de résolutions (voir Figure 37). Chaque composite est également capable de spécifier une liste des composites (grâce au mot clé « *Composites »*). Il pourra alors profiter des dépôts de code spécifiés par chacun de ces composites. L'ordre dans lequel les composites sont listés définit les priorités dans lesquelles les dépôts vont être consultés. Ces composites doivent être présents dans la plate-forme, sinon ils sont simplement ignorés.

Il est possible de spécifier le comportement par défaut du manager OBRMAN pour le composite « *root* » d'ApAM. Pour cela, il suffit de créer un fichier « *root.OBRMAN.cfg* » dans lequel on décrit les différents dépôts de code.

Toutes les implémentations produites au développement sont publiées dans des dépôts spécifiques. En effet, chaque composite spécifie des dépôts privés ou publiques dans lesquels se

trouvent certains de ses composants. À l'exécution, ApAM utilise les dépôts pour déployer les composants souhaités.

6.2.5. EXEMPLE 2 : GESTION DE DISTRIMAN PAR COMPOSITE

Dans le cas du manager de distribution, le modèle contient des expressions logiques qui définissent les machines auxquelles le composite souhaite se connecter. Les machines qui vérifient les expressions peuvent potentiellement collaborer avec ce composite en échangeant des ressources ou des données.

Le modèle de DISTRIMAN décrit les stratégies de résolution distantes pour un composite ou un ensemble de composites. Ce modèle définit les composants qu'il souhaite importer ou exporter à partir de machines distantes. Ci-dessous un exemple de ce modèle :

```
<distriman>
      <composite type-name="Exp">
            <import specification="Exp" machines="Exp" install="Exp" />
            <export implementation="Exp" />
      </composite>

      <composite type-name= …. />

</distriman>
```

La balise *<import>* signifie qu'une résolution distante est possible si :

- La source de la demande de résolution est un composite qui satisfait l'expression décrite dans l'attribut « *type-name* ».
- La destination de la résolution satisfait l'expression donnée dans l'attribut « *specification* ». Nous pouvons remplacer cet attribut par « *implementation* », « *interface* » ou « *message* ».
- La description de la machine distante satisfait l'expression spécifiée dans « *machines* ». La résolution de DISTRIMAN interroge machine par machine. Il s'agit donc de la première machine qui satisfait l'expression qui est interrogée en premier. Par défaut, *machines="true"*, permet aux applications de chercher dans toutes les machines découvertes.

DISTRIMAN est aussi capable de provoquer le déploiement distant des composants en collaboration avec OBRMAN. L'attribut *install="Exp"* exprime les conditions que les machines distantes doivent satisfaire pour que la résolution puisse provoquer le déploiement automatique et distant des composants nécessaires pour répondre à une demande de résolution.

La balise *<export>* indique l'ensemble des composants qui peuvent être atteints par des machines distantes. Les composants qui satisfont les expressions spécifiées dans « *specification* » ou « *implementation* » sont alors exposés pour les machines distantes.

6.3. SYNTHÈSE

Notre approche vise à contrôler l'application par la gestion de son architecture. L'élément central est que les opérations d'adaptation d'architecture doivent être réalisées de la manière la plus transparente possible pour l'utilisateur de l'application. Nous avons donc proposé un ensemble de mécanismes et de stratégies pour maintenir une application en exécution malgré les changements imprévisibles du contexte d'exécution. Nous avons ainsi défini un mécanisme automatique pour résoudre les dépendances de l'application à l'exécution. Nous avons mis en place les espaces de résolution pour offrir à l'application un environnement riche et diversifié lui permettant de répondre aux différentes demandes de résolution. Finalement, pour faire face à l'échec de la résolution, nous

avons spécifié un ensemble de stratégies localisées visant à maintenir l'application en exécution malgré l'échec de la résolution de ses dépendances.

Cependant, nous avons souhaité ouvrir notre plate-forme pour permettre l'exécution de plusieurs applications réalisées par différents développeurs. Notre approche a alors été étendue pour gérer la cohabitation des applications. Ainsi, nous détaillerons dans la partie suivante comment cette cohabitation a été rendue possible et les différents impacts qu'elle a eue sur nos mécanismes et stratégies.

7. APAM UNE PLATE-FORME MULTI-APPLICATIONS

ApAM permet l'exécution simultanée de plusieurs applications. Ces applications peuvent coopérer, partager des composants, partager des ressources (capteur, actionneurs) et peuvent (ou doivent) réagir aux modifications du même contexte. ApAM fournit le moyen de contrôler cette collaboration à travers des mécanismes de visibilité. Ces mécanismes ont pour but de structurer la composition de l'application et de gérer le partage des composants entre applications. D'autres, problèmes liés à l'exécution simultanée de plusieurs applications (exemple : la gestion des conflits d'accès aux ressources partagées), sont gérés par d'autres mécanismes (voir Chapitre 7 §2.1.2).

La cohabitation des applications dans ApAM se fait dans un environnement ouvert, où les ressources de chaque application peuvent être accessibles par toutes les autres applications présentes sur la plate-forme. Notre objectif est d'offrir des mécanismes d'isolation pour gérer le partage des ressources entre ces différentes applications. Pour cela, ApAM fournit les fonctionnalités nécessaires pour que le partage des ressources soit contrôlé de manière fine et efficace. Le système de contrôle mis en place vient influencer le mécanisme de résolution d'ApAM à travers les décisions prises au préalable par les développeurs d'applications. Ainsi, nous nous appuyons sur les concepts d'implémentation composite et d'instance composite présentés précédemment. Cela nous permet d'éviter une structure à plat (où n'importe quel composant peut être vu et utilisé par n'importe quel autre) et de mieux contrôler les différents échanges de ressources entre les applications, tout en assurant la protection des applications et de leurs données.

Un des rôles principaux du composite dans la cohabitation des applications est de définir l'isolation et le niveau de partage. Ainsi, les développeurs d'applications peuvent contrôler le degré de collaboration ou d'isolation avec les applications tierces au sein de la même plate-forme d'exécution. En se basant sur le concept de composite, ApAM fournit les fonctionnalités pour gérer la visibilité et la portée des composants de chaque application. En effet, ApAM encourage la collaboration entre applications en définissant les frontières du composite avec une perméabilité variable, permettant aux dépendances « de passer à travers » la paroi du composite. Le degré de perméabilité est spécifié par l'architecte qui peut ainsi décider si l'application est (i) une « boîte blanche » permettant aux autres applications d'utiliser ses propres composants, (ii) « opportuniste » et « profiter » des composants disponibles (prêtés par d'autres applications en exécution), (iii) « auto-contenue » avec sa propre liste de composants, ou (iv) « boîte noire » et ne partager aucun composant avec son environnement, ce qui donne alors une encapsulation complète (comme la plupart des approches à composants traditionnelles).

Chaque composite gère la portée de la résolution des dépendances des composants qu'il contient. Le développeur d'une application ApAM peut spécifier en intention (au niveau de l'implémentation composite) les composants de son application qui peuvent être utilisés par d'autres applications. C'est l'ensemble des composants qu'il souhaite prêter : « l'export ». Au niveau de chaque implémentation composite, il peut aussi spécifier en intention les composants qu'il souhaite emprunter des autres applications de la plate-forme : « l'import » (voir Figure 38). Ces ensembles de composants sont définis, comme pour les compositions, par des expressions interprétées à l'exécution. Cela donne à la composition une grande variabilité dans le choix des composants.

7.1. IMPORT DE COMPOSANTS

Dans une implémentation composite il est possible de spécifier les instances que nous souhaitons emprunter (importer) et qui sont exposées par les autres composites. À l'aide de la balise ***« import »***, une expression permet de définir les instances ou implémentations des composants qui doivent être empruntées. Les demandes de résolution de dépendances qui vérifient l'expression fournie ont alors le droit de profiter des ressources exposées dans la plate-forme. Les requêtes qui ne vérifient pas l'expression d'import doivent se contenter des ressources qui se trouvent à l'intérieur de leur propre composite, ou elles devront déployer les composants qui leur sont nécessaires.

Le développeur d'une implémentation composite doit être capable de décider s'il a besoin ou non d'importer des implémentations ou des instances exposées par d'autre composites. La balise ***<import implementation=expression instance=expression>*** permet de définir l'import de composant de type implémentation et de type instance. Si le composant vérifie les contraintes exprimées par l'expression, la plate-forme va tenter d'importer ce composant. Par défaut, l'expression est toujours vraie (*« implementation=true instance=true »)*. Exemple :

```
<import implementation="(vendor=Philips)" instance="false"/> <!—default is true -->
```

Dans cet exemple, l'implémentation qui vérifie l'expression *«vendor=Philips »* sera importée dans l'implémentation composite mais aucune instance (*« instance = false »)* ne sera importée dans les instances composites de cette application.

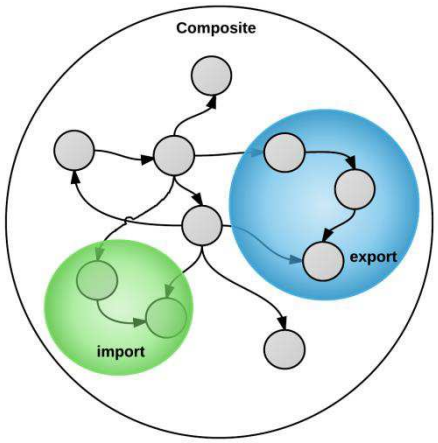


Figure 38 : La portée d'une application

Si une implémentation composite définit l'expression suivante :

```
<import implementation="false" instance="false"/>
```

Alors, l'implémentation et l'instance composite seront auto contenues. L'implémentation composite devra déployer ses propres implémentations et créer ses propres instances qui seront contenues dans l'instance composite.

7.2. EXPORT DE COMPOSANT

La balise « export » définit quels sont les composants contenus dans l'implémentation composite qui seront visibles à l'extérieur. La valeur de cette balise est une expression. Les différents composants (implémentations ou instances) qui sont encapsulés dans le composite devront vérifier l'expression pour pouvoir être exposés aux applications tierces s'exécutant dans la plate-forme. Quant aux composants spécifications, ils sont eux publiques et visibles par tous.

Si aucun export n'est spécifié, tous les composants du composites peuvent être utilisés par les autres applications de la plate-forme. La balise ***« export »*** permet de décrire le niveau d'export pour chaque composite.

```
<export  implementation="Expression" instance="Expression"/> <!-- true by default -->
```

Les composants qui sont dans le composite et qui vérifient les expressions définies dans la balise export seront exposés et donc visibles par les composants des autres composites. Par exemple, l'expression :

```
<export implementation="false" instance="false"/>
```

Celle-ci signifie que le composite est une boîte noire qui cache son contenu. Il ne rend visible aucune ressource (ou composant). Par défaut, si aucune balise de visibilité n'est définie (import ou export), le composite exporte tous ses composants et importe toutes les ressources disponibles.

Les règles de visibilité sont modélisées au niveau de l'architecture de l'application en associant à chaque composite une expression qui exprime la contrainte d'import ou export :

$$A \in \textit{Architecture} \stackrel{\text{def}}{=} \{ (\textit{C, matérialise, déployé-dans, contraintes-contextuelles, import, export}) \mid \textit{C} \subseteq \textit{Composant} \wedge \textit{matérialise} \subseteq \textit{C} \times \textit{C} \wedge \textit{déployé-dans} \subseteq \textit{Implémentation} \times \textit{Composite} \wedge \textit{contraintes-contextuelles} \in \textit{Composite} \rightarrow \textit{Composant} \times \mathbb{D} \times (\mathbb{R} \cup \mathbb{C}) \rightarrow P(\mathbb{E}) \} \wedge \textit{import} \in \textit{Composite} \rightarrow \mathbb{E} \wedge \textit{export} \in \textit{Composite} \rightarrow \mathbb{E} \}$$

À l'exécution, l'état de l'application doit respecter les règles de visibilité. En particulier, tous les wires qui traversent la frontière d'une instance composite doivent vérifier les contraintes d'importation et d'exportation. La fonction de validation de l'état par rapport à son architecture est étendue ainsi :

$$\textit{valide}_{\textit{visibilité}} : (\textit{Architecture} \times \textit{Etat}) \rightarrow \mathbb{B}$$

$$((\textit{C, matérialise, déployé-dans, contraintes-contextuelles, import, export}), (\textit{I}, \omega, \textit{parent})) \mapsto$$

$$\textit{valide}_{\textit{contextuel}}((\textit{C, matérialise, déployé-dans, contraintes-contextuelles}), (\textit{I}, \omega, \textit{parent})) \wedge \forall (\textit{source, id, cible}) \in \omega : \textit{parent}(\textit{source}) \neq \textit{parent}(\textit{cible}) \Rightarrow (\textit{parent}(\textit{source}) \neq \emptyset \Rightarrow [\![\textit{import}(\textit{implémentation-de}(\textit{parent}(\textit{source})))]\!](\textit{propriétés}(\textit{cible}))) \wedge (\textit{parent}(\textit{cible}) \neq \emptyset \Rightarrow [\![\textit{export}(\textit{implémentation-de}(\textit{parent}(\textit{cible})))]\!](\textit{propriétés}(\textit{cible})))$$

7.3. CONCLUSION

Les règles de visibilité présentées dans cette section représentent un mécanisme de structuration qui permet la modularisation de l'application et la gestion des partages des ressources. Le mécanisme de visibilité structure les ressources partagées en permettant un accès plus fin et contrôlé. Cependant, les règles de visibilité ne constituent pas un mécanisme de sécurité pour les applications. Pour définir le contrôle d'accès, ApAM s'appuie sur les mécanismes de sécurité de Java et d'OSGi™. À la résolution de chaque dépendance, ApAM vérifie que le client dispose des autorisations nécessaires pour accéder aux ressources souhaitées. ApAM suit la spécification de sécurité OSGi™ et utilise le *« ServivePermission »* pour valider les accès.

TROISIÈME PARTIE :
RÉALISATIONS ET EXPÉRIMENTATIONS

CHAPITRE 6
IMPLÉMENTATION

1. VUE GLOBALE

Afin de mettre en œuvre notre approche pour la construction et l'exécution des applications ubiquitaires, nous avons fait le choix de nous appuyer sur la plate-forme à services OSGi™ et d'étendre le modèle à composants orienté services Apache iPOJO. Dans ce chapitre, nous justifierons le choix de ces deux technologies, puis nous décrirons notre implémentation d'ApAM.

La réalisation d'une application ubiquitaire à l'aide d'ApAM passe par deux phases : la première phase est la compilation du code Java et de nos métadonnées. Un plugin « Maven[13] » entre alors en jeu pour réaliser la manipulation et la vérification du code. La réussite de cette étape est matérialisée par la création d'une unité de déploiement sous forme de fichier jar, appelé bundle.

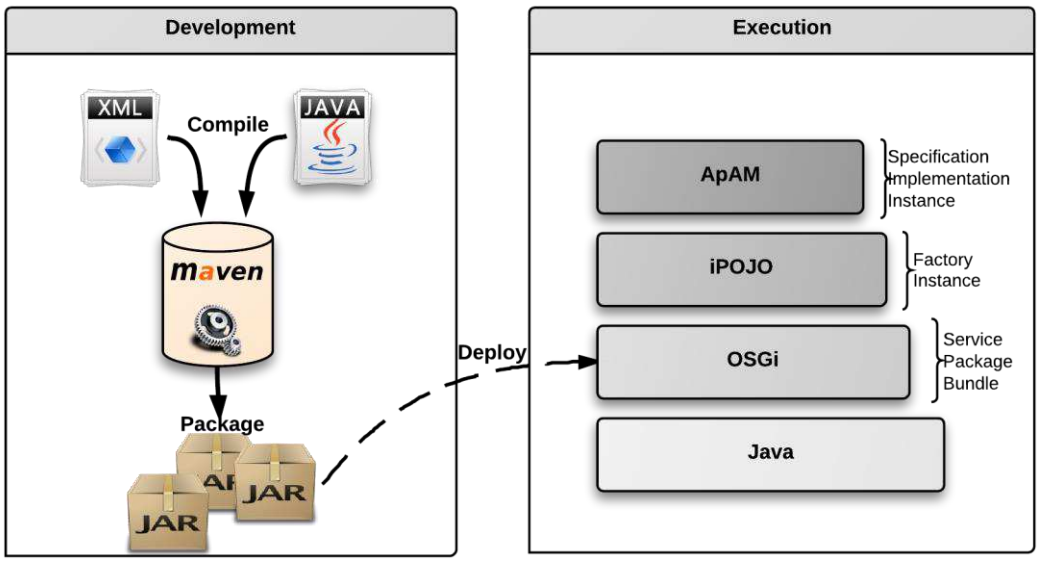


Figure 39 : Vue globale d'implémentation

Lors de l'exécution, les bundles vont être déployés dynamiquement sur la plate-forme OSGi™, les différents composants contenus dans un bundle seront alors réifiés dans ApAM. La construction de l'application peut alors commencer. Dans la suite du chapitre, nous détaillerons ces différentes étapes.

2. TECHNOLOGIES DE BASE

2.1. LA PLATE-FORME OSGI™

À l'origine, la plate-forme OSGi™ [Osgi00a] est une spécification qui vise des passerelles résidentielles et industrielles, dans le but de gérer et d'administrer les équipements réseaux et la mise à jour dynamique des applications. Cependant, son utilisation actuelle comprend aussi bien la gestion de « plugins » dans des outils de développement comme Eclipse [Sugr08], dans des serveurs d'applications comme Jonas [Ow00b] et GlassFish [Orac00c], ou encore dans des systèmes embarqués (téléphone portable) et dans des voitures BMW [Mich03].

Grâce à ces unités de déploiement, appelé bundle, OSGi™ est capable de déployer indépendamment les différentes parties d'une application. La plate-forme décrit un cycle de vie complet pour ces bundles : ils peuvent être démarrés, arrêtés, installés, désinstallés et peuvent être mis à jour sans redémarrer la plate-forme OSGi™.

[13] http://maven.apache.org/

Le premier découplage entre les bundles dans OSGi™ est obtenu grâce à l'utilisation de langage objet dans son implémentation (comme le langage Java) et l'utilisation des paquetages ou « *packages* ». En effet, un bundle peut importer ou exporter des packages. Les liaisons de packages sont gérées par la plate-forme à l'exécution.

L'utilisation de services pour interagir entre les entités de l'application constitue un second niveau de découplage. Les services sont décrits par des interfaces (Java dans le cas de l'implémentation Apache Felix [Apac00a]). Ce mode d'interaction permet de rendre l'application flexible notamment grâce à l'utilisation de registre de services, mais il est difficile à utiliser car il demande une grande connaissance d'OSGi™.

L'architecture d'une plate-forme OSGi™ est organisée en couches (voir Figure 40) :

- **La couche service**, quin permet de connecter les bundles dynamiquement en offrant des mécanismes de publication de recherche et de liaisons.
- **La couche cycle de vie,** qui définit l'API pour installer, démarrer, arrêter, mettre à jour et désinstaller des bundles.
- **La couche module,** qui a en charge de gérer les imports et exports de code (packages) entre bundles.
- **La couche sécurité,** qui permet de gérer les aspects liés à la sécurité.

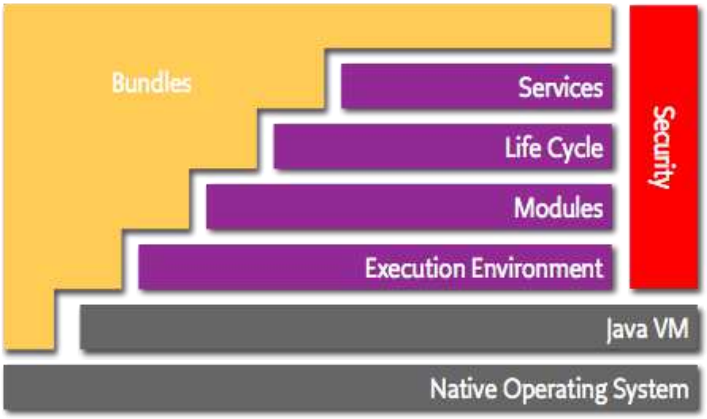


Figure 40 : Architecture OSGi™ en Java [Osgi00b]

Le choix d'OSGi™ est motivé par sa capacité à déployer et mettre à jour dynamiquement des bundles, ce qui est une propriété importante pour faire évoluer dynamiquement des applications ubiquitaires. C'est pour cette raison que la plupart des modèles à composants présentés dans l'état de l'art s'appuient sur cette plate-forme pour gérer le dynamisme. Ce choix s'est fait aussi du fait car l'équipe ADELE possède une forte expertise de la technologie OSGi™.

2.2. APACHE FELIX IPOJO

Le modèle à composants iPOJO présenté dans notre état de l'art a pour objectif de simplifier le développement d'applications à base de services dynamiques. Il a pour objectif principal de simplifier la gestion des services dans OSGi™. La membrane d'iPOJO permet d'automatiser la gestion des dépendances de services. À la compilation, iPOJO introspecte les méthodes et les propriétés du code Java, et injecte du (byte) code Java qui permet de gérer automatiquement les dépendances entre services. Pour iPOJO, la gestion des dépendances dynamiques est un besoin non-fonctionnel parmi d'autres [Esco08]. iPOJO propose divers mécanismes d'extension et la plupart les fonctionnalités d'iPOJO sont extensibles, y compris la gestion des liaisons entre composants.

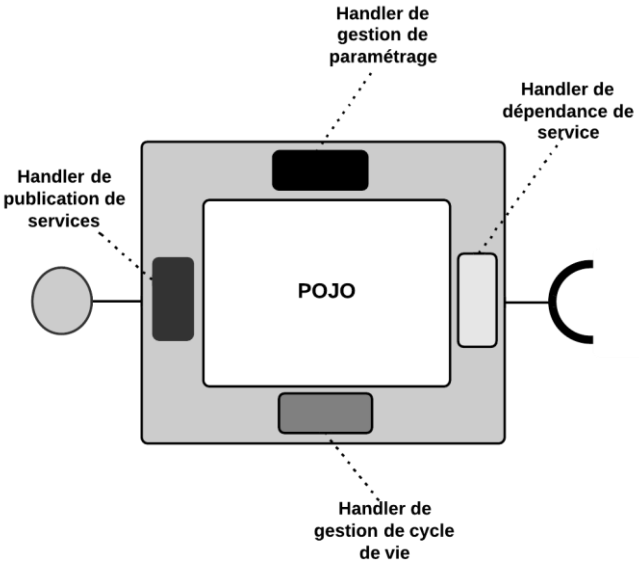


Figure 41 : Architecture d'un composant iPOJO

Le mécanisme d'extension d'iPOJO se base sur le concept de « *handlers* » qui sont aussi des composants iPOJO. Ils permettent la construction de POJOs, c'est-à-dire de programmes qui gèrent seulement les aspects métiers. Dans notre implantation, nous nous sommes appuyés sur les capacités d'extensibilité d'iPOJO. Chaque type de composant ApAM (*specification, implementation, instance*, etc.) est implanté sous la forme d'un composant iPOJO et pourra profiter des *handlers* iPOJO existants. Dans iPOJO, chaque définition de composant est associée à une fabrique (*factory* en anglais). Cette fabrique prend en charge l'instanciation du conteneur et de l'ensemble des *handlers* du composant. Nous nous appuyons sur ce mécanisme pour définir les nouveaux types de composants ApAM.

3. DÉTAILS D'IMPLÉMENTATION

ApAM est réalisée en couche. Ce découpage permet de contrôler l'évolution de notre framework. Chacune des couches a une fonctionnalité précise :

- **ApForm :** est en charge de lier les concepts d'ApAM avec la plate-forme d'exécution. C'est grâce à cette couche que les concepts d'ApAM, d'OSGi™ et d'iPOJO sont reliés. Elle permet de (i) réifier les entités en exécution en utilisant les concepts d'ApAM et de (ii) remonter les interfaces de contrôle vers les couches hautes d'APAM. Cette couche est extensible et permet d'intégrer d'autres plates-formes d'exécution. L'intégration se fait en fonction du degré d'extensibilité de la plate-forme d'exécution.
- **ApAm Core :** contient la description des concepts de base d'ApAM. Il s'agit de toutes les entités décrites dans le modèle à composants, ainsi que des interfaces nécessaires pour interagir avec ApFom et les managers.
- **Managers :** ils sont en charge de gérer les espaces de résolution. ApAM fournit un ensemble de managers pour un certain nombre d'espaces de résolution, mais il est facile d'ajouter de nouveaux managers pour de nouveaux types d'espaces.

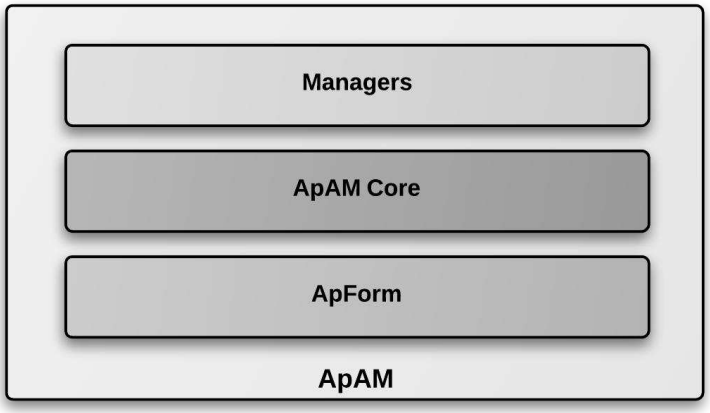


Figure 42 : Les couches ApAM

3.1. SPÉCIALISATION DES COMPOSANTS IPOJO & INTROSPECTION

Pour pouvoir réutiliser les différents *handlers* déjà présents dans iPOJO, un composant ApAM est vu comme un composant iPOJO et dispose donc d'une fabrique. Un composant ApAM hérite des fonctionnalités de la fabrique iPOJO : « *org.apache.felix.ipojo.ComponentFactory* ».

Grâce à cette extension, ApForm réifie et maintient un modèle qui représente l'architecture d'exécution actuelle des applications dans ApAM. Ce modèle est lié causalement à l'architecture du système en cours d'exécution (sur la plate-forme sous-jacente OSGi™/iPOJO) : toute modification du modèle cause une modification de l'architecture en cours d'exécution et toute modification de l'architecture en cours d'exécution provoque une modification du modèle. ApAM fournit une API qui permet de consulter, naviguer et même modifier l'architecture courante lors de l'exécution. Cette API est surtout utilisée par les managers ApAM et par l'administrateur de la plate-forme.

Il faut noter que cette extension d'iPOJO n'est pas triviale. En effet, le modèle de composant d'ApAM est très différent de celui d'iPOJO, en particulier pour les spécifications et les composites ApAM car iPOJO fait l'hypothèse qu'un composant est toujours associé à une classe Java et à une fabrique, ce qui n'est pas le cas pour les spécifications et les composites.

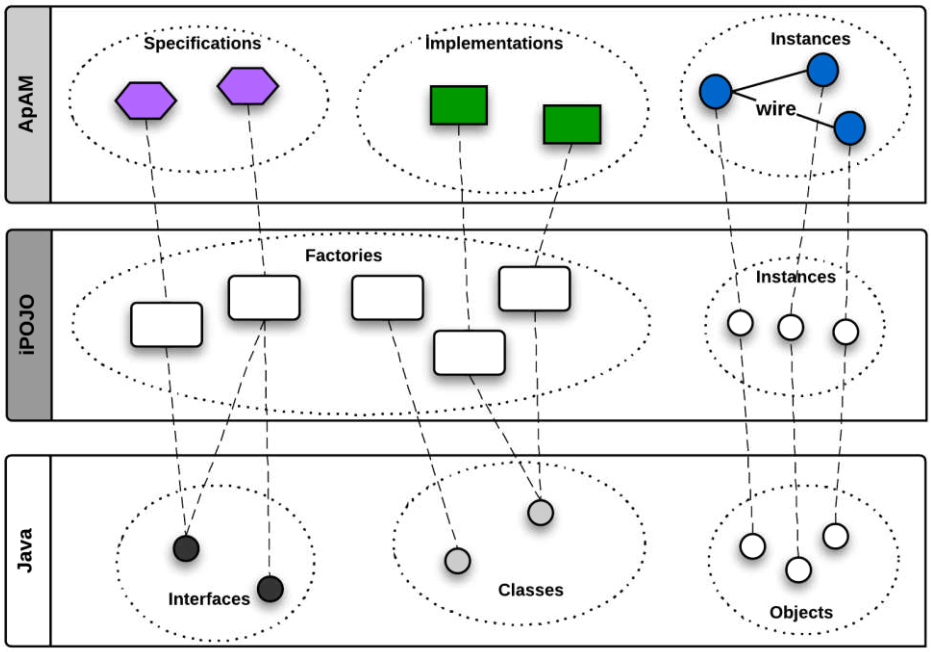


Figure 43 : Introspection dans ApAM

3.2. MANIPULATION, VÉRIFICATION ET INJECTION

ApAM s'appuie sur la machine d'injection d'iPOJO. Nous avons réalisé un plugin « *Maven* » appelé « *apam-maven-plugin* » qui intervient lors du packaging des composants. Ce plugin utilise la manipulation iPOJO. Cette manipulation transforme les classes déclarées dans la description des composants en lui injectant du byte code : on parle de « classe manipulée ». Elle permet d'encapsuler les accès aux champs et aux méthodes d'une classe par des appels de méthodes déléguées au conteneur du composant qui peut alors superviser les objets. Ce sont les classes manipulées qui vont être utilisées à l'exécution. Elles permettent ainsi d'intercepter les accès aux champs et aux méthodes. Ainsi, à chaque accès à un champ déclaré comme étant une dépendance de service, ApAM est appelée pour réaliser la résolution de la dépendance requise, puis ApAM injecte cet objet dans le champ intercepté de la classe manipulée.

Le plugin ApAM vérifie que la déclaration des composants est bien cohérente avec le code Java et avec la déclaration des autres composants. En effet, même avant l'exécution, le contenu de la description est validé par le plugin qui vérifie :

- **La validité des propriétés :** la valeur des propriétés est vérifiée par rapport au type déclaré pour chaque propriété. Par exemple, quand il s'agit d'un type énuméré, les valeurs des propriétés doivent alors être comprises dans cet ensemble.
- **Le lien entre composants :** la vérification concerne aussi bien les liens de type « *implements* » et « *instanceof* » que les composants décrits comme dépendances. Nous nous assurons que les propriétés sont bien héritées, que les interfaces sont bien implémentées, que les messages sont bien fournis, etc.
- **Le diagnostic des filtres des contraintes et préférences :** avant l'exécution il paraît impossible de vérifier la validité des contraintes, surtout quand il s'agit d'un environnement imprévisible. Mais dans certain cas, l'interprétation des filtres avant l'exécution permet de détecter des erreurs de programmation qui peuvent être corrigées avant l'exécution. Ceci minimise le risque d'erreur à l'exécution. Dans ApAM, les déclarations de dépendances ainsi que leurs filtres sont évaluées par rapport aux

composants destinations. Les filtres sont validés par rapport aux valeurs possibles que peuvent avoir ce composant.

- **La cohérence de description des composites :** la validation se fait de la même manière qu'un composant primitif. De plus, elle prend en compte le raffinement des contraintes de dépendances et les stratégies de résiliences.

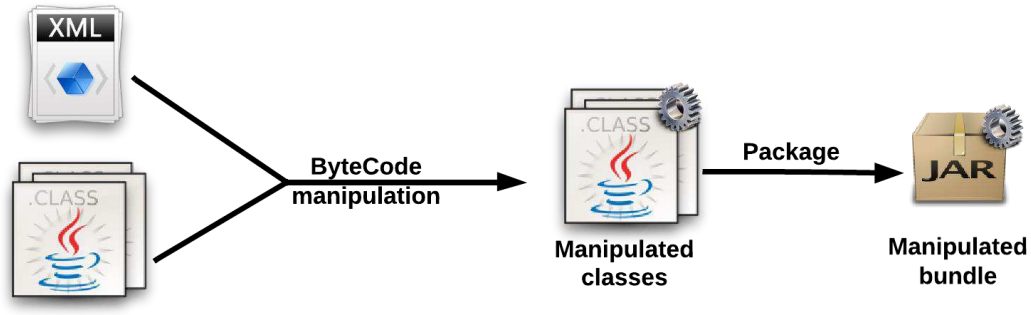


Figure 44 : Processus de manipulation iPOJO

3.3. GESTION DES DÉPENDANCES

ApAM supporte deux types de dépendances : les services et les messages. Malgré les différences apparentes entre ces deux mécanismes d'interaction, notre implémentation combine les deux modes de manière homogène : les dépendances sont résolues de la même manière pour les services et les messages.

3.3.1. DÉPENDANCES DE SERVICES

Les services dans ApAM sont définis par une interface Java, de la même manière que dans OSGi™ et iPOJO. C'est le mode d'interaction de base d'OSGi™. Celui-ci s'inspire du modèle requête-réponse et les liens entre composants ne sont pas explicites. La liaison est représentée uniquement par le champ qui référence l'interface requise. Grâce aux mécanismes d'injection, la valeur de ce champ est donc une référence Java vers l'objet implémentant l'interface : un champ vide signifie que la dépendance n'a pas été résolue.

Par contraste, dans ApAM, la dépendance est représentée explicitement par un objet de type wire créé pour chaque dépendance résolue et inversement, ce qui permet l'introspection des liaisons entre composants. Il devient alors possible de contrôler l'architecture en exécution en manipulant les wires entre composants.

Un autre changement majeur concerne la publication des services. En effet, les interfaces de services décrits dans ApAM ne sont pas publiées dans le registre OSGi™ mais maintenus par ApAM, dans le but d'offrir plus de contrôle lors du choix de fournisseur. Cependant, il est possible pour les composants de publier leurs services dans le registre OSGi™ pour qu'ils soient utilisés par les applications dites legacy.

3.3.2. DÉPENDANCES DE MESSAGES

Les dépendances de messages dans ApAM sont basées sur le modèle producteur-consommateur. Ce modèle est particulièrement adapté pour les applications qui communiquent avec des équipements physiques [MaDe05], c'est le cas des applications ubiquitaires. Cela est dû essentiellement à la difficulté de synchroniser la production des données par les capteurs et leur consommation par les applications.

L'implémentation des dépendances des messages dans ApAM s'appuie sur la spécification WireAdmin d'OSGi™. Cette spécification décrit les mécanismes nécessaires pour connecter des producteurs et des consommateurs de données. Elle décrit les concepts suivants :

- **Le producteur :** est en charge de générer les données.
- **Le consommateur :** reçoit les données générées par le producteur.
- **Le Wire** : est un objet créé par le service WireAdmin qui représente l'association entre producteur et consommateur.
- **Le service WireAdmin :** est le service qui fournit les fonctionnalités permettant de créer, mettre à jour et détruire les objets wire.

Dans WireAdmin, la liaison entre producteur et consommateur est réalisée manuellement en appelant le service WireAdmin.

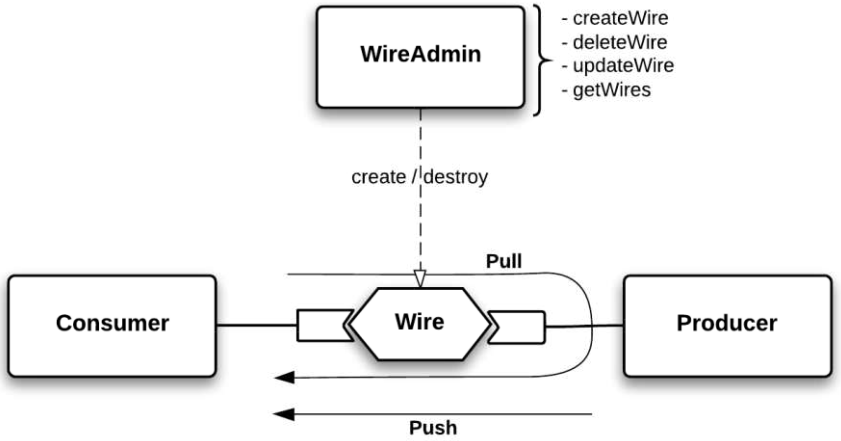


Figure 45 : OSGi™ WireAdmin

ApAM s'appuie sur ces concepts pour implémenter les dépendances de message, mais les producteurs et les consommateurs sont décrits simplement de manière abstraite et la liaison est réalisée de manière automatique et dynamique.

Un producteur de message dans ApAM doit déclarer (dans sa description XML) les types de messages qu'il produit. Cette déclaration doit être associée à une méthode du code de l'implémentation qui a comme valeur de retour un objet du type du message produit. Pour illustrer l'utilisation des messages, nous reprenons l'exemple de notre scenario (Chapitre 5§3). Un détecteur de présence peut être vu comme un producteur de message, sa spécification comportera alors l'information suivante :

```
<specification name="PresenceSensor " message="fr.liglab.apam.presence.SensedMessage ">
 ...
</specification>
```

Le code métier qui gère le capteur de présence devra produire des messages de type « *fr.liglab.apam.presence.SensedMessage*». Dans la description de l'implémentation, les types de messages produits peuvent être déduits en inspectant le code Java. Il n'est donc pas obligatoire de déclarer le type de message produit :

```
<implementation name="PresenceSensorImpl "
                classname="fr.liglab.apam.presence.PresenceSensor"
                push="sensed"   specification="PresenceSensor " >
              …
</implementation>
```

La classe *« fr.liglab.apam.presence.PresenceSensor »* associée à la description doit contenir la méthode suivante « *sensed* » :

```
public SensedMessage sensed (…);
```

À l'exécution, chaque fois que la méthode *« sensed »* est appelée, ApAM considère la valeur de retour de la méthode « *sensed* » comme la production d'un nouveau message de type *« fr.liglab.apam.presence.SensedMessage ».* Ceci est vérifié dès la compilation de la classe. La méthode doit renvoyer un objet du même type que celui déclaré dans la description. Par contre, il n'y a aucune contrainte sur les paramètres de la méthode *« sensed »*. Enfin, pour permettre la production d'un ensemble de message, ApAM s'appuie les types génériques. Ainsi, une méthode qui produit un ensemble de message est déclarée comme suit :

```
public Set<SensedMessage> sensed (…);
```

Afin de profiter des deux modes d'interaction entre les producteurs et les consommateurs, ces derniers peuvent consommer des messages en mode PULL ou en mode PUSH. La consommation de message en pull est déclarée comme un appel de service ce qui permet d'avoir une politique commune pour la résolution de dépendances de services et des messages (voir §3.3.3 ci-dessous).

Les consommateurs en mode pull doivent utiliser des champs de type *« java.util.Queue »* alors que ceux qui consomment en push doivent déclarer des méthodes dont le paramètre est du même type que celui du message à consommer. Toujours d'après le scénario (Chapitre 5 §3) où le *LIGPresenceSensorDriver*, consomme les messages du *PresenceSenor*, l'implémentation en pull est alors décrite comme :

```
<implementation name=" LIGPresenceSensorDriver "
                    classname="fr.liglab.apam.driver.PresenceSensorDriver" ... >
  <dependency id= "presence-device" specification="PresenceSensor" >
       <message pull="queueSensedMessage"/>   <!-- pulled message __>
  </dependency>
</implementation>
```

Le code associé à cette description doit alors comporter la déclaration suivante :

```
Queue<SensedMessage> queueSensedMessage;
```

Le type de message peut être extrait à partir du type générique décrit. L'objet *« Queue »* est spécial dans ApAM. Il est instancié automatiquement lors de la première utilisation. La queue sera ensuite remplie par les nouveaux messages qui arrivent au consommateur. Cette queue peut être dans trois états :

- **Non vide :** il existe des messages qui n'ont pas encore été lus par le consommateur.
- **Vide :** dans le cas où tous les messages ont été consommés ou aucun nouveau message n'a été généré par les producteurs présents.
- **Nul** : dans le cas où il n'existe aucun producteur de type de message souhaité par le consommateur. Dès qu'un producteur du type de message souhaité « *apparaît* » la file de message est instanciée automatiquement et la dépendance vers ce producteur est résolue.

Il est du ressort du consommateur de gérer cette file de messages. Cependant, il est envisagé d'ajouter des propriétés dans la description pour permettre au développeur de choisir la taille et l'implémentation qu'il souhaite pour sa file de messages.

Une implémentation en PUSH est déclarée de la manière suivante :

```
<implementation name=" LIGPresenceSensorDriver"
                    classname="fr.liglab.apam.driver.PresenceSensorDriver" ... >
   <dependency id= "presence-device" specification="PresenceSensor" >
         <message push="notifyNewPresence"/>   <!-- pushed message -->
  </dependency>
</implementation>
```

La classe d'implémentation associée à cette description (*fr.liglab.apam.driver .PresenceSensorDriver*) doit définir une méthode dont le nom est « *notifyNewPresence* ». Cette méthode doit avoir pour argument un champ simple ou un ensemble avec le type des messages à consommer :

```
public void notifyNewPresence (SensedMessage message) { ....}
```

À l'exécution, cette méthode sera appelée automatiquement dès qu'un message est produit par l'un des producteurs de message. Pour permettre au producteur et au consommateur d'ajouter ou de consulter les métadonnées associées au message, il est possible d'encapsuler les données dans un objet ApAM du type *« fr.imag.adele.apam.message.Message ».* L'encapsulation des données utilise elle aussi des types génériques, nous avons alors :

```
public void notifyNewPresence (Message<SensedMessage> message) {....}
```

Cet objet permet d'identifier des informations telles que l'identité du producteur ou la date de production du message ainsi que d'autres informations.

3.3.3. RÉSOLUTION DES DÉPENDANCES

La résolution des dépendances dans ApAM est réalisée lors de l'exécution. Par défaut, ApAM promeut une résolution paresseuse et opportuniste, les dépendances sont alors résolues à la demande et uniquement si elles sont utilisées. La résolution ne fait pas de distinction entre les dépendances de messages et de services : c'est la même mécanique qui est appliquée indépendamment du mode d'interaction.

Par défaut, une dépendance est résolue par le manager noyau appelé APAMMAN qui a pour objectif de chercher une instance de composant qui satisfait la dépendance parmi tous les composants présents sur la plate-forme. Pour satisfaire une dépendance, l'instance du composant est choisie en fonction de ses propriétés et des contraintes de la dépendance. Plus la dépendance est exprimée à haut niveau d'abstraction plus APAMMAN aura de liberté pour choisir une instance de composant satisfaisante. Par exemple, si une dépendance est exprimée par une interface, APAMMAN considèrera toutes les implémentations et instances fournissant cette interface. Si la dépendance est exprimée vers une spécification, l'algorithme de résolution d'APAMMAN commencera par sélectionner les implémentations conformes à cette spécification et si la dépendance est exprimée vers une implémentation, seule cette implémentation sera considérée. Dans tous ces cas, s'il n'existe pas d'instance de l'implémentation choisie, une instance sera créée. Par contre, pour une dépendance décrite par le nom de l'instance, APAMMAN se contentera de sélectionner l'instance indiquée.

La résolution des dépendances dans ApAM est extensible, c'est ce qui permet de gérer plusieurs espaces de résolution. La résolution de base dans ApAM peut être étendue voire remplacée. En effet, la résolution est organisée autour d'un système de « manager », chacun d'eux peut intervenir dans la

résolution d'une dépendance, d'ailleurs APAMMAN n'est qu'un manager parmi d'autres. À chaque exécution de la plate-forme, nous pouvons choisir l'ensemble des managers que nous souhaitons déployer. ApAM joue le rôle d'arbitre pour l'ensemble des managers. Un algorithme organise la résolution entre les différents managers participant à la résolution. Cet algorithme est de type « Best-Effort » car nous nous contentons de prendre la première solution proposée par un manager et qui satisfait toutes les contraintes décrites par les autres managers. Cependant, il est possible d'implémenter de nouveaux algorithmes pour la gestion des managers. Notre algorithme de résolution est découpé en deux phases :

- **La première phase :** consiste à parcourir l'ensemble des managers existants. Chacun de ces managers peut exprimer s'il souhaite contribuer à cette résolution et dans ce cas, il donne son ordre dans la résolution et donne l'ensemble des contraintes qu'il souhaite ajouter lors de la résolution de cette dépendance.
  Cette première phase consiste donc à fournir la liste (ordonnée) des managers qui seront impliqués et l'ensemble des contraintes additionnelles.
- **La seconde phase :** appelle chaque manager impliqué avec l'ensemble des contraintes calculées. C'est la réponse du premier manager qui sera prise en compte, celui-ci devra retourner uniquement un composant qui vérifie toutes les contraintes.

Les managers intervenant dans la résolution des dépendances implémentent une interface spécifique *« DependencyManager »* (voir Annexe 1) qui étend elle-même l'interface « Manager » décrite dans le chapitre précédent (Chapitre 5 §6.2.3 Gestion des espaces de résolution). Des exemples d'implémentation de l'interface *« DependencyManager »* sont disponibles depuis le site web d'ApAM[14].

3.4. NOTIFICATIONS DU CONTEXTE D'EXÉCUTION

Nous supposons que le contexte d'exécution des applications en exécution au-dessus d'ApAM est en constante évolution. Les composants des applications peuvent apparaître, disparaître ou être mis à jour à tout moment. Tous ces changements dans le contexte d'exécution sont capturés par ApAM qui propose un mécanisme extensible pour permettre à des managers de réagir à cette évolution.

Les notifications de l'évolution du contexte d'exécution dans ApAM sont séparées en trois catégories :

- **Notifications de dynamisme** qui permettent d'être notifiées de l'évolution dynamique des composants et des wires. Pour cela, ApAM fournit une spécification permettant de s'abonner aux notifications (voir Annexe 2). Un gestionnaire de dynamisme est notifié à chaque création et destruction d'un composant et/ou d'un wire.
- **Notifications de propriétés** qui permettent d'être notifiées des mises à jour des propriétés des composants en exécution. ApAM fournit une spécification permettant de s'abonner à l'ajout, la suppression et la modification des propriétés des composants en exécution (voir Annexe 3).
- **Notifications de demandes de résolution** qui permettent d'être notifiées des nouvelles demandes de résolution (voir § 3.3.3 ci-dessus).

Ces types de notifications peuvent avoir différentes utilisations. ApAM fournit une implémentation des interfaces présentées précédemment. En effet, ApAM fournit un manager appelé DYNAMAN capable de gérer l'évolution dynamique des applications. DYNAMAN est capable de maintenir à jour les dépendances des composants en les adoptant suite à l'apparition, à la disparition des composants ou à la mise à jour des propriétés. Évidemment, il est possible d'étendre le comportement décrit par DYNAMAN (ou même de le remplacer), il suffit alors de fournir d'autres implémentations des interfaces *PropertyManager* et *DynamicManager*.

[14] http://adeleresearchgroup.github.io/ApAM/

3.5. COMMANDE D'ADMINISTRATION

ApAM fournit un ensemble de commandes permettant à l'administrateur de la plate-forme d'exécuter ou de superviser l'exécution des applications en cours d'exécution. Ces commandes sont aussi utiles aux développeurs des applications. Elles permettent d'introspecter la plate-forme d'exécution.

Ci-après la liste des commandes de la console disponibles sur la plate-forme.

Commande	Description
l* ou launch *<component_name>	Déclenche le mécanisme de résolution pour trouver une instance du composant <*component_name*>. Cela peut avoir pour conséquence le déploiement, l'instanciation ou la création de proxy (selon les managers présents).
inst	Liste l'ensemble des instances ApAM en exécution.
inst *<instance-name>*	Affiche les informations concernant l'instance <*instance-name*>.
implem	Liste l'ensemble des implémentations ApAM en exécution.
implem *<implementation-name>*	Affiche les informations concernant l'implémentation <*implementation-name*>.
spec	Liste l'ensemble des spécifications ApAM en exécution.
spec *<specification-name>*	Affiche les informations concernant la specification <*specification-name*>.
compo	Liste l'ensemble des composites ApAM en exécution.
compo *<composite-name>*	Affiche les informations concernant <*composite-name*>.
changeproperty *<component-name> <property-name> <property-value>*	Permet de mettre à jour dynamiquement la valeur de la propriété <*property-name*> avec la valeur <*property-value*> pour le composant <*component-name*>.
pending	Liste l'ensemble des dépendances en attente (ces dépendances n'ont pas pu être résolues, elles sont en attente d'un composant).
displayWires *<instance-name>*	Liste ensemble des *wires* créés par ApAM pour l'instance <*instance-name*>.

4. IMPLÉMENTATION D'OBRMAN

Chaque espace de résolution dans ApAM est dirigé par un Manager. Dans cette partie nous détaillons un exemple d'implémentation : le manager OBRMAN. Il est en charge de résoudre les dépendances dans l'espace de résolution de type OBR.

4.1. PRÉSENTATION D'OBR

OSGi™ Bundle Repository est une proposition de spécification RFC-0112 d'OSGi Alliance [Osgi05]. Il s'agit d'une plate-forme de provisionnement de ressources (des bundles). Cette spécification définit un dépôt utilisable directement depuis un Framework OSGi™ pour déployer des bundles à partir d'un dépôt de code distant. La spécification définit la structure du fichier XML qui décrit le contenu du dépôt ainsi que le service OSGi™ permettant l'accès au dépôt depuis une plate-forme OSGi™.

Cette spécification (née dans notre équipe) est restée longtemps sous la forme d'une proposition et nous avons vu apparaître différentes implémentation d'OBR. Ces implémentations sont légèrement similaires mais chacune se démarque par des propriétés spécifiques. Parmi les implémentations les plus connues nous trouvons : Apache Felix OSGi™ Bundle Repository (appelé OBR, implémentation réalisée par notre équipe) [Apac00c] ou encore Equinox P2 [Ecli00] utilisé par Eclipse Update Sites. Cependant ces implémentations partagent les buts principaux de la spécification :

- Simplifier le déploiement des bundles sur les plates-formes OSGi™.
- Encourager la construction de bundles indépendants et réutilisables par la communauté OSGi™.

Notre choix s'est porté sur Apache Felix OSGi™ Bundle Repository (OBR) pour sa facilité d'intégration et d'extension, ainsi que pour la présence d'outils permettant d'étendre la description du dépôt code à moindre effort. OBR a l'avantage de rassembler la description du dépôt dans un seul fichier, généralement appelé *« repository.xml »* (ce nom peut être remplacé), alors que P2 doit jongler entre deux fichiers *« content.xml »* et *« artificats.xml ».* Finalement, la description du dépôt P2 ne supporte pas l'expression de contraintes et elles ne sont pas prises en charges par le « *Resolver* ». Dans OBR, les contraintes LDAP sont supportées, limitant ainsi le risque de déployer des ressources incorrectes sur la plate-forme.

4.2. INTÉGRATION OBR ET APAM

OBRMAN permet de résoudre les dépendances entre composants en utilisant des dépôts de code OBR. Pour intégrer ces dépôts dans la résolution d'ApAM, le manager OBR (OBRMAN) s'appuie sur l'implémentation Apache Felix OBR pour déployer les bundles contenant les composants nécessaires pour satisfaire une demande de résolution de dépendances. Cette intégration est réalisée en deux parties :

- La première partie consiste à étendre la description des dépôts OBR avec les concepts d'ApAM.
- La seconde partie consiste à étendre l'algorithme de résolution d'OBR pour prendre en compte les contraintes de dépendances dans la résolution.

4.2.1. EXTENSION DE LA DESCRIPTION

Le dépôt OBR est basé sur le concept générique de « ressource ». Une ressource est une entité composée d'une liste de besoins « *requirement* » et de capacités « *capabilities* ». Une « *capacbility* » est décrite par son nom et un ensemble de propriétés propre à chaque type de « *capablility* ». Un « *require* » est défini par un filtre de type LDAP permettant de décrire les contraintes de la ressource, il doit indiquer aussi le nom de la « *capability* » sur laquelle le filtre doit s'appliquer. Cette description générique peut donc décrire n'importe quelle ressource informatique, la ressource de type bundle n'est qu'un type de ressources parmi d'autres.

Dans le cas d'OBRMAN nous avons étendu la ressource de type bundle. Nous avons défini le concept de *« apam-component »* comme une « *capability* » et un « *require* ». Ceci nous permet de décrire un composant ApAM quel que soit son niveau d'abstraction. Cette extension est réalisée automatiquement par le plugin *«apam-maven-plugin* ». Durant la compilation d'un bundle, notre plugin génère toute l'information nécessaire pour enrichir la description du dépôt de code. Cette information est ensuite placée dans un fichier, appelé *« obr.xml ».* Le contenu de ce fichier est automatiquement fusionné avec la description de la ressource (le bundle) dans le dépôt OBR. L'emplacement du dépôt de code peut être spécifié dans la configuration du plugin « *maven-bundle-plugin* »[15] grâce à la propriété <*obrRepository*>. Elle indique le chemin vers le fichier XML décrivant le dépôt (repository.xml).

4.2.2. EXTENSION DU RÉSOLVEUR

Pour interagir avec le dépôt de code à l'exécution, OBRMAN se base sur le service OSGi™ « *RepositoryAdmin* » (voir Figure 46). Grâce à ce service OBRMAN récupère la liste des dépôts de code ainsi que la liste de leurs ressources. Pour chaque demande de résolution soumise à OBRMAN, ce dernier parcourt la liste des ressources à la recherche du composant qui satisfait les contraintes de la demande de résolution. Puis il sélectionne la ressource qui contient le composant choisi.

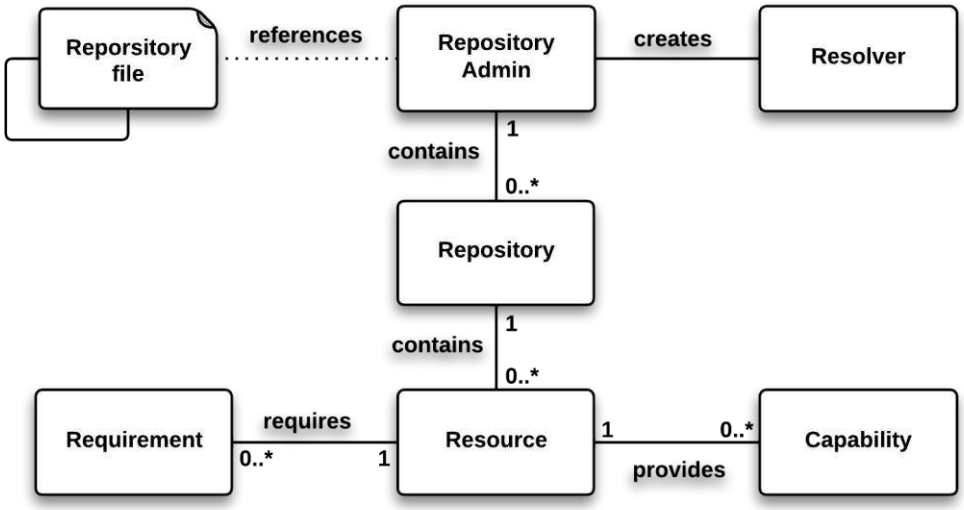


Figure 46 : OSGi RepositoryAdmin

Le déploiement de la ressource est une tâche plus complexe. En effet, il ne suffit pas de récupérer le lien vers la ressource et de l'installer dans la plate-forme. Le bundle décrit par la ressource peut contenir des dépendances vers d'autre composants ou packages qu'il faut également satisfaire pour que le processus de déploiement et d'exécution se déroule correctement. C'est pour cette raison que le déploiement de la ressource est délégué au « *Resolver* » du *RepositoryAdmin*. En fonction des bundles en cours d'exécution, celui-ci s'assure que toutes les dépendances de la ressource décrites dans les « *require* » de la ressource seront bien satisfaites. Le résolveur déploie pour cela toutes les ressources manquantes dans la plate-forme d'exécution. Si le *« Resolver »* n'arrive pas à trouver une ressource requise, le déploiement n'est pas réalisé.

[15] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html

4.2.3. OBRMAN ET LES COMPOSITES

OBRMAN est chargé de résoudre des dépendances dans l'espace de résolution constitué des dépôts OBR. Pour les composites cet ensemble de dépôt peut être différent d'un composite à un autre. Il était donc nécessaire d'étendre le *RepositoryAdmin* existant pour répondre à ce besoin.

Chaque composite peut décrire dans un fichier l'ensemble de ses dépôts (voir Chapitre 5 §6.2.4). OBRMAN crée pour chaque composite un « *Resolver* » adapté, il contient uniquement les ressources obtenues par la liste des dépôts présents dans son modèle. Ceci nous permet d'une part de protéger le contenu et l'accès aux dépôts du composite et d'autre part de ne prendre en compte que les ressources disponibles dans le dépôt du composite en question.

5. CHIFFRES ET SYNTHÈSE

Nous avons présenté dans ce chapitre quelques détails concernant l'implémentation actuelle d'ApAM. Pour conclure nous proposons quelques chiffres concernant l'implémentation.

ApAM est divisée en 14 sous-projets (voir Tableau 9), eux-mêmes classé en 5 catégories. Une grande partie du développement d'ApAM a été consacrée à l'élaboration de tests d'intégration afin de stabiliser la plate-forme. Le tableau suivant présente le nombre de lignes de code par projet.

Catégories	Projets	Nombre de lignes de code
Noyau	Déclaration	1448
	Cœur	8728
ApForms	ApForm iPOJO & OSGi	2822
Managers	OBR Manager	926
	Distribution Manager	1428
	Manager d'historique	271
	Manager de conflits	477
Outils	Plugin Maven	2845
	ApAM Shell	607
	ApAM JMX	469
Tests	Noyau	10697
	ApForms	
	Distribution Manager	
	OBR Manager	

Tableau 9 : Nombre de lignes code

Le noyau ApAM est composé des concepts de base d'ApAM. On y trouve tous les mécanismes d'injection et d'interception. Il constitue la machine d'exécution d'ApAM. Après compilation, le cœur d'ApAM ne « pèse » que 186 ko. L'ApForm iPOJO & OSGi™ sont actuellement encapsulés dans un même projet, ceci s'expliquant par la proximité des deux technologies.

Les managers ont été développés pour intégrer de nouveaux espaces de résolution dans ApAM. Nous avons déjà présenté les managers OBR et le manager de distribution. Le Manager d'historique permet de faire persister les états de la plate-forme d'exécution, ces informations sont enregistrées dans une base de donnée de type *MongoDB*, le but étant de réaliser une analyse a postériori de l'évolution dynamique des applications d'ApAM. Le manager de conflits a été réalisé dans le but de résoudre les conflits d'accès entre les composants ApAM et les équipements physiques, ce travail est en cours de validation au sein du projet Open The Box (voir Chapitre 7).

Différents outils ont été développés. Nous avons déjà abordé le plugin Maven et le shell ApAM. L'outil JMX permet d'exposer les fonctionnalités d'ApAM en utilisant le protocole JMX, permettant ainsi de réaliser une administration à distance de la plate-forme.

CHAPITRE 7
ÉVALUATION ET EXPÉRIMENTATION

1. INTRODUCTION

Dans le chapitre précédent, nous avons décrit l'implantation d'ApAM et montré comment cette plate-forme permet de créer des applications ubiquitaires en définissant un nouveau modèle à composants orienté services.

Le présent chapitre s'intéresse à la validation d'ApAM. Dans un premier temps, nous présenterons les différents projets de recherche collaborative au niveau Européen mais aussi dans le cadre de projets industriels dans lesquelles ApAM est impliqué, nous illustrerons alors les cas d'utilisation permettant de décrire le rôle de notre plate-forme d'exécution. Dans un second temps, nous évaluerons la plate-forme ApAM avec une comparaison fonctionnelle par rapport à d'autres plates-formes existantes.

2. CADRE DU PROJET

ApAM est un projet en cours de validation dans le cadre de deux grands projets : Open The Box et AppsGate. Ces projets sont à mi-parcours et définissent le carnet de route de l'évolution d'ApAM.

2.1. OPEN THE BOX

Le Laboratoire Informatique de Grenoble est l'unique partenaire académique du projet « **Open The Box ».** L'équipe ADELE est la seule équipe du LIG qui collabore dans ce projet financés par le programme *« Investissements d'avenir* [16]*»* de l'État Français. Le projet regroupe plusieurs grands industriels (France Telecom Orange, Bouygues Telecom, Delta Dore, Sagemcom, Sogeti High Tech, STMicroelectronics, IS2T) et porte sur les logiciels pour la domotique utilisant les box d'accès à Internet pour partager de services sur réseau domestique. Ces services peuvent représenter des réseaux de capteurs dans la maison comprenant des équipements de types capteurs et actionneurs.

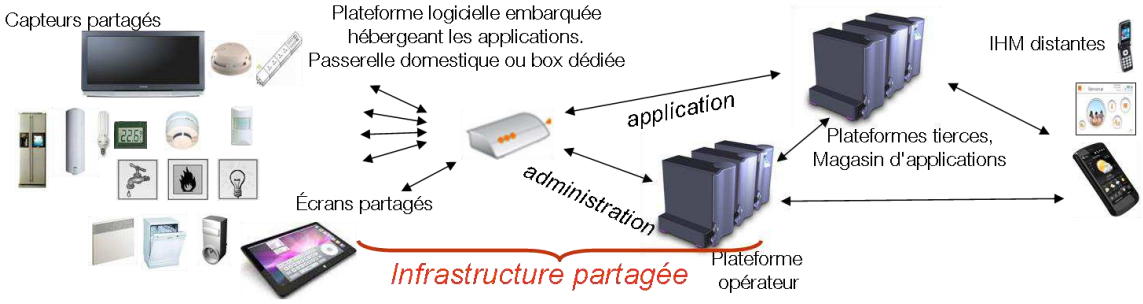


Figure 47 : Partage de services dans OTB[Oran11]

2.1.1. CAS D'UTILISATION : GESTION DE CONFLITS ENTRE APPLICATIONS EMBARQUÉES

Comme la box est l'élément central dans la maison (dans le cadre du projet), la cohabitation de différentes ressources (services et équipements) au sein d'une même infrastructure est un défi auquel s'attaque ce scénario. Nous proposons l'utilisation d'ApAM étendu pour une gestion des conflits d'accès et de partage de ressources entre les différentes applications de la plate-forme, les services et les équipements physiques de la maison.

Pour illustrer un cas de gestion de conflits, nous nous sommes inspirés du scénario de « gestion d'urgence » décrit dans [Jako11]. Le scénario a pour but de décrire les priorités entre différentes applications pour l'accès aux équipements de la maison. Il permet de valider un modèle de

[16] http://investissement-avenir.gouvernement.fr/

construction des applications qui autorise les arbitrages d'usage des équipements domotiques (i.e., capteurs/actionneurs) qui y sont connectés. Ce scénario décrit deux applications:

- **La détection d'incendie :** Cette application permet de détecter un incendie grâce à des détecteurs de fumée, puis réalise des actions pour protéger les habitants : déclenchement d'alerte sonore, activation des têtes d'extinction automatique à eau (sprinkler) et le déverrouillage des portes pour permettre l'évacuation du bâtiment.
- **La détection d'intrusion :** Cette application permet de réagir face à une intrusion dans une maison. Face à cette intrusion l'application se charge de déclencher une alarme sonore et de s'assurer que les portes de la maison sont verrouillées pour protéger les habitants. La nuit cette application verrouille les portes.

Il existe un risque que ces deux applications entrent en conflits dans le cas où une alerte incendie se déclenche la nuit ou en même temps qu'une alerte intrusion. Les deux applications vont réaliser des actions contradictoires. Par exemple, l'application incendie va déclencher le déverrouillage des portes, alors que l'application d'intrusion va activer leur verrouillage.

2.1.2. SOLUTION PROPOSÉE

Notre solution utilise les composites ApAM pour définir les silos fonctionnels (media, confort, sécurité, santé, bien-être, jeux) au-dessus des applications. L'attribution des équipements est gérée au niveau de ces « silos » fonctionnels plutôt qu'au niveau des applications elles-mêmes car celles-ci ne peuvent pas savoir avec qui elles sont en conflit, par exemple en fonction d'une priorisation de criticité.

Chaque silo se voit attribuer un état interne qui varie au cours du temps et peut déclarer posséder des équipements seulement dans certains de ces états. Par exemple pendant la nuit c'est le silo sécurité qui commande la porte d'entrée sauf si le silo « incendie » est dans l'état « actif ». Le propriétaire d'une ressource peut garantir une ressource à un silo ou directement à une application via une préemption et une réallocation des équipements (voir la Figure 48).

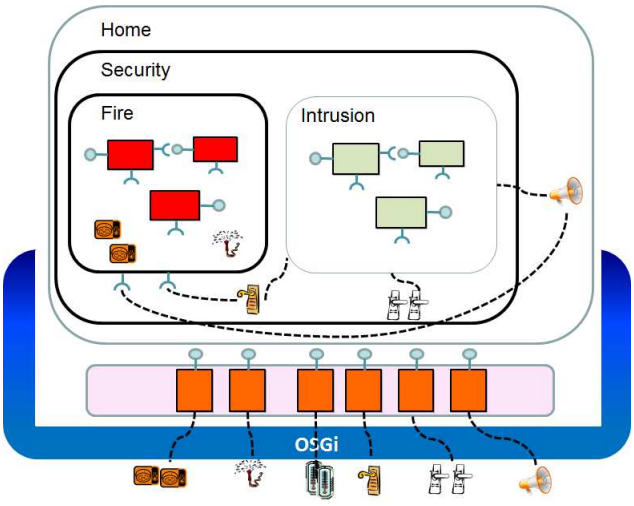


Figure 48 : Silos fonctionnels

L'objectif de notre solution est de faciliter le travail des développeurs. D'une part, l'accès aux équipements est réalisé d'une manière transparente (comme un simple service applicatif). D'autre part, la gestion de l'accès aux équipements est ignoré par les applications, elle est déclarée par les silos réalisée à un niveau au-dessus de l'application. Ceci permet de réaliser l'application simplement sans se soucier de la possibilité d'accéder à l'équipement.

Le manager mettant en œuvre cette solution : *CONFLICTMAN*, permet d'associer aux composites (et donc aux silos) de nouvelles propriétés :

- **State :** Désormais un composite est capable d'exprimer son **état (state)**. Il représente l'état du contexte d'exécution courant pour les applications contenues dans le composite. Cet état est calculé par un composant spécialisé qui est lui aussi spécifié dans la description du composite.
- **Own :** Chaque silo fonctionnel doit « posséder » le contrôle sur les équipements pour lesquelles il souhaite gérer l'accès. Le rôle de cette propriété, associée à un composite, est de définir quelles sont les instances de composant qui doivent appartenir à ce composite. Ces instances peuvent représenter des équipements. Le silo fonctionnel peut contrôler les demandes d'accès pour toutes les instances des équipements qu'il possède. Dans l'exemple ci-dessous, les verrous des portes d'entrées et de sorties (*entrance, exit*) appartiennent au composite *« security »,* qui va donc pouvoir gérer les accès vers ces équipements.
- **Grant :** Le « Grant » est une expression qui définit, en fonction de l'état du composite, à quelle dépendance l'accès à une instance est garanti. Grâce à l'expression du « *Grant* », le composite est capable d'accorder l'accès aux instances qu'il possède pour les applications contenu dans ce composite. Dans l'exemple ci-dessous, l'accès aux portes (d'entrées et de sorties) est accordé à l'application « *Fire* » quand le composite est en état d'urgence (*houseState = emergency*)

Voici un exemple de l'expression de la gestion des autorisations d'accès dans un composite :

```
<composite name="security" … singleton="true">
...
  <conflictman>
     <state implementation="HouseState" property=" houseState "/>
    ...
    <own specification="Door" property=”location” value=”entrance, exit”>
      <grant when="emergency" implementation =”Fire” dependency=”door” />
      <grant when=”threat” specification=”Break” dependency=”entranceDoor” />
    </own>
  ...
 </conflictman>
...
</composite>
```

Comme la définition du *« Own »* est décrite aux propriétés des instances, le changement dynamique des valeurs des propriétés peut provoquer le changement de l'emplacement des instances. Ainsi, une instance peut changer de silo fonctionnel en fonction des propriétés qu'elle exprime. À l'exécution, ces expressions sont gérées par le manager *CONFLICTMAN*. Ce manager intervient à chaque changement de propriétés ou demande de résolution pour s'assurer que les accès sont bien autorisés avant l'établissement de liens entre composants

Les 12 premiers mois, sur les 30 que compte le projet, viennent de s'achever. Ce projet va permettre de valider notre solution et éventuellement d'identifier d'autres besoins liés à la gestion du dynamisme.

2.2. APPSGATE

Ce projet européen CATRENE(Applications Gateway)[17] rassemble différents acteurs industrielles et universitaires : STMicroelectronics, Pace, Technicolor, NXP, 4MOD Technology, ARD, Immotronic, Ripple Motion, Simon Tech, Video Stream Network, SoftKinetic Software, Softkinetic Sensors, Vestel, Université Joseph Fourrier (dont le LIG) et Institut Mines-Telecom. Il vise à réaliser une plate-forme ouverte pour fournir des applications domestiques intégrées. Cette plate-forme est déployée dans une Set-Top-Box (STB) domestique. Celle-ci est en charge d'agréger l'ensemble des services et des équipements disponibles dans la maison.

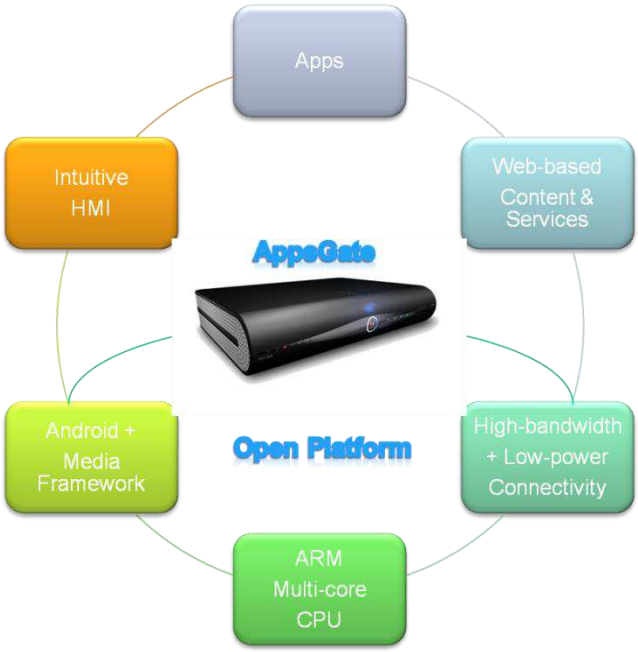


Figure 49 : Projet AppsGate (depuis les documents AppsGate)

L'objectif de ce projet est d'analyser les besoins des résidents d'un habitat intelligent. Dans le but de développer plusieurs applications informatiques qui seront ensuite installées et testées chez des particuliers. Les applications envisagées n'étant pas réellement nouvelles, l'enjeu réside dans la capacité d'intégrer l'ensemble de ces applications au sein d'une même plate-forme grand public.

2.2.1. EXPÉRIMENTATION

Dans le cadre de ce projet, ApAM est utilisé pour:

- Réifier l'ensemble des équipements et des services hétérogènes accessibles depuis la maison.
- Construire une application au-dessus d'un environnement hétérogène, réparti et dynamique. Ces applications sont implémentées sous la forme d'un composant ou d'un composite ApAM selon la complexité de l'application.
- Permettre aux différentes entités de communiquer entre elles, par des appels synchrones (appel de service) ou asynchrones (utilisation des messages).

Pour les expérimentations du projet, différents scénarios ont été imaginés à partir de l'observation du comportement des individus dans leur environnement quotidien à la maison. Les

[17] http://www.catrene.org/web/projects/projects_call456.php

scénarios mettent en jeux plusieurs applications qui interagissent entre elles pour faciliter les tâches quotidiennes des habitant, par exemple utiliser un ensemble d'équipements et de services pour aider les personnes dans les taches de réveil quotidien en associant réveil, météo, lampes, réfrigérateur, etc.

Tout comme pour le projet Open the box, les 12 premiers mois, sur les 36 que compte le projet AppsGate, viennent de s'achever. Les travaux d'expérimentation sont toujours en cours d'élaboration. Cependant, le retour des différents partenaires qui utilise ApAM dans le cadre du projet sont très encourageants.

3. ÉVALUATION FONCTIONNELLE D'APAM

Notre évaluation fonctionnelle tente de quantifier l'effort nécessaire pour la description de l'application dans chaque technologie. Trois environnements d'exécution ont été sélectionnés pour ce test : Tuscany, iPOJO et Spring.

3.1. L'APPLICATION FIBONACCI

Cette application calcule le nombre de Fibonacci [Wiki00b] où f(n) = f(n-1)+ f(n-2). Au lieu de réaliser une récursivité, nous faisons en sorte que chaque appel f(i) soit effectué vers une nouvelle instance ce qui conduit à réaliser une composition dynamique de service. Comme le montre la Figure 50, la suite définit le nombre d'instances qui doivent être créés à chaque étape. Chaque instance définit des dépendances vers les instances du prochain nombre de Fibonacci. Les instances sont exclusives, c'est-à-dire qu'elles ne peuvent pas être partagées, chaque résolution de dépendance doit donner une instance différente. Ainsi, l'algorithme génère un arbre d'instances.

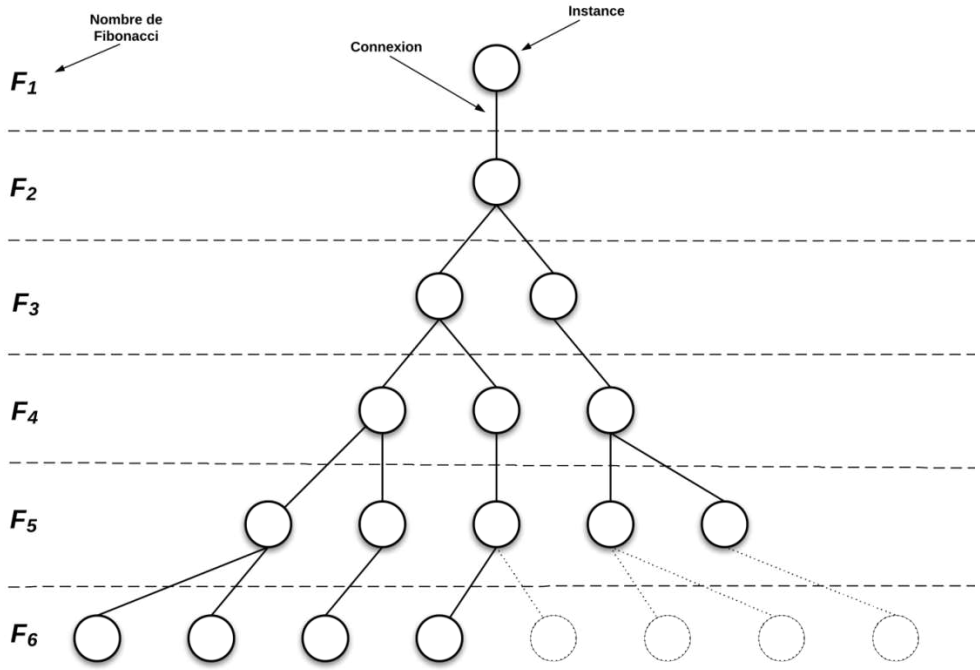


Figure 50 : Structure de l'application de test

Bien sûr, une telle implémentation est extrêmement maladroite, elle ne sert qu'à illustrer quelques fonctionnalités d'ApAM. Chaque instance est créée à la demande, ce qui permet de simuler une évolution croissante de l'application et nous permettre de d'illustrer le comportement résilient de l'application face à aux apparitions et disparitions des instances de l'application. Après un premier appel F(n) qui nous a permis d'instancier toute la chaine d'instances, nous supprimons aux hasards quelques instances, puis nous relançons un appel de F(n) depuis la racine. Cette opération va nous permettre d'illustrer la réaction de l'application face au dynamisme.

Pour chacune des plates-formes comparées nous montreront l'effort nécessaire au développeur pour la réalisation d'une telle application ainsi que leur comportement face à des variations du contexte d'exécution.

3.2. TECHNOLOGIES COMPARÉES

Pour cette comparaison, nous avons définis une partie commune à toutes les plates-formes. Cette partie commune implémente l'algorithme de récurrence de Fibonacci :

```
F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2) for n>1.
```

Cet algorithme a été implémenté de façon à ce qu'il soit commun à toutes les plates-formes comparées. Le composant principal est décrit alors de la manière suivante :

```
public class FibonacciRecursive implements Fibonacci {

   Fibonacci moins1;

   Fibonacci moins2;

   public int compute(int n) {

      if (n < 2)
         return 1;

      int returns = moins1.compute(n - 1) + moins2.compute(n - 2);

      return returns;

   }
}
```

L'ensemble du code source lié à l'implémentation de cette comparaison est disponible depuis le site web du projet ApAM[18] . Dans la suite nous allons détailler les particularités d'implémentation pour chacune des technologies testées.

3.2.1. APAM

Dans le cas d'ApAM, aucune modification du code n'est nécessaire. ApAM supporte l'instanciation automatique des composants, ainsi il suffit de décrire le composant ApAM dans un fichier XML :

```
<apam xmlns="fr.imag.adele.apam"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="fr.imag.adele.apam http://repository-
                apam.forge.cloudbees.com/release/schema/ApamCore-0.0.3.xsd">

  <implementation name="FibonacciRecursive" shared="false"
          classname="fr.imag.adele.apam.pax.test.performance.FibonacciRecursive">
    <dependency field="moins1" />
    <dependency field="moins2" />
  </implementation>
</apam>
```

Dans cette description, il suffit d'indiquer que les instances du composant décrit ne doivent pas être partagées (grâce à l'attribut *shared="false"*). La plate-forme est alors en charge d'instancier les composants nécessaires. Ces composants sont ensuite injectés dans les champs appropriés (« moins1 » et « moins2 ».

[18] http://adeleresearchgroup.github.io/ApAM/

Dans ApAM, le premier appel *F(n)* va instancier ensemble de la chaine d'appel par récursivité. Ce comportement est dû au mécanisme de résolution qui instancie automatiquement un composant (ou même le déployer) si aucune instance libre n'est disponible. Enfin, après la suppression de quelques instances, le comportement résilient va faire en sorte de recréer les instances manquantes dans la chaine d'appel.

3.2.2. iPOJO

Nous avons essayé de réaliser cette application en utilisant iPOJO, mais malgré sa simplicité, cette application est difficile à réaliser avec iPOJO. La première raison vient du fait que cette technologie ne permet pas d'instancier. On a donc dû modifier le code de l'application pour ajouter une partie permettant de réaliser explicitement l'instanciation. Nous allons montrer la solution la plus simple qu'il est possible de mettre en œuvre réaliser l'application. Cette solution est celle qui nécessite le moins de modification mais elle comporte quelques défauts. Nous allons exposer cette solution et nous discuterons ensuite du moyen de la rendre plus correcte.

Pout notre solution simple, on a réalisé les modifications suivantes :

```
   @Requires(filter = "(factory.name=FibonacciFactory)")
   Factory fibFactory;

...
  if(moins1==null){
    try {
      InstanceManager f=InstanceManager)fibFactory.createComponentInstance(null);
      moins1=(Fibonacci)f.getPojoObject();
     } catch (Exception e) {
         e.printStackTrace();
     }
   }
...

  if(moins2==null){
   try {
    InstanceManager f=(InstanceManager)fibFactory.createComponentInstance(null);
     moins2=(Fibonacci)f.getPojoObject();
     } catch (Exception e) {
         e.printStackTrace();
     }
   }
...
```

Pour pouvoir créer des instances à la volée dans iPOJO, on récupère la fabrique du composant Fibonacci, puis pour chaque dépendance, on crée l'instance satisfaisante. Notons que cette solution requiert une expertise des mécanismes internes d'iPOJO. La description du composant iPOJO est la manière suivante :

```
<ipojo
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="org.apache.felix.ipojo"
    xsi:schemaLocation="org.apache.felix.ipojo
                http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd">

  <component name="FibonacciFactory"
      classname="fr.imag.adele.apam.pax.test.performance.FibonacciRecursive">
    <provides />
    <requires field="moins1" optional="true"/>
    <requires field="moins2" optional="true"/>
  </component>

  <instance component="FibonacciFactory" />


</ipojo>
```

Les instances iPOJO décrivent un état qui est lié à la résolution de leurs dépendances. Si une dépendance n'est pas résolue l'instance du composant ne démarre pas. Nous avons donc déclaré ces dépendances optionnelles (ce qui n'est pas le cas en réalité) pour forcer l'instance de composant à démarrer même s'il lui manque des dépendances, qui peuvent alors être créées à la volée.

Malgré notre effort, cette implémentation n'est pas correcte. En effet, si les instances sont créées à la volé à l'aide de l'API de iPOJO, ce n'est pas lui qui réalise la connexion entre les instances. L'expression « *moins1=(Fibonacci)f.getPojoObject();* » injecte directement l'instance dans le champ approprié. Il ne nous est pas été possible laisser iPOJO connecter les instances car ces instances ne devaient pas être partagées.

Par défaut, en cas de suppression d'une instance, iPOJO réalise une nouvelle connexion avec une autre instance disponible (créée au préalable), mais si aucune instance n'est disponible, iPOJO invalide les instances dont les dépendances n'ont pas été résolues. Comme notre application est récursive, ceci aura pour conséquence d'invalider toutes les instances de l'application ce qui provoque l'arrêt de cette dernière.

Pour réaliser l'application correctement, il aurait fallu faire d'autre modifications mais cette fois au niveau d'iPOJO lui-même. Celui-ci permet d'étendre le mécanisme de création d'instances en implémentant la classe « *org.apache.felix.ipojo.handlers.providedservice.CreationStrategy* ». Cependant cette solution nécessite une expertise plus importante dans la technologie. Son développement est beaucoup de plus complexe pour un simple développeur.

3.2.3. SCA (TUSCANY)

Pour SCA nous avons fait le choix d'utiliser l'implémentation Tuscany 2.0. Ce choix de l'implémentation n'a aucun n'impact sur la description de l'application car elle est la même quelle que soit l'implémentation SCA choisie.

Nous proposons la description suivante pour l'application Fibonacci en utilisant SCA :

```
<composite  xmlns=http://docs.oasis-open.org/ns/opencsa/sca/200912
            xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.1"
            targetNamespace="http://sample"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://docs.oasis-open.org/ns/opencsa/sca/200912
             http://docs.oasis-open.org/opencsa/sca-assembly/sca-1.1-cd06.xsd"
           name="fibonacci.composite">

  <component name="FibonacciComponent" >
    <implementation.java
         class="fr.imag.adele.apam.pax.test.performance.FibonacciRecursive" />
       <service name="Fibonacci">
         <interface.java
             interface="fr.imag.adele.apam.pax.test.performance.Fibonacci" />
       </service>
       <reference name="moins1" />
       <reference name="moins2" />
   </component>

    <wire source="FibonacciComponent/moins1"
          target="FibonacciComponent/Fibonacci" />
    <wire source="FibonacciComponent/moins2"
          target="FibonacciComponent/Fibonacci" />
</composite>
```

Pour créer notre Fibonacci il faut générer une description complète de tout l'arbre de Fibonacci dans le fichier XML, ce qui implique d'écrire des centaines ou miliers de lignes de description. Par conséquent, l'architecture est statique, elle ne pourra pas se rétablir face à un comportement dynamique du contexte.

Un palliatif, quand l'implémentation est statiquement connues et que les instances sont sans état (stateless) consiste à annoter la classe par *@Scope("STATELESS")* ce qui provoque la création d'une nouvelle instance à chaque appel, et donc une instanciation à la volée. Ceci a pour avantage de simuler un comportement dynamique, dans ce cas bien précis ; dans le cas de suppression d'instances, la plate-forme recrée automatiquement les instances manquantes. Mais il n'est pas possible de garder la même instance d'un appel à un autre.

3.2.4. BLUEPRINT (ECLIPSE VIRGO[19])

Cette spécification a pour but de combiner les caractéristiques du canevas Spring[Spri00] avec celles de la plate-forme OSGi™. Ainsi, d'un côté Spring profite des capacités de déploiement et de modularité fournie par OSGi™, de l'autre côté, Spring fournit un modèle à composants très connu utilisable au-dessus d'OSGi™. Afin de supporter le dynamisme, la notion de service a été rajouté au canevas Spring. Un *Bean* (c'est-à-dire un composant Spring) peut fournir et requérir des services OSGi™.

[19] http://www.eclipse.org/virgo/

Nous avons décrit notre application de la façon suivante :

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
         <bean id="Fibonacci"
                  class="fr.imag.adele.apam.pax.test.performance.FibonacciRecursive">
                  <property name="moins1" ref="FibonacciRef" />
                  <property name="moins2" ref="FibonacciRef" />
         </bean>

         <service ref="Fibonacci"
                  interface="fr.imag.adele.apam.pax.test.performance.Fibonacci" />

         <reference id="FibonacciRef"
                  interface="fr.imag.adele.apam.pax.test.performance.Fibonacci"
                  availability="optional" />
</blueprint>
```

Malheureusement, il nous a été impossible d'implémenter un algorithme capable de construite notre arbre d'instance dynamiquement. Seule une instance de service a été créée. Il faut décrire statiquement l'ensemble des 'instances statiquement. De plus, les composants ne communiquent pas avec des services s'ils sont dans un même contexte d'application (c'est-à-dire *Bundle*), mais utilisent les mécanismes fournis par Spring. Le modèle de composition proposé est très limité, il ne permet pas la substitution de fournisseur de service (dans une composition) et n'est pas reconfigurable.

CHAPITRE 8
CONCLUSION ET PERSPECTIVES

1. SYNTHÈSE

Les environnements ubiquitaires sont par nature imprévisibles. Notre approche a montré comment définir une application ubiquitaire par une architecture de référence. Celle-ci nous permet de décrire aussi bien des architectures statiques que des architectures flexibles et opportunistes. La description d'application proposée par ApAM apporte plus d'expressivité par rapport aux autres modèles de composants dans le domaine des applications ubiquitaires.

Notre approche est une première étape pour conférer aux applications ubiquitaires une résilience vis-à-vis du contexte d'exécution. La résilience est la capacité d'une application à continuer à fournir ses services malgré les changements imprévisibles du contexte. Nous avons démontré que nous obtenons un certain niveau de résilience par le contrôle de la conformité entre chaque architecture concrète et l'architecture de référence.

Notre proposition nous a permis de satisfaire aux exigences que nous avons identifiées dans notre état de l'art. La plate-forme ApAM offre l'abstraction nécessaire pour assurer l'évolution et la résilience vis-à-vis du contexte d'exécution pour les applications ubiquitaires.

1.1. ABSTRACTION

Notre modèle à composants offre trois niveaux d'abstraction : Spécification, Implémentation et Instance. Ceci nous permet de décrire l'application avec des buts de plus haut niveau sans se soucier de l'implémentation et de la technologie. En utilisant le plus haut niveau d'abstraction (les spécifications), on est capable de définir des applications en intension. La combinaison des niveaux d'abstraction nous permet de créer des applications hybrides (à la fois implicites et explicites). De plus, cette abstraction nous permet d'avoir des applications flexibles, pouvant faire face à un environnement dynamique.

Notre plate-forme d'exécution se base sur cette description pour construire l'application pas à pas. La construction de l'application est également guidée par les contraintes de dépendance exprimées sur l'application et ses composants. Les contraintes sont exprimées en utilisant un langage proche de LDAP, étendu pour pouvoir faire référence (directement ou par navigation) aux propriétés du contexte. Les contraintes deviennent alors sensibles au contexte d'exécution et l'application peut adapter ses besoins selon les ressources disponibles.

La définition d'une application ApAM est validée statiquement. Les propriétés, les contraintes et les préférences contextuelles d'une application sont vérifiées statiquement afin d'assurer la validité de la définition d'une application. Dès la phase de compilation, le développeur est averti des anomalies qui ont été détectées dans sa description de l'application. Le risque d'exécuter une application incohérente est alors réduit. Cette vérification statique ne considère pas les propriétés liées aux contextes car celles-ci sont différentes d'une exécution à une autre.

L'environnement d'exécutions étant partagé par plusieurs applications, notre approche vise à garantir leur protection. L'encapsulation proposée dans ApAM permet aux applications de spécifier les règles de partage et de visibilité pour les entités qui les composent (implémentations et instances). Cette encapsulation est réalisée par les composites, eux aussi décrits à plusieurs niveaux d'abstractions. L'abstraction du composite nous permet de réduire la complexité de la conception des applications en fournissant des représentations de haut niveau réutilisables et compréhensibles.

Enfin, pour masquer l'hétérogénéité entre les équipements et les services, cette même abstraction à trois niveaux est utilisée pour modéliser le contexte d'exécution. Nous avons alors obtenu un système homogène, sur lequel il est plus facile de raisonner et d'agir.

1.2. ÉVOLUTION

Pour pallier au dynamisme du contexte d'exécution, notre approche s'appuie sur le paradigme à service. Ainsi, nous utilisons les concepts de l'approche à service dans notre modèle à composants : nos applications sont faiblement couplées et robuste face au dynamisme du contexte d'exécutions.

Notre approche, utilise aussi le paradigme à service pour réaliser la composition incrémentale et dynamique des applications. Nos applications peuvent évoluer dynamiquement. À partir de l'architecture de référence, notre plate-forme d'exécution ApAM garantie la conformité de la composition d'une application. ApAM utilise les entités présentes dans la plate-forme pour répondre aux demandes de résolution. La connexion entre les composants, réalisée suite à une demande de résolution, prend en considération les propriétés dynamiques des composants avant et après la réalisation des connexions. De plus, les composants ApAM (quel que soit leur niveau d'abstraction) ainsi que les connexions sont dynamiques, ils peuvent apparaitre et disparaitre à tout moment. La construction incrémentale de l'application prend en considération la disponibilité des composants, et peut installer, instancier des composants, et créer et de détruire les connexions entre composants.

1.3. RÉSILIENCE

La résilience des applications ubiquitaires est d'abord rendue possible grâce aux propriétés d'abstraction et d'évolution dynamique : l'abstraction fournit la flexibilité nécessaire pour le choix des composants et l'évolution dynamique permet la prise en compte du contexte d'exécution dans la sélection des composants de l'application.

Afin d'augmenter cette résilience nous avons introduit le concept d'espace de résolution. Celui-ci permet d'intégrer différentes plates-formes hétérogènes dans un même contexte d'exécution. Chaque plate-forme peut participer à des demandes de résolution de dépendance et fournir les composants nécessaires au bon fonctionnement de l'application. À travers ce concept, on a élargi le contexte d'exécutions des applications ubiquitaires pour leur permettre d'atteindre un grand nombre de composants qui peuvent potentiellement répondre à leur besoins. La liaison avec ces espaces est établie grâce à des managers. Nous avons d'ailleurs réalisé différents managers génériques tels qu'*OBRMAN* qui nous permet de profiter des dépôts de code de type OBR ou *DISTRIMAN* qui nous permet d'atteindre des services distants.

Malgré tout, il peut exister des cas pour lesquels on n'arrive pas à trouver le composant souhaité. Pour éviter, dans ce cas, l'arrêt de l'application, nous avons introduit des stratégies d'échec. Ces stratégies peuvent être placées dans la description de l'application ou du composant. Elles définissent le comportement à adopter quand une demande de résolution de dépendance échoue. L'application des stratégies permet de mettre l'application en attente, de renvoyer une exception localisée ou encore de revenir sur les choix des résolutions précédentes. Ces stratégies sont appliquées d'une manière locale, elles n'interrompent pas la totalité de l'application qui continue à fonctionner dans un mode dégradé.

Comme le résume le tableau suivant, ApAM relève les défis que nous nous sommes fixés dans le cadre de cette thèse.

	Abstraction		**Évolution**		**Résilience**	
	Niveaux d'abstraction	**Contraintes**	**Composants**	**Connexions**	**Emplacements**	**Stratégies d'échec**
ApAM	●	●	●	●	●	●

Tableau 10 : ApAM est les défis de l'ubiquitaire

2. PERSPECTIVES

Nous pensons qu'ApAM n'est qu'un premier pas vers une infrastructure complète pour le développement et l'exécution des applications ubiquitaires. Pour atteindre cet objectif, de nombreuses perspectives sont en cours d'études. Nous en présentons quelques une.

2.1. ALGORITHME DE RÉSOLUTION

L'algorithme de résolution que nous avons implémenté dans ApAM permet de sélectionner les composants pour satisfaire une dépendance. Cet algorithme se base sur l'architecture de référence de l'application, il vieille à ce que le composant choisit satisfasse les contraintes de la description de l'application. Ce processus garantie la conformité entre l'architecture de référence qui décrit l'application, et l'architecture concrète qui représente l'exécution de l'application à un moment donné.

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. Ainsi, nous souhaitons que la concrétisation de l'architecture de référence, nous produise la « meilleure » composition, la plus adaptée au contexte d'exécution courant et la moins encline à échouer.

2.2. EXTENSION DES STRATÉGIES D'ÉCHEC

Pour améliorer la résilience des applications ubiquitaires vis-à-vis du contexte d'exécution, nous avons introduit quelques stratégies de résilience. Ces stratégies permettent à l'application d'adopter un mode d'exécution dégradé et d'éviter l'arrêt complet de l'application. L'émission d'exception, la mise en attente, et le retour arrière sont des stratégies génériques mises à disposition du développeur d'application. Cependant, nous pensons que d'autres stratégies peuvent être implémentées. Par exemple, le domaine métier d'une application peut exprimer des nouveaux comportements à adopter pour réagir face à l'échec de l'application.

Cette extension permettra prendre en considération des nouvelles stratégies pour faire face à l'échec de la résolution à l'exécution.

2.3. MEILLEUR GESTION DE LA CONFIGURATION

La plate-forme ApAM est capable d'instancier les composants lors d'une demande de résolution. Dans le cas où la plate-forme ne trouve aucune instance disponible pour réponde à la demande, le mécanisme de résolution utilise les composants de type implémentation pour créer une instance satisfaisante. Cette instance est créée en utilisant les contraintes et les préférences de résolution comme paramètres pour la création d'instances. Cette solution est pratique pour des configurations simples. Nous souhaitons cependant l'améliorer pour prendre en considération des cas plus complexe d'instanciation.

Cette perspective suppose alors que des configurations prédéfinies sont stockées dans des dépôts. La plate-forme est alors capable de les interroger à chaque instanciation. Une autre solution consiste à utiliser des managers spécialisés dans l'instanciation. Ils pourront intervenir à cette étape et décrire l'ensemble des propriétés nécessaires à l'instanciation. Cette solution à l'avantage d'être extensible. Nous pourrons alors définir autant de manager d'instanciation que l'on souhaite. Cependant, l'enjeu est de fournir une méthode simple pour réaliser ces managers.

2.4. ENVIRONNEMENT DE DÉVELOPPEMENT ET D'EXÉCUTION OUTILLÉS

Les environnements de développement jouent un rôle important pour guider les développeurs d'application. Ils définissent le processus de réalisation de l'application et offrent un accès simple aux différentes librairies nécessaires pour développement des applications ubiquitaires. De plus, s'ils sont associés à des environnements de tests et de simulation, ils permettent aux développeurs de s'assurer

du fonctionnement de son application dans un contexte d'exécution prédéfini. Grâce à cet ensemble d'outils on peut fiabiliser la procédure de production des applications ubiquitaires.

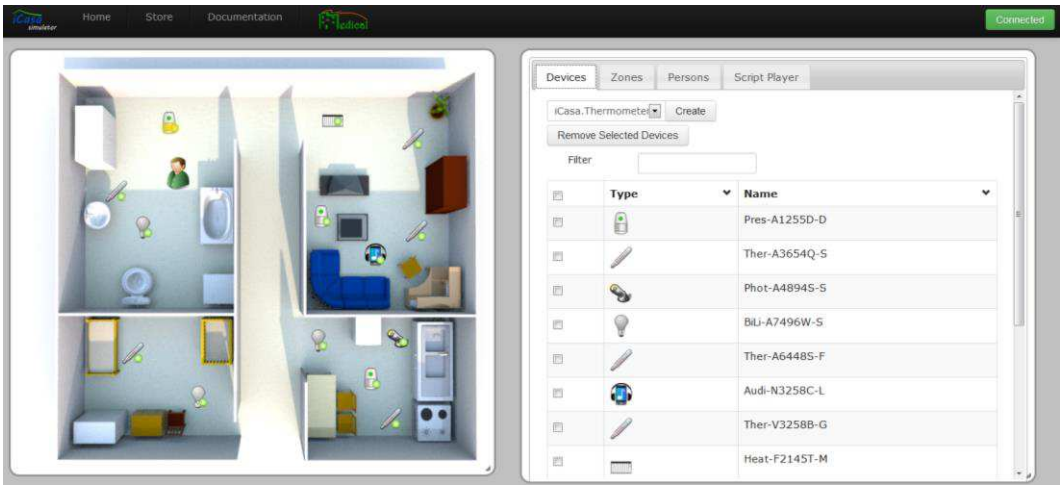


Figure 51 : Environnement de simulation iCASA

L'importance de cette perspective réside dans la capacité de l'environnement à faciliter le travail de conception et de développement de ce genre d'applications. Il permettra de mieux comprendre l'évolution de l'application et automatisera les tâches récurrentes de développement.

Les applications ubiquitaires sont réalisées pour interagir avec un ensemble d'équipements. Cependant, il est couteux d'investir dans tout un ensemble d'équipements domotiques pour réaliser les tests des applications. C'est pour cela que l'on envisage d'intégrer l'environnement de simulation iCASA (voir la Figure 51) dans cet environnement complet de développement.

Enfin, pour faciliter le travail d'un administrateur de plate-forme, nous envisageons de réaliser un outil de visualisation d'architecture. Cet outil aura la capacité de montrer en temps réel l'exécution courante de toutes les applications disponibles dans une plate-forme ApAM et ainsi d'identifier plus facilement les pannes. D'autre part, il sera possible d'agir directement sur la visualisation pour rajouter, supprimer ou modifier des composants, ou encore les déplacer d'une application à une autre.

2.5. VERS DES APPLICATIONS AUTONOMIQUES

Le concept d'application autonomique a été proposé à l'origine par IBM[Horn01]. Une application autonomique est un système informatique capable de s'adapter à des évolutions internes et externes avec un minimum d'interventions humaines. Cette idée est née du fait que la complexité des applications ne cessant de croitre, il devient de plus en plus difficile de diagnostiquer les causes d'erreurs par un administrateur humain. Il existe plusieurs travaux visant à réaliser ce genre d'applications, à titre d'exemple on peut citer le projet Autonomia [DHXC03], ou encore le projet Rainbow[HSCG04].

La perspective à long terme d'ApAM est d'offrir la capacité autonomique aux applications ubiquitaires. Pour atteindre cet objectif il faut une infrastructure permettant la réalisation et le support de gestionnaires autonomiques. ApAM fournit les mécanismes d'introspection et d'interrogation nécessaires. Notre objectifs est de réaliser des gestionnaires autonomiques de haut niveau, qui en s'appuyant sur ApAM et ses manager, va être capable de superviser l'exécution de l'application, puis qui déterminera les adaptations appropriées pour maintenir les applications en exécution.

ANNEXE 1
SPÉCIFICATION D'UN GESTIONNAIRE DE DÉPENDANCES

```
public interface DependencyManager extends Manager {

/**
* Provided that a dependency resolution is required by client(source), each manager
* is asked if it want to be involved.
* @param source the source asking for a resolution
* @param dependency the dependency to resolve. It contains the target type and
* name and the constraints.
* @param selPath the managers currently involved in this resolution.
*/
  public void getSelectionPath(Component source, Dependency dependency,  List<DependencyManager>
selPath);

/**
* Performs a complete resolution of the dependency.
*  The manager is asked to find the "right" implementations and instances for the provided
dependency.
*  If dependency is simple (not multiple), returns a singleton, and a single element in insts.
* @param source the instance asking for the resolution (and where to create
*           implementation, if needed). Cannot be null.
* @param dependency a dependency declaration containing the type and name of the dependency target.
* @return a resolved object contains all valid implementations and instances
*/
   public Resolved resolveDependency(Component source, Dependency dependency);

/**
* Once the resolution terminated, either successful or not, the managers
* are notified of the current selection.
```

```
* @param client the client (or source) of that resolution
* @param resName either the interfaceName, the spec name or the implementation name to resolve
*      depending on the fact newWireSpec or newWireImpl has been called.
* @param depName the dependency to resolve.
* @param impl the implementation selected
* @param insts the set of instances selected
*/
   public void notifySelection(Component client, ResolvableReference resName, String depName,
Implementation impl, Set<Instance> insts);
}
```

ANNEXE 2
SPÉCIFICATION D'UN GESTIONNAIRE DE DYNAMISME

```
public interface DynamicManager extends Manager {

	/**
	 * The manager asks to be notified of the creation of an instance or implem
	 * (or un-hidden)
	 *
	 */
	public abstract void addedComponent(Component newComponent);

	/**
	 * The manager asks to be notified of the removing of a an instance or
	 * implem (or hidden)
	 *
	 */
	public abstract void removedComponent(Component lostComponent);

	/**
	 * The manager asks to be notified of the removing of a an wire
	 *
	 */
	public abstract void removedWire(Wire wire);

	/**
	 * The manager asks to be notified of the creation of a wire
	 *
	 */
	public abstract void addedWire(Wire wire);
}
```

ANNEXE 3
SPÉCIFICATION D'UN GESTIONNAIRE DE PROPRIÉTÉS

```
public interface PropertyManager extends Manager {
          /**
           * The attribute "attr" has been modified.
           * @param component The component (Spec, Implem, Instance) holding that attribute.
           * @param attr The attribute name.
           * @param newValue The new value of that attribute.
           * @param oldValue The previous value of that attribute.
           */
          public void attributeChanged(Component component, String attr,
                                String newValue, String oldValue);

          /**
           * The attribute "attr" has been removed.
           * @param component The component (Spec, Implem, Instance) holding that attribute.
           * @param attr The attribute name.
           * @param oldValue The previous value of that attribute.
           */
          public void attributeRemoved(Component component, String attr,
                                String oldValue);

          /**
           * The attribute "attr" has been added (instantiated for the first time).
           * @param component The component (Spec, Implem, Instance) holding that  attribute.
           * @param attr The attribute name.
           * @param newValue The new value of that attribute.
           */
          public void attributeAdded(Component component, String attr, String newValue);
}
```

RÉFÉRENCES

[AAHL97] ABOWD, GREGORY D ; ATKESON, CHRISTOPHER G ; HONG, JASON ; LONG, SUE ; KOOPER, ROB ; PINKERTON, MIKE: Cyberguide: A mobile context-aware tour guide. In: *Wireless networks*. vol. 3 : Kluwer Academic Publishers, 1997, pp. 421–433

[AlCN02] ALDRICH, JONATHAN ; CHAMBERS, CRAIG ; NOTKIN, DAVID: ArchJava: connecting software architecture to implementation. In: *Proceedings of the 24rd International Conference on Software Engineering. ICSE* (2002), pp. 187–197

[AlCN06] ALDRICH, JONATHAN ; CHAMBERS, CRAIG ; NOTKIN, DAVID: Architectural reasoning in ArchJava. In: *ECOOP 2002—Object-Oriented Programming* : Springer, 2006, pp. 334–367

[AlDG98] ALLEN, ROBERT ; DOUENCE, RÉMI ; GARLAN, DAVID: Specifying and analyzing dynamic software architectures. In: *Fundamental Approaches to Software Engineering* : Springer, 1998, pp. 21–37

[AlLo03] ALEXANDRESCU, ANDREI ; LORINCZ, KONRAD: ArchJava: An Evaluation. In: *Rapport technique, University of Washington* (2003)

[ALRL04] AVIZIENIS, ALGIRDAS ; LAPRIE, J-C ; RANDELL, BRIAN ; LANDWEHR, CARL: Basic concepts and taxonomy of dependable and secure computing. In: *IEEE Transactions on Dependable and Secure Computing* vol. 1 (2004), Nr. 1, pp. 11–33

[Apac00a] APACHE SOFTWARE FOUNDATION: *Apache Felix Documentation*. URL http://felix.apache.org/documentation.html

[Apac00b] APACHE SOFTWARE FOUNDATION: *Apache Tuscany*. URL http://tuscany.apache.org/

[Apac00c] APACHE SOFTWARE FOUNDATION: *Apache Felix : Apache Felix OSGi Bundle Repository*. URL http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

[Arch00] ARCHJAVA: *ArchJava Reference Manual 1.0*. URL http://archjava.fluid.cs.cmu.edu/papers/archjava-language.pdf

[Arsa04] ARSANJANI, ALI: Service-oriented modeling and architecture. In: *IBM developer works* (2004), Nr. January, pp. 1–15

[ASCN02] ALDRICH, JONATHAN ; SAZAWAL, VIBHA ; CHAMBERS, CRAIG ; NOTKIN, DAVID: Architecture-centric programming for adaptive systems. In: *Proceedings of the first workshop on Self-healing systems* : ACM, 2002 — ISBN 1581136099, pp. 93–95

[AtKü01] ATKINSON, COLIN ; KÜHNE, THOMAS: The essence of multilevel metamodeling. In: *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools* : Springer, 2001, pp. 19–33

[BaCK12] BASS, LEN ; CLEMENTS, PAUL ; KAZMAN, RICK: *Software Architecture in Practice*. 3. ed. : Addison-Wesley, 2012 — ISBN 013294278X, 9780132942782

[BaGh10] BARESI, LUCIANO ; GHEZZI, CARLO: The disappearing boundary between development-time and run-time. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research* : ACM, 2010 — ISBN 9781605587912, pp. 17–22

[BBBC00] BACHMANN, FELIX ; BASS, LEN ; BUHMAN, CHARLES ; COMELLA-DORDA, SANTIAGO ; LONG, FRED ; ROBERT, JOHN ; SEACORD, ROBERT ; WALLNAU, KURT: *Volume II: Technical concepts of component-based software engineering*. vol. II : Carnegie Mellon University, Software Engineering Institute, 2000

[BBCH07] BEISIEGEL, MICHAEL ; BOOZ, DAVE ; COLYER, ADRIAN ; HILDEBRAND, HAL ; MARINO, JIM ; TAM, KEN: *SCA Service Component Architecture*. URL https://www.oasis-open.org/committees/download.php/25348/SCA_SpringComponentImplementationSpecification-V100.pdf

[BCLQ06] BRUNETON, ERIC ; COUPAYE, THIERRY ; LECLERCQ, MATTHIEU ; QUÉMA, VIVIEN ; STEFANI, JEAN-BERNARD: The fractal component model and its support in java. In: *Software: Practice and Experience* vol. 36 (2006), pp. 1257–1284

[BDBM00] BARAIS, OLIVIER ; DARTOIS, JEAN-ÉMILE ; BOURCIER, JOHANN ; MORIN, BRICE ; NAIN, GRÉGORY ; PLOUZEAU, NOËL ; SUNYE, GERSON ; JÉZÉQUEL, JEAN-MARC: *Kevoree*. URL http://kevoree.org/

[BeBC12] BERTRAN, BENJAMIN ; BRUNEAU, JULIEN ; CASSOU, DAMIEN: DiaSuite: A tool suite to develop Sense/Compute/Control applications. In: *Science of Computer Programming* (2012), Nr. Fourth special issue on Experimental Software and Toolkits, pp. 1–28

[BHTV04] BARESI, LUCIANO ; HECKEL, REIKO ; THONE, SEBASTIAN ; VARRÓ, DÁNIEL: Style-based refinement of dynamic software architectures. In: *Proceedings Fourth Working IEEE/IFIP Conference Software Architecture. WICSA* : IEEE, 2004, pp. 155–164

[BMDL08] BARAIS, OLIVIER ; MEUR, ANNE FRANÇOISE LE ; DUCHIEN, LAURENCE ; LAWALL, JULIA: Software architecture evolution. In: *Software Evolution* : Springer, 2008, pp. 233–262

[BPHT06] BOUCHENAK, SARA ; PALMA, NOEL DE ; HAGIMONT, DANIEL ; TATON, CHRISTOPHE: Autonomic management of clustered applications. In: *IEEE International Conference on Cluster Computing* : IEEE, 2006, pp. 1–11

[BPPT06] BUCCHIARONE, ANTONIO ; POLINI, ANDREA ; PELLICCIONE, PATRIZIO ; TIVOLI, MASSIMO: Towards an architectural approach for the dynamic and automatic composition of software components. In:

Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis : ACM, 2006, pp. 12–21

[Brad04] BRADBURY, JEREMY S.: Organizing definitions and formalisms for dynamic software architectures. In: *Techinical Report* vol. 477 (2004)

[Bure06] BUREŠ, TOMÁŠ: *Generating Connectors for Homogeneous and Heterogeneous Deployment*, CHARLES UNIVERSITY IN PRAGUE, 2006

[CaBC10] CASSOU, DAMIEN ; BRUNEAU, JULIEN ; CONSEL, CHARLES: A tool suite to prototype pervasive computing applications. In: *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, Ieee (2010), pp. 820–822 — ISBN 978-1-4244-6605-4

[Cass11] CASSOU, DAMIEN: *Développement logiciel orienté paradigme de conception : la programmation dirigee par la spécification*, Université de Bordeaux 1, 2011

[CBCL11] CASSOU, DAMIEN ; BALLAND, EMILIE ; CONSEL, CHARLES ; LAWALL, JULIA: Leveraging software architectures to guide and verify the development of sense/compute/control applications. In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, USA : ACM, 2011 — ISBN 9781450304450, pp. 431–440

[CBLC09] CASSOU, DAMIEN ; BERTRAN, BENJAMIN ; LORIANT, NICOLAS ; CONSEL, CHARLES: A generative programming approach to developing pervasive computing systems. In: *Proceedings of the eighth international conference on Generative programming and component engineering - GPCE '09*. New York, New York, USA, ACM Press (2009), p. 137 — ISBN 9781605584942

[CBME10] CASSOU, DAMIEN ; BRUNEAU, JULIEN ; MERCADAL, JULIEN ; ENARD, QUENTIN ; BALLAND, EMILIE ; LORIANT, NICOLAS ; CONSEL, CHARLES: Towards a tool-based development methodology for sense/compute/control applications. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*. New York, USA : ACM Press, 2010 — ISBN 9781450302401, p. 247

[CCGL07] COSTA, PAOLO ; COULSON, GEOFF ; GOLD, RICHARD ; LAD, MANISH ; MASCOLO, CECILIA ; MOTTOLA, LUCA ; PICCO, GIAN PIETRO ; SIVAHARAN, THIRUNAVUKKARASU ; ET AL.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: *Fifth Annual IEEE International Conference on Pervasive Computing and Communications, 2007. PerCom'07* : IEEE, 2007, pp. 69–78

[CeHa04] CERVANTES, HUMBERTO ; HALL, RICHARD S: Autonomous adaptation to dynamic availability using a service-oriented component model. In: *Proceedings of the 26th International Conference on Software Engineering* : IEEE Computer Society, 2004, pp. 614–623

[Cerv04] CERVANTES, HUMBERTO: *Vers un modèle à composants orienté services pour supporter la disponibilité dynamique*, Université Joseph Fourier - Grenoble I, 2004

[CFGJ03] CHEUNG, DANIEL ; FUCHET, JÉRÔME ; GRILLON, FLORENT ; JOULIÉ, GABRIEL ; TIGLI, J-Y: Wcomp: Rapid application development toolkit for wearable computer based on java. In: *Man and Cybernetics, 2003. IEEE International Conference on Systems* : IEEE, 2003 — ISBN 0780379527, pp. 4198–4203

[CGBS02] CLEMENTS, PAUL ; GARLAN, DAVID ; BASS, LEN ; STAFFORD, JUDITH: *Documenting software architectures: views and beyond* : Addison-Wesley Professional, 2002 — ISBN 0321552687

[CGSS02] CHENG, SHANG-WEN ; GARLAN, DAVID ; SCHMERL, BRADLEY ; SOUSA, JOÃO PEDRO ; SPITZNAGEL, BRIDGET ; STEENKISTE, PETER ; HU, NINGNING: Software architecture-based adaptation for pervasive systems. In: *Trends in Network and Pervasive Computing—ARCS 2002* : Springer, 2002, pp. 67–82

[Char05] CHARPY, GEORGE ; LA SOCIÉTÉ (ed.): *Mémoires et compte-rendu des travaux de la Société des ingénieurs civils*, 1905

[ChFJ03] CHEN, HARRY ; FININ, TIM ; JOSHI, ANUPAM: An intelligent broker for context-aware systems. In: *Adjunct Proceedings of Ubicomp*. vol. 2003 : Citeseer, 2003, pp. 183–184

[ChKo00] CHEN, GUANLING ; KOTZ, DAVID: A survey of context-aware mobile computing research. In: *Dartmouth College, Hanover, NH* vol. 3755, Citeseer (2000), pp. 1–16

[CLLC05] CHEN, CHIH-MING ; LEE, YUH-RUEY ; LIN, CHIA-WEN ; CHEN, YUNG-CHANG: Error resilience transcoding using prioritized intra-refresh for video multicast over wireless networks. In: *IEEE International Conference on Multimedia and Expo, 2005. ICME* : IEEE, 2005 — ISBN 0780393325, pp. 1310–1313

[Cyru99] CYRULNIK, BORIS: *Un merveilleux malheur* : Odile Jacob, 1999 — ISBN 2738111254

[DaLe05] DAVID, PIERRE-CHARLES ; LEDOUX, THOMAS: WildCAT: a generic framework for context-aware applications. In: *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing* : ACM, 2005 — ISBN 1595932682, pp. 1–7

[Daml00] DAMLS: *DAML-S: Semantic Markup For Web Services*. URL http://www.daml.org/services/daml-s/2001/05/daml-s.html. - abgerufen am 2013-07-08

[DaPZ02] DAHCHOUR, MOHAMED ; PIROTTE, ALAIN ; ZIMÁNYI, ESTEBAN: Materialization and its metaclass implementation. In: *IEEE Transactions on Knowledge and Data Engineering* vol. 14 (2002), Nr. 5, pp. 1078–1094

[DeCH12] DEY, ANIND K. ; CHU, HAO-HUA ; HAYES, GILLIAN: Ubicomp '12 The 2012 ACM Conference on Ubiquitous Computing. In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing* : ACM New York, USA, 2012 — ISBN 978-1-4503-1224-0, p. 1268

[Dey01] DEY, ANIND K: Understanding and Using Context. In: *Personal and Ubiquitous Computing* vol. 5, Springer-Verlag (2001), Nr. 1, pp. 4–7

[DHXC03] DONG, X. ; HARIRI, S. ; XUE, L. ; CHEN, H. ; ZHANG, M. ; PAVULURI, S. ; RAO, S.: Autonomia: an autonomic computing environment. In: *Proc. of the 2003 IEEE International Performance, Computing, and Communication Conference*, 2003, pp. 61–68

[DoCa01] DOWLING, JIM ; CAHILL, VINNY: The k-component architecture meta-model for self-adaptive software. In: *Metalevel Architectures and Separation of Crosscutting Concerns*. vol. 3 : Springer, 2001, pp. 81–88

[DoCC01] DOWLING, JIM ; CAHILL, VINNY ; CLARKE, SIOBHÀN: Dynamic software evolution and the k-component model. In: *Workshop on Software Evolution, OOPSLA* vol. 2001 (2001)

[EAAC04] ENDREI, MARK ; ANG, JENNY ; ARSANJANI, AL ; CHUA, SOOK ; COMTE, PHILIPPE ; KROGDAHL, PÀL ; LUO, MIN ; NEWLING, TONY: *Patterns: service-oriented architecture and web services*. First Edit. ed. : IBM Corporation, International Technical Support Organization, 2004

[Ecli00] ECLIPSEPEDIA: *Equinox p2*. URL http://wiki.eclipse.org/P2

[EjSB07] EJIGU, DEJENE ; SCUTURICI, MARIAN ; BRUNIE, LIONEL: CoCA: A Collaborative Context-Aware Service Platform for Pervasive Computing. In: *Fourth International Conference on Information Technology (ITNG'07)*, Ieee (2007), pp. 297–302 — ISBN 0-7695-2776-0

[Esco08] ESCOFFIER, CLEMENT: *iPOJO: Un modèle à composant à service flexible pour les systèmes dynamiques*, Université JOSEPH FOURIER, 2008

[EsHa07] ESCOFFIER, CLEMENT ; HALL, RICHARD S.: Dynamically adaptable applications with iPOJO service components. In: *Software Composition* : Springer, 2007, pp. 113–128

[EsHL07] ESCOFFIER, CLÉMENT. ; HALL, RICHARD S ; LALANDA, PHILIPPE: iPOJO: An extensible service-oriented component framework. In: *IEEE International Conference on Services Computing. SCC 2007* : IEEE, 2007, pp. 474–481

[Fabr76] FABRY, ROBERT S.: How to design a system in which modules can be changed on the fly. In: *Proceedings of the 2nd international conference on Software engineering* : IEEE Computer Society Press, 1976. — From Duplicate 2 (How to design a system in which modules can be changed on the fly - Fabry, RS) , pp. 470–476

[FDPB12a] FOUQUET, FRANÇOIS ; DAUBERT, ERWAN ; PLOUZEAU, NOËL ; BARAIS, OLIVIER ; BOURCIER, JOHANN ; JÉZÉQUEL, JEAN-MARC: Dissemination of reconfiguration policies on mesh networks. In: *Distributed Applications and Interoperable Systems* : Springer, 2012

[FDPB12b] FOUQUET, FRANÇOIS ; DAUBERT, ERWAN ; PLOUZEAU, NOEL ; BARAIS, OLIVIER: Kevoree: une approche model@ runtime pour les systèmes ubiquitaires. In: *UbiMob2012* vol. 2012 (2012)

[FHSE06] FLOCH, JAQUELINE JACQUELINE ; HALLSTEINSEN, SVEIN ; STAV, ERLEND ERLENV ; ELIASSEN, FRANK ; LUND, KETIL ; GJORVEN, ELI: Using architecture models for runtime adaptability. In: *IEEE Software* vol. 23 (2006), Nr. 2, pp. 62–70

[Flor11] DE FLORIO, VINCENZO: Robust-and-evolvable resilient software systems: open problems and lessons learned. In: *Proceedings of the 8th workshop on Assurances for self-adaptive systems* : ACM, 2011 — ISBN 9781450308533, pp. 10–17

[FMFB12] FOUQUET, FRANCOIS ; MORIN, BRICE ; FLEUREY, FRANCK ; BARAIS, OLIVIER ; PLOUZEAU, NOEL ; JEZEQUEL, JEAN-MARC: A dynamic component model for cyber physical systems. In: *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering* : ACM, 2012 — ISBN 9781450313452, pp. 135–144

[Fowl00] FOWLER, MARTIN: *POJO : An acronym for: Plain Old Java Object.* URL http://www.martinfowler.com/bliki/POJO.html

[FoZa94] FORMAN, G.H. ; ZAHORJAN, J.: The challenges of mobile computing. In: *Computer*, Published by the IEEE Computer Society (1994), pp. 38–47

[FSLM02] FASSINO, JEAN-PHILIPPE ; STEFANI, JEAN-BERNARD ; LAWALL, JULIA L. ; MULLER, GILLES: Think: A Software Framework for Component-based Operating System Kernels. In: *USENIX Annual Technical Conference, General Track* (2002), pp. 73–86

[GaKu03] GADDAH, ABDULBASET ; KUNZ, THOMAS: A survey of middleware paradigms for mobile computing. In: *Techical Report, July*, Citeseer (2003), Nr. July

[Garz12] GARZA, ISSAC NOÉ GARCÍA: *Modèles de conception et d'exécution pour la médiation et l'intégration de services*, 2012

[GaSc09] GARLAN, DAVID ; SCHMERL, BRADLEY: Ævol: A tool for defining and planning architecture evolution. In: *2009 IEEE 31st International Conference on Software Engineering*, Ieee (2009), pp. 591–594 — ISBN 978-1-4244-3453-4

[GaSh93] GARLAN, DAVID ; SHAW, MARY: An introduction to software architecture. In: *An Introduction to Software Architecture* (1993), Nr. January

[GaSS02] GARLAN, DAVID ; SIEWIOREK, DP DANIEL P ; STEENKISTE, PETER: Project aura: Toward distraction-free pervasive computing. In: *IEEE Pervasive computing*, Published by the IEEE Computer Society (2002), pp. 22–31

[Gham97] G. HAMILTON: Java Beans Specification v1.01. In: *Sun Microsystems, Techical Report* (1997)

[GoHe06] GONZALEZ-PEREZ, CESAR ; HENDERSON-SELLERS, BRIAN: A powertype-based metamodelling framework. In: *Software & Systems Modeling* vol. 5 (2006), Nr. 1, pp. 72–90

[GuPZ05] GU, T ; PUNG, H.K. ; ZHANG, D.Q.: A service-oriented middleware for building context-aware services. In: *Journal of Network and computer applications* vol. 28, Elsevier (2005), Nr. 1, pp. 1–18

[HeCo01] HEINEMAN, GEORGE T. ; COUNCILL, WILLIAM T.: *Component-based software engineering: putting the pieces together*. illustrate. ed. : Addison-Wesley, 2001, 2001 — ISBN 0201704854

[HéTL05] HÉRAULT, COLOMBE ; THOMAS, GAËL ; LALANDA, PHILIPPE: Mediation and enterprise service bus: A position paper. In: *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*, 2005, pp. 67–80

[Holl73] HOLLING, CS: Resilience and stability of ecological systems. In: *Annual review of ecology and systematics* (1973)

[HoPu93] HOFMEISTER, CHRISTINE ; PURTILO, JAMES: Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In: *Proceedings the 13th International Conference on Distributed Computing Systems*, 1993, pp. 101–110

[Horn01] HORN, PAUL: Autonomic computing: IBM's Perspective on the State of Information Technology. In: *Computing Systems* vol. 2007, IBM (2001), Nr. Jan, pp. 1–40

[HSCG04] HUANG, AN-CHENG ; STEENKISTE, PETER ; CHENG, SHANG-WEN ; GARLAN, DAVID ; SCHMERL, BRADLEY: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In: *Computer* vol. 37 (2004), Nr. 10, pp. 46–54

[HSPL03] HOFER, THOMAS ; SCHWINGER, WIELAND ; PICHLER, MARIO ; LEONHARTSBERGER, GERHARD ; ALTMANN, JOSEF ; RETSCHITZEGGER, WERNER ; HAGENBERG, A- ; KEPLER, JOHANNES: Context-awareness on mobile devices - the hydrogen approach. In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the* vol. 43, Ieee (2003), Nr. 7236, p. 10 pp. — ISBN 0-7695-1874-5

[Ibm00] IBM: *SCA in WebSphere Application Server: Overview*. URL http://pic.dhe.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.soafep.multiplatform.doc/info/ae/ae/csca_overview.html

[Jako11] JAKOB, HENNER: *Vers la sécurisation des systemes d'informatique ubiquitaire par le design: une approche langage*, 2011

[JeLe00] JEN, L ; LEE, Y: IEEE recommended practice for architectural description of software-intensive systems. In: *IEEE Architecture* (2000)

[JeWe03] JENRONIMO, M ; WEAST, JACK: UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play, Intel Press (2003), p. 481 — ISBN 0971786119

[JoWo96] JOHNSON, RALPH ; WOOLF, BOBBY: The Type Object Pattern (1996), pp. 1–13

[KiLM97] KICZALES, G ; LAMPING, J ; MENDHEKAR, A: *Aspect-oriented programming*, 1997

[KLTO12] KAY, J. ; LUKOWICZ, P. ; TOKUDA, H. ; OLIVIER, P. ; KRÜGER, A: 10th Internationnal Conference, Pervasive Computing 2012. In: , 2012

[Koch98] KOCH, RICHARD: *The 80/20 Principle: The Secret of Achieving More With Less*, 1998 — ISBN 0385491700

[KrMa90] KRAMER, J. ; MAGEE, J.: The evolving philosophers problem: dynamic change management. In: *IEEE Transactions on Software Engineering* vol. 16 (1990), Nr. 11, pp. 1293–1306

[Lapr08] LAPRIE, JEAN-CLAUDE: From dependability to resilience. In: *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks* (2008)

[LeCh11] LEE, SANG-GOO ; CHANG, JUNO: Survey and Trend Analysis of Context-Aware Systems. In: *Info Information-An International* (2011)

[LeQS04] LECLERCQ, MATTHIEU ; QUÉMA, VIVIEN ; STEFANI, JEAN-BERNARD: DREAM: a component framework for the construction of resource-aware, reconfigurable MOMs. In: *Proceedings of the 3rd workshop on Adaptive and reflective middleware*. New York, USA : ACM, 2004 — ISBN 1581139497, pp. 250–255

[LeSH03] LEE, JINYOUL ; SIAU, KENG ; HONG, SOONGOO: Enterprise integration with ERP and EAI. In: *Communications of the ACM* vol. 46, ACM (2003), Nr. 2, pp. 54–60

[Love00] LOVELOCK, JAMES: *Gaia: A New Look at Life on Earth* : Oxford University Press, USA, 2000 — ISBN 0192862189

[LYOA02] LYYTINEN, KALLE ; YOUNGJIN, YOO ; OF, COMMUNICATIONS ; ACM, T H E: Issues and Challenges in Ubiquitous Computing. In: *Communications of the ACM* vol. 45 (2002), Nr. 12, pp. 62–65

[MaDe05] MARIN, CRISTINA ; DESERTOT, MIKAEL: Sensor bean: a component platform for sensor-based services. In: *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, ACM (2005), pp. 1–8 — ISBN 1-59593-268-2

[MaDK94] MAGEE, J. ; DULAY, N. ; KRAMER, J.: Regis: A constructive development environment for distributed programs. In: *Distributed Systems Engineering*, Ieee (1994), pp. 4–14 — ISBN 0-8186-5390-6

[MaEK95] MAGEE, JEFF ; EISENBACH, SUSAN ; KRAMER, JEFF: Modelling darwin in the π-calculus. In: *Theory and Practice in Distributed Systems* : Springer, 1995, pp. 133–152

[MaHa99] MATENA, V ; HAPNER, M: Enterprise Java Beans Specification v1. 1-Final Release. In: *Sun Microsystems, May* (1999)

[MBJF09] MORIN, BRICE ; BARAIS, OLIVIER ; JÉZÉQUEL, JEAN-MARC ; FLEUREY, FRANCK ; SOLBERG, ARMOR: Models@ run. time to support dynamic adaptation. In: *IEEE Computer* vol. 42 (2009), Nr. 10, pp. 41–51

[McHa00] MCGUINNESS, DEBORAH L. ; HARMELEN, FRANK VAN: *OWL Web Ontology Language Overview*. URL http://www.w3.org/TR/owl-features/

[Mcil69] MCILROY, DOUG: Mass-produced Software Components (1969), pp. 138 – 155

[McRo98] MCCANN, P J ; ROMAN, G C: Compositional programming abstractions for mobile computing. In: *IEEE Transactions on Software Engineering* vol. 24 (1998), pp. 97–110

[Merc11] MERCADAL, JULIEN: *Approche langage au développement logiciel: application au domaine des systèmes d'informatique ubiquitaire*, Université de Bordeaux, 2011

[MeTa00] MEDVIDOVIC, NENAD ; TAYLOR, RICHARD N.: A classification and comparison framework for software architecture description languages. In: *IEEE Transactions on Software Engineering*. vol. 26 : IEEE, 2000, pp. 70–93

[Meye09] MEYER, JOHN F.: Defining and evaluating resilience: A performability perspective. In: *Proceedings of the International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS),*. vol. 1, 2009

[Mich03] MICHEL, HANS-ULRICH: *OSGi Technology in the Vehicle*. URL http://www.osgi.org/wiki/uploads/Congress2003/OSGiWorld_BMW_AutomotiveTrack_Day2.pdf

[NiDa95] NIERSTRASZ, OSCAR ; DAMI, LAURENT: Component-Oriented Software Technology. In: *Object-Oriented Software Composition* (1995), pp. 3–28

[Oasi00] OASIS OPEN SCA: *Service Component Architecture*. URL http://www.oasis-opencsa.org/

[Odel98] ODELL, JJ: *Advanced object-oriented analysis and design using UML* : SIGS, 1998 — ISBN 052164819X

[Orac00a] ORACLE: *Oracle*. URL http://www.oracle.com/index.html

[Orac00b] ORACLE (SUNMICROSYSTEMS): *Enterprise JavaBeans Technology.* URL http://www.oracle.com/technetwork/java/index-jsp-140203.html

[Orac00c] ORACLE: *GlassFish Server*. URL http://glassfish.java.net/

[Oran11] ORANGE LABS: *Open the Box! Challenges in the Smart Home*. URL http://www.iestcfa.org/presentations/etfa2011/keynote_Bottaro.pdf

[Osgi00a] OSGI ALLIANCE: *OSGi Specifications*. URL http://www.osgi.org/Specifications/HomePage

[Osgi00b] OSGI ALLIANCE: *The OSGi Architecture*. URL http://www.osgi.org/Technology/WhatIsOSGi

[Osgi05] OSGI ALLIANCE: *RFC-0112 Bundle Repository*. URL http://www.osgi.org/download/rfc-0112_BundleRepository.pdf

[Ow00a] OW2: *Cecilia framework*. URL http://fractal.ow2.org/cecilia-site/current/

[Ow00b] OW2: *JOnAS*. URL http://jonas.ow2.org/xwiki/bin/view/Main/

[PaHe07] PAPAZOGLOU, MP MIKE P. ; HEUVEL, WJ VAN DEN WILLEM-JAN: Service oriented architectures: approaches, technologies and research issues. In: *The VLDB Journal* vol. 16 (2007), Nr. 3, pp. 389–415

[Papa03a] PAPAZOGLOU, MP: Service-oriented computing: Concepts, characteristics and directions. In: *, 2003. WISE 2003. Proceedings of the* vol. p3. Washington, DC, USA, IEEE Computer Society (2003), p. 3 — ISBN 0-7695-1999-7

[Papa03b] PAPAZOGLOU, MIKE P.: *Service -Oriented Computing: Concepts, Characteristics and Directions*. URL http://www.item.ntnu.no/fag/ttm5/papers_2007/SOA-Concep-Char-Dir.pdf. - abgerufen am 2012-03-20

[Perc00] PERCOM: *IEEE International Conference on Pervasive Computing and Communications (PerCom 2014)*. URL http://www.percom.org/

[Perr96] PERRY, DE: System compositions and shared dependencies. In: *Software Configuration Management* (1996)

[Proj02] PROJET ACCORD: *Etat de l'art sur les Langages de Description d'Architecture (ADLs)*. URL http://projects.infres.enst.fr/accord/lot1/lot_1.1-2.pdf

[PTDL08] PAPAZOGLOU, MICHAEL P. ; TRAVERSO, PAOLO ; DUSTDAR, SCHAHRAM ; LEYMANN, FRANK: Service-oriented computing: a research roadmap. In: *International Journal of Cooperative Information Systems*. vol. 17 : World Scientific, 2008, pp. 223–255

[RHCR02] ROMÁN, MANUEL ; HESS, CHRISTOPHER ; CERQUEIRA, RENATO ; RANGANATHAN, ANAND ; CAMPBELL, R.H. ROY H. ; NAHRSTEDT, KLARA ; ROMAN, M. ; RANGANAT, ANAND: Gaia: A middleware infrastructure to enable active spaces. In: *IEEE Pervasive Computing* vol. 20, Citeseer (2002), Nr. 4, pp. 74–83

[RHTN10] ROMERO, DANIEL ; HERMOSILLO, GABRIEL ; TAHERKORDI, AMIRHOSEIN ; NZEKWA, RUSSEL ; ROUVOY, ROMAIN ; ELIASSEN, FRANK: RESTful integration of heterogeneous devices in pervasive environments. In: *Distributed Applications and Interoperable Systems* : Springer, 2010, pp. 1–14

[RMCM03] RANGANATHAN, ANAND ; MCGRATH, ROBERT E ; CAMPBELL, ROY H ; MICKUNAS, M DENNIS: Use of ontologies in a pervasive computing environment. In: *The Knowledge Engineering Review* vol. 18 (2003), Nr. 3, pp. 209–220

[SaDA99] SALBER, DANIEL ; DEY, A.K. ; ABOWD, G.D.: The context toolkit: aiding the development of context-enabled applications. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit* : ACM, 1999, pp. 434–441

[SaMB03] SAHA, DEBASHIS ; MUKHERJEE, AMITAVA ; BANDYOPADHYAY, S.: *Networking infrastructure for pervasive computing: enabling technologies & systems* : Kluwer Academic Publishers, 2003

[SaMu03] SAHA, DEBASHIS ; MUKHERJEE, AMITAVA: Pervasive computing: a paradigm for the 21st century. In: *Computer* vol. 36, IEEE (2003), Nr. 3, pp. 25–31

[Saty01] SATYANARAYANAN, M.: Pervasive computing: vision and challenges. In: *IEEE Personal Communications* vol. 8 (2001), Nr. 4, pp. 10–17

[ScTh94] SCHILIT, B.N. ; THEIMER, M.M.: Disseminating active map information to mobile hosts. In: *Network, IEEE* vol. 8, IEEE (1994), Nr. 5, pp. 22–32

[SeKo03] SENDALL, S. ; KOZACZYNSKI, W.: Model transformation: The heart and soul of model-driven software development. In: *Software, IEEE* vol. 20 (2003), Nr. 5, pp. 42–45

[Seli05] SELIC, BRAN: *Unified Modeling Language version 2.0*. URL http://www.ibm.com/developerworks/rational/library/05/321_uml

[SeMe12] SEINTURIER, LIONEL ; MERLE, PHILIPPE: A component-based middleware platform for reconfigurable service-oriented architectures. In: *Software: Practice and Experience* (2012), Nr. June 2011, pp. 559–583

[ShGa96] SHAW, MARY ; GARLAN, DAVID: *Software architecture: perspectives on an emerging discipline*. 2. ed. : Prentice Hall, 1996 — ISBN 0131829572

[Soft00] SOFTWARE ENGINEERING INSTITUTE: *Community Software Architecture Definitions*. URL http://www.sei.cmu.edu/architecture/start/glossary/community.cfm

[Sole00] SOLEY, R: Model driven architecture. In: *OMG white paper* (2000), pp. 1–12

[SPDC06] SEINTURIER, LIONEL ; PESSEMIER, NICOLAS ; DUCHIEN, LAURENCE ; COUPAYE, THIERRY: A component model engineered with components and aspects. In: *Component-Based Software Engineering* : Springer, 2006, pp. 139–153

[Spri00] SPRING: *Spring Framework*. URL http://www.springsource.org/spring-framework

[StLi04] STRANG, THOMAS ; LINNHOFF-POPIEN, CLAUDIA: A Context Modeling Survey. In: *workshop on advanced context modelling, reasoning and management, ubicomp the sixth international conference on ubiquitous computing, nottingham/england* (2004)

[StVo06] STAHL, THOMAS (TOM) ; VOELTER, M: *Model-driven software development*, 2006

[Sugr08] SUGRUE, JAMES: *Eclipse Plug-in Development*. URL http://refcardz.dzone.com/refcardz/eclipse-plug-development. — DZone Refcardz

[Szyp02] SZYPERSKI, CLEMENS ; DOMINIK W. GRUNTZ, S. M. (ed.): *Component software: beyond object-oriented programming*. 2. ed. : Pearson Education, 2002 — ISBN 0201745720

[TaMD09] TAYLOR, R.N. ; MEDVIDOVIC, NENAD ; DASHOFY, E.M.: *Software architecture: foundations, theory, and practice* : Wiley, 2009 — ISBN 0470167742

[TCLR06] TIGLI, JEAN-YVES ; CHEUNG-FOO-WO, DANIEL ; LAVIROTTE, STÉPHANIE ; RIVEILL, MICHEL: Adaptation au contexte par tissage d'aspects d'assemblage de composants déclenchés par des conditions contextuelles. In: *Ingénierie des systèmes d'information* vol. 11 (2006), Nr. 5, pp. 89–114

[Thel06] THELIN., J. SHLIMMER AND J.: *Devices Profile for Web Services*, 2006

[Tilm96] TILMAN, D: Biodiversity: population versus ecosystem stability. In: *Ecology* (1996)

[TLRH11] TIGLI, JEAN-YVES J.Y. ; LAVIROTTE, STÉPHANE ; REY, GAËTAN ; HOURDIN, VINCENT ; RIVEILL, MICHEL: Lightweight service oriented architecture for pervasive computing. In: *Arxiv preprint arXiv:1102.5193* vol. 4 (2011), Nr. 1, pp. 1–9

[Vino03] VINOSKI, STEVE: Integration with Web services. In: *Internet Computing, IEEE* (2003), Nr. December, pp. 75–77

[Vino97] VINOSKI, STEVE: CORBA: Integrating diverse applications within distributed heterogeneous environments. In: *Communications Magazine, IEEE* (1997), pp. 1–12

[Webc04] WEB CONSORTIUM, WORLD WIDE: Resource Description Framework (RDF) : concepts and abstract syntax, World Wide Web Consortium (2004)

[Weis91] WEISER, MARK: The computer for the 21st century. In: *Scientific American* vol. 265, New York (1991), Nr. 3, pp. 94–104

[Weis96] WEISER, MARK: *Ubiquitous Computing*. URL http://www.ubiq.com/hypertext/weiser/UbiHome.html

[WHFG92] WANT, ROY ; HOPPER, ANDY ; FALCÃO, VERONICA ; GIBBONS, JONATHAN: The active badge location system. In: *ACM Transactions on Information Systems* vol. 10 (1992), Nr. 1, pp. 91–102

[Wied07] WIEDERHOLD, GIO: Mediators , Concepts and Practice. In: *Interfaces* (2007)

[Wiki00a] WIKIPÉDIA: *Résilience (écologie)*. URL http://fr.wikipedia.org/wiki/Résilience_(écologie)

[Wiki00b] WIKIPEDIA: *Fibonacci number*. URL http://en.wikipedia.org/wiki/Fibonacci_number

[YuML05] YU, PING ; MA, XIAOXING ; LU, JIAN: Dynamic software architecture oriented service composition and evolution. In: *The Fifth International Conference on Computer and Information Technology. CIT 2005.* : IEEE, 2005, pp. 1123–1129

[ZhWa10] ZHAO, RONGYING ; WANG, JU: Visualizing the research on pervasive and ubiquitous computing. In: *Scientometrics* vol. 86 (2010), Nr. 3, pp. 593–612

[ZiPY94] ZIMANYI, ESTEBAN ; PIROTTE, ALAIN ; YAKUSHEVA, TATIANA: Materialization : a powerful and ubiquitous pattern abstraction (1994), pp. 630–641