Scorpio : une Approche d'Adaptation Structurelle de Composants Logiciels - Application aux Environnements Ubiquitaires

Gautier Bastide

Université de Nantes **École des Mines de Douai**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2008

N° attribué par la bibliothèque

Scorpio : une Approche d'Adaptation Structurelle de Composants Logiciels

Application aux Environnements Ubiquitaires

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Gautier BASTIDE

*Le 12 décembre 2007 à l'*UFR *Sciences & Techniques, Université de Nantes,*
devant le jury ci-dessous

Président	:	Henri HABRIAS, Professeur	Université de Nantes
Rapporteurs	:	Noureddine BELKHATIR, Professeur	IMAG
		Lionel SEINTURIER, Professeur	LIFL
Examinateurs	:	Antoine BEUGNARD, Enseignant-chercheur	ENST-Bretagne
		Henri HABRIAS, Professeur	Université de Nantes
		Abdelhak-Djamel SERIAI, Enseignant-chercheur	Ecole des Mines de Douai
		Mourad OUSSALAH, Professeur	Université de Nantes

Directeur de thèse : Pr. Mourad OUSSALAH

Encadrant de thèse : Dr. Abdelhak-Djamel SERIAI

Laboratoire: **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 366-334

SCORPIO : UNE APPROCHE D'ADAPTATION STRUCTURELLE DE COMPOSANTS LOGICIELS

APPLICATION AUX ENVIRONNEMENTS UBIQUITAIRES

Scorpio : an Approach for Software Component Structural Adaptation

Usage for adaptation in Ubiquitous Environments

Gautier BASTIDE

⋈

favet neptunus eunti

Université de Nantes

Gautier BASTIDE

Scorpio : une Approche d'Adaptation Structurelle de Composants Logiciels

Application aux Environnements Ubiquitaires

IV+X+276 p.

Ce document a été préparé avec LaTeX2e et la classe `these-LINA` version v. 2.7 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse : `http://login.irin.sciences.univ-nantes.fr/`.
Cette classe est conforme aux recommandations du ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche (circulaire n° 05-094 du 29 mars 2005), de l'Université de Nantes, de l'école doctorale « Sciences et Technologies de l'Information et des Matériaux » (ED-STIM), et respecte les normes de l'association française de normalisation (AFNOR) suivantes :

- AFNOR NF Z41-006 (octobre 1983)
  Présentation des thèses et documents assimilés ;
- AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure ;
- AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Impression : these.tex – 08/01/2008 – 10:45.

Révision pour la classe : these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp

L’intelligence, c’est la faculté d’adaptation.

— André GIDE.

Résumé

La réutilisation à grande échelle de composants logiciels se révèle être un challenge pour la conception de nouvelles applications. Dans la grande majorité des cas, pour être intégrés à une application, les composants disponibles ont besoin d'être adaptés afin de faire face à la multiplicité des environnements de déploiement dotés de caractéristiques variables. Ainsi, pour éviter le redéveloppement de nouveaux composants et favoriser la réutilisation, de nombreuses approches ont proposé des techniques permettant d'adapter le comportement de composants existants. Cependant, adapter le comportement de composants n'est pas suffisant pour permettre leur réutilisation : il faut également adapter leur structure. Or, aucune approche existante ne permet de répondre pleinement à ces besoins en adaptation structurelle. Ainsi, notre objectif est de proposer une approche, appelée Scorpio, permettant d'adapter la structure de composants. Nous nous focalisons plus particulièrement sur des composants existants. Dans un premier temps, nous nous sommes intéressés à l'adaptation structurelle de composants existants en proposant un processus permettant leur ré-ingénierie vers de nouvelles structures. Puis, pour répondre aux besoins liés à une adaptation sans interruption de l'exécution, nous avons proposé des mécanismes permettant de prendre en charge l'adaptation dynamique de ces composants. Partant du constat qu'un certain nombre d'environnements, tels que les environnements ubiquitaires, nécessite une automatisation du processus d'adaptation, nous avons proposé alors de prendre en charge ces besoins à travers une approche permettant l'auto-adaptation structurelle de composants logiciels. Enfin, nos propositions ont été mises en œuvre d'une part par la réalisation du prototype Scorpio-Tool implémenté en Fractal et d'autre part, par la définition et le développement d'un scénario ubiquitaire permettant l'expérimentation de ces propositions.

Mots-clés : Composant logiciel, réutilisation, ré-ingénierie, adaptation structurelle, adaptation statique, adaptation dynamique, auto-adaptation, restructuration, distribution, refactorisation, orienté objet, Fractal.

Abstract

Software component re-use is a challenge for designing new applications. In many cases, the existing components require to be adapted because of the large variety of existing software and hardware environments. To avoid component redevelopments, many approaches proposed techniques allowing an administrator to adapt the component behaviors. However, in certain cases such as in ubiquitous environments, adapting behavior is insufficient to allow its re-use: it also requires adapting its structure. However, few works propose approaches allowing us to adapt the component structure and in these works, only composite component can be adapted. Thus, we aim at defining, in this thesis, an approach for adapting monolithic and composite component structures in order to increase their reusability. First, we propose a static structural adaptation technique which is based on a restructuring process which allows us to transform an existing component into a component whose structure matches with the new needs. Then, we develop an approach for runtime structural adaptation which is based on a runtime adaptable component model. Finally, we introduce self-adaptation mechanisms into our model, dedicated to components deployed in ubiquitous environments. We experiment our approach by implementing a prototype called Scorpio-Tool, which allows us to adapt Fractal components and by creating an ubiquitous scenario where structural adaptation is required.

Keywords: Software component, software reuse, reengineering, structural adaptation, static adaptation, runtime adaptation, self-adaptation, restructuration, distribution, refactoring, object-oriented, Fractal.

Classification ACM

Catégories et descripteurs de sujets : D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*.

Termes généraux : Algorithms, Design, Experimentation, Performance.

Remerciements

Tout d'abord je remercie mon directeur de thèse, M. Mourad Oussalah, professeur à l'Université de Nantes Atlantique, et mon encadrant M. Abdelhak-Djamel Seriai, enseignant-chercheur à l'école des Mines de Douai, pour leur confiance et les précieux conseils qu'ils m'ont dispensés tout au long de cette thèse.

Mes remerciements s'adressent également à l'ensemble des membres de mon jury. Je suis très honoré que M. Noureddine Belkhatir, professeur à l'IMAG et M. Lionel Seinturier, professeur au LIFL, aient accepté d'être rapporteurs de ma thèse. Je remercie également M. Antoine Beugnard, enseignant-chercheur à l'ENST Bretagne et M. Henri Habrias, professeur à l'Université de Nantes Atlantique pour leur participation à mon jury de thèse en tant qu'examinateurs.

Je remercie M. Philippe Hasbroucq, Chef du Département Informatique et Automatique (IA), pour m'avoir accueilli à l'école des Mines de Douai. J'adresse mes plus vifs remerciements à tous les membres de l'équipe IA et plus particulièrement à M. Noury Bouraqadi, enseignant-chercheur, et M. Stéphane Lecoeuche, Adjoint au Chef du Département IA, pour leurs conseils et le soutien qu'ils m'ont témoigné pendant ces trois ans de thèse.

Je tiens à remercier M. Arnaud Doniec, enseignant-chercheur à l'école des Mines de Douai, pour tous les commentaires qu'il a réalisés sur mon manuscrit.

Je remercie également tous les membres de l'équipe Modal du LINA et plus particulièrement Nassima Sadou et Olivier Le Goaer.

Je n'oublierai pas mes amis Guillaume Grondin, Sylvain Chardigny, Tuan Le Van, Isabelle Bisotto, Michaël Piel, Misagh Shakeri, Houssam Fakih et Abdelaziz Gacemi pour leurs encouragements et leur soutien.

Je remercie également toutes les personnes du laboratoire de l'École des Mines de Douai et du LINA que j'ai côtoyées au cours de cette thèse.

Je tiens également à associer à ces remerciements Mme Danièle Hérin, Professeur au LIRMM, qui m'a fait découvrir la recherche et qui m'a encouragé à poursuivre mes études par une thèse.

Enfin, je remercie vivement mes parents et ma grand-mère pour leur soutien moral et les nombreuses relectures de ce manuscrit.

Sommaire

— *Corps du document* —

— *Annexes* —

— *Pages annexées* —

Introduction

Problématique et objectifs de la thèse

L'ingénierie des composants logiciels [101, 123] vise à réduire les coûts de développement d'une application par la réutilisation et l'assemblage de composants préfabriqués tels que les COTS[1] [67]. Cependant, dans la majorité des cas, les composants logiciels existants ne peuvent pas être réutilisés tels qu'ils sont fournis. En fait, l'utilisation d'un composant logiciel dans un contexte différent de celui pour lequel il a été conçu, relève du défi à cause notamment de la grande diversité des environnements logiciels et matériels existants ne permettant pas de concevoir des systèmes génériques capables d'être déployés sur tout type de support et dans n'importe quel contexte.

Dans les cas où un composant ne peut être réutilisé de manière *ad-hoc*, deux solutions peuvent être envisagées : la première solution consiste à développer à nouveau le composant afin qu'il réponde aux nouvelles attentes liées à son utilisation. Cette solution ne peut être envisagée que dans des cas extrêmes car elle engendre d'importants surcoûts dans la conception d'une nouvelle application logicielle. De plus, elle est contraire aux principes mêmes des composants logiciels qui ont pour objectif principal de permettre la réutilisation à grande échelle. La deuxième solution consiste à adapter les composants logiciels existants afin qu'ils répondent à de nouvelles attentes.

Dans la littérature, de nombreuses approches permettant d'adapter le comportement des composants logiciels ou bien des applications à base de composants ont été proposées. Cependant, adapter le comportement n'est pas toujours suffisant pour la prise en compte de nouveaux besoins. Dans certains cas, l'adaptation de la structure du composant peut se révéler indispensable.

En effet, la nécessité d'adapter la structure d'un composant se ressent fortement dans certains environnements, tels que les environnements ubiquitaires, où l'architecture matérielle de déploiement peut continuellement évoluer. Cette évolution de la configuration de déploiement (par exemple, diminution des ressources disponibles sur un composant matériel, disparition d'un composant matériel, etc.) peut entraîner une dégradation de la qualité de services des applications déployées ou même une rupture de leur continuité de service. Il devient alors indispensable d'adapter la structure de l'application concernée et des entités qui la composent, en fonction des ressources matérielles disponibles de manière à garantir une continuité de service et une qualité de service. L'adaptation de la structure d'un composant peut ainsi consister à le fragmenter de manière à pouvoir ajuster son déploiement aux propriétés de l'architecture matérielle (par exemple, répartition des services du composant suivant les ressources disponibles).

Or, malgré ce besoin d'adapter la structure de composants logiciels, peu d'approches ont été proposées, dans la littérature, pour le prendre en compte. De plus, la plupart des approches qui traitent l'adaptation de la structure de composants proposent leur propre modèle de composants structurellement adaptables. Ceci peut être avantageux pour des nouveaux développements mais non profitable pour des systèmes et des composants déjà existants.

En outre, les approches qui permettent d'adapter la structure de composants logiciels existants présentent de fortes limitations. En effet, elles considèrent les composants monolithiques[2] comme des unités de base à la composition. De ce fait, l'adaptation structurelle n'est possible que pour des composants

[1] Composants sur étagères.

[2] Composants construits comme un seul bloc, ne contenant pas de sous composants.

déjà sous forme composite[3]. Elle consiste généralement à modifier la configuration du composant en utilisant des opérations de base fournies par l'infrastructure logicielle de déploiement (*i.e.* modifier les connexions entre les sous-composants, remplacer des sous-composants, etc.). Concernant les composants monolithiques, la seule adaptation possible réside dans la reconfiguration de ses attributs ; ce qui se révèle insuffisant dans de nombreux cas (par exemple, si aucun nœud de l'infrastructure ne dispose de ressources matérielles suffisantes pour déployer le composant monolithique dans son intégralité).

En se basant sur ces considérations, notre objectif dans cette thèse est d'étudier la problématique de l'adaptation structurelle. Nous proposons d'étudier différentes facettes de cette problématique qui concerne la ré-ingénierie de composants existants pour leur adaptation structurelle, l'adaptation structurelle dynamique de composants et enfin, leur auto-adaptation structurelle.

Plan de ce manuscrit

La problématique traitée dans ce travail résulte de l'étude menée tout au long de la thèse autour des approches d'adaptation dans le domaine de l'ingénierie des composants logiciels. Ce mémoire est organisé en cinq chapitres :

- **chapitre 1 :** *état de l'art : l'adaptation dans l'ingénierie des composants logiciels*
  Ce chapitre permet de situer le sujet de notre étude grâce à l'introduction de la problématique liée à la réutilisation et à l'adaptation logicielle ainsi qu'à la présentation des travaux existants sur ce thème. Ce chapitre est structuré en quatre sections principales. L'objectif de la première section est de positionner la problématique d'adaptation logicielle par rapport à celle liée à la réutilisation. Nous montrons le besoin d'introduire de la variabilité au niveau des entités logicielles et des applications logicielles afin de pouvoir les adapter et ainsi augmenter leur capacité à être réutilisées. Nous introduisons dans la deuxième section les principaux concepts et principes concernant l'adaptation des logiciels de manière générale et ceux à base de composants en particulier. Dans la troisième section, nous proposons une étude comparative des approches existantes permettant l'adaptation par rapport à l'ingénierie des composants logiciels. Ainsi, nous étudions les approches d'adaptation proposées respectivement au niveau infrastructure et au niveau applicatif. La dernière section de ce chapitre se focalise sur l'étude de la problématique d'adaptation dans un type particulier d'environnement : celui des environnements ubiquitaires. Ainsi, nous montrons au travers de cette section les besoins spécifiques des applications ubiquitaires en termes d'adaptation ;

- **chapitre 2 :** *adaptation structurelle de composants logiciels : contexte et problématique*
  Le deuxième chapitre introduit l'approche d'adaptation structurelle que nous avons appelé Scorpio (*Software COmponent stRuctural adaPtatIOn*). Ce chapitre est structuré en cinq sections principales. Dans la première section, nous présentons le cadre dans lequel ces travaux ont été initiés. La deuxième section introduit l'adaptation structurelle ainsi que les concepts qui lui sont associés. Dans la troisième section, nous présentons les motivations de notre approche au travers de ses applications possibles. Ensuite, la quatrième section introduit la démarche pour laquelle nous avons opté afin d'aborder les différentes facettes de l'adaptation structurelle. Enfin, nous terminons ce chapitre en présentant, dans la cinquième section, l'exemple d'une application qui constitue notre cas d'étude tout au long de ce manuscrit ;

[3] Composants contenant, par encapsulation, d'autres composants appelés sous-composants.

- **chapitre 3 :** *adaptation structurelle par la ré-ingénierie de composants existants*
  Dans le troisième chapitre, nous nous focalisons sur l'étude de l'adaptation structurelle de composants existants. Ainsi, ce chapitre introduit notre approche visant à réaliser la ré-ingénierie de ces composants pour mettre à jour leur structure. Ce chapitre est structuré en quatre sections principales. Nous présentons dans la première section le processus permettant la transformation structurelle de composants existants. Ainsi, nous décrivons les objectifs de ce processus, ses contraintes et sa stratégie de réalisation. Nous détaillons également dans cette section les différentes étapes de ce processus. La deuxième section de ce chapitre permet de présenter le modèle de composants que nous proposons pour supporter l'adaptation structurelle. Puis, nous présentons un modèle étendu afin de prendre en compte la distribution du composant résultat de l'adaptation. Enfin, la quatrième section présente notre analyse des performances de l'approche d'adaptation structurelle proposée ;

- **chapitre 4 :** *auto-adaptation de composants logiciels*
  Dans le chapitre précédent, nous avons introduit l'adaptation structurelle sans présenter les mécanismes permettant une prise en compte des aspects dynamiques et automatiques de cette adaptation. Ainsi, l'objectif de ce chapitre est de montrer notre approche d'adaptation structurelle dynamique et automatique. Ce chapitre est structuré en sept sections principales. La première section présente les objectifs de l'auto-adaptation de manière générale puis appliqués aux environnements ubiquitaires en particulier. Dans la deuxième section, nous exposons les besoins de l'auto-adaptation structurelle et présentons notre démarche. La troisième section détaille notre modèle de composants dynamiquement adaptables ainsi que l'architecture d'un composant auto-adaptatif. Ensuite, dans la quatrième section, nous détaillons les processus d'auto-adaptation intégrés aux composants leurs permettant d'adapter automatiquement leur structure en fonction de leur contexte d'exécution. Afin d'illustrer les mécanismes de prise de décisions introduits dans le composant (*i.e.* déclenchement et génération de la spécification du résultat de l'adaptation), nous avons appliqué notre approche sur un type particulier d'environnement : les environnements ubiquitaires. Par ailleurs, nous avons pu constater que l'adaptation d'un composant peut avoir des impacts sur les autres composants de l'application. De ce fait, afin de réaliser l'adaptation d'une application, il est indispensable de concevoir une stratégie de gestion de l'adaptation au niveau macro (*i.e.* niveau global à l'application). Pour cela, nous exposons, dans la cinquième section, deux stratégies pour gérer la coordination de l'adaptation des composants d'une application. Puis, dans la sixième section, nous proposons un processus de ré-ingénierie permettant de transformer un composant existant en un composant auto-adaptatif. Enfin, la dernière section présente notre analyse des performances de l'approche d'auto-adaptation structurelle proposée ;

- **chapitre 5 :** *implémentation et expérimentation de l'adaptation structurelle dans des environnements ubiquitaires*
  Ce chapitre a pour objectif de montrer l'implémentation et l'expérimentation réalisées pour mettre en œuvre toutes les propositions introduites dans le cadre de cette thèse. Il est structuré en quatre sections principales. La première section présente les choix d'implémentation de notre approche. Dans la deuxième section, nous décrivons les principaux éléments d'implémentation de l'outil Scorpio-Tool permettant de réaliser l'adaptation structurelle par la ré-ingénierie de composants existants. Puis, la troisième section détaille l'implémentation des mécanismes d'auto-adaptation. Enfin, dans la dernière section, nous présentons notre expérimentation de l'adaptation structurelle dans un environnement ubiquitaire ;

- enfin, nous concluons ce manuscrit par un rappel des contributions de nos travaux et nous mettons également en évidence les limites et les perspectives d'ouverture de notre étude.

CHAPITRE 1

État de l'art : l'adaptation dans l'ingénierie des composants logiciels

1.1 Introduction

Dans la lignée des approches orientées objet [92, 100], celles à base de composants logiciels [101, 123], visent à faciliter le développement d'applications logicielles en appliquant des procédés de réutilisation à grande échelle. En fait, une application créée en utilisant ce paradigme correspond à un assemblage de briques de base préfabriquées appelées composants logiciels.

Cependant, en pratique, la réutilisation à grande échelle de composants existants peut s'avérer problématique. En effet, pour être réutilisé de manière efficace, un composant préfabriqué doit généralement être adapté avant de l'intégrer dans une nouvelle application. Ce constat est dû, entre autre, à la grande diversité des environnements logiciels et matériels existants ne permettant pas de concevoir des systèmes génériques capables d'être déployés sur tout type de supports et dans n'importe quel contexte. Pour résoudre ces problèmes, beaucoup d'approches permettant d'adapter des applications à base de composants ont été proposées. L'étude de ces approches va nous permettre de montrer certaines de leurs limitations telles que l'impossibilité d'adapter la structure de composants monolithiques existants bien qu'une telle capacité puisse se révéler indispensable dans certains cas.

Dans un premier temps, nous positionnons la problématique d'adaptation logicielle par rapport à celle liée à la réutilisation. Ce positionnement nous permet de montrer le besoin d'introduire de la variabilité au niveau des entités logicielles et des applications logicielles afin de pouvoir les adapter et ainsi afin d'augmenter leur capacité d'être réutilisées. Puis, nous introduisons les principaux concepts et principes relatifs à l'adaptation des logiciels de manière générale et ceux à base de composants en particulier. Ensuite, nous proposons une étude comparative des approches existantes permettant l'adaptation par rapport à l'ingénierie des composants logiciels. Cette étude nous permet de mettre en évidence leurs avantages et leurs limitations, en particulier, par rapport à la prise en compte de l'adaptation de la structure de composants existants. Enfin, nous nous positionnons dans le cadre des environnements ubiquitaires afin de souligner l'intérêt d'adapter la structure de composants logiciels dans ce type d'environnement.

1.2 L'adaptation comme une solution pour la réutilisation d'entités logicielles

En génie logiciel, la réutilisation est au centre de la conception de nouvelles applications à moindre coût de développement. En effet, de plus en plus d'applications sont construites par l'assemblage d'entités logicielles existantes. Cette stratégie est grandement encouragée par le fort développement de nouveaux paradigmes tels que les « composants logiciels », que nous détaillerons par la suite, et qui positionnent la réutilisation au centre du cycle de vie d'une application.

En fait, afin de faire face à la complexité croissante des applications qui rend leur production très coûteuse et peu fiable, diverses solutions ont été proposées. Parmi celles-ci, la réutilisation se définit comme une nouvelle approche de développement d'applications selon laquelle il est possible de construire une application à partir d'entités logicielles existantes ayant été produites à l'occasion de précédents développements et par des personnes généralement différentes de celles qui conçoivent la nouvelle application [115].

La mise en œuvre du principe de réutilisation pour la création de nouvelles applications se révèle être un choix privilégié par la majorité des concepteurs et des développeurs d'applications. En effet, la réutilisation procure de nombreux avantages. A titre d'exemples, nous pouvons citer :

- *l'accroissement de la productivité*
  La première des raisons qui poussent les concepteurs et les développeurs d'applications à réutiliser des entités logicielles pour créer de nouvelles applications est l'accroissement de la productivité. En effet, la réutilisation permet de réduire la quantité de code nécessaire pour réaliser une fonctionnalité donnée [27]. Ainsi, les nouvelles applications ne sont plus construites à partir de rien mais plutôt à partir de briques de base existantes. De ce fait, la durée nécessaire à la conception et au développement d'une application en est réduite, étant donné que certaines parties de l'application ont déjà été conçues et développées, et donc sont disponibles aux développeurs à moindre coût. De plus, la complétude de l'application en est grandie car vu que les développeurs n'ont pas à créer toutes les parties de leur application, il leur est plus facile de créer des applications répondant à un grand nombre de besoins de l'utilisateur ;

- *la focalisation, lors du développement, sur les fonctionnalités spécifiques à l'application*
  La réutilisation permet aux concepteurs d'applications de se consacrer uniquement sur des fonctionnalités spécifiques qui constitueront la valeur ajoutée de l'application ; les autres fonctionnalités proposées ou utilisées (fonctionnelles ou non fonctionnelles) pouvant être issues de la réutilisation d'entités logicielles existantes. Les créateurs d'une nouvelle application n'ont donc pas l'obligation de concevoir et de développer toutes les fonctionnalités fournies ou bien utilisées dans le cadre de l'application qu'ils conçoivent. Ainsi, la durée nécessaire à la création d'une application est réduite et la qualité de l'application est accrue ;

- *la facilitation de la mise à jour d'une application*
  La réutilisation permet également de composer facilement et rapidement des applications de façon modulaire, par assemblage d'entités logicielles existantes avec de nouvelles entités spécifiques à l'application. Il devient alors plus facile de remplacer ou de modifier les entités logicielles mises en jeu par de nouvelles entités existantes qui correspondent à de nouveaux besoins.

1.2.1 La variabilité comme un mécanisme favorisant l'adaptation pour l'ingénierie par réutilisation

Comme nous l'avons vu précédemment, la réutilisation d'entités logicielles pour la création et la modification d'applications offre de nombreux avantages. Cependant, dans la majorité des cas, une entité logicielle ne peut être réutilisée de manière *ad-hoc* (*i.e.* sans subir de mises à jour préalables). Cette affirmation est la conséquence de la présence de nombreux obstacles rendant difficile, voir impossible la réutilisation directe d'entités logicielles existantes. Ainsi, il est indispensable d'introduire dans une entité logicielle de nouvelles propriétés configurables (*i.e.* points de variabilité) lui permettant d'être réutilisée dans de nouvelles conditions.

1.2.1.1 Les obstacles à la réutilisation

Parmi les obstacles à la réutilisation, nous pouvons évoquer à titre d'exemples :

- *la spécificité d'une entité logicielle à un utilisateur ou à des conditions d'utilisation spécifiques*
  Une entité logicielle peut avoir été conçue pour répondre aux besoins spécifiques d'un utilisateur ou d'un groupe d'utilisateurs. Par exemple, un service d'une application ne peut être utilisé que par un certain type d'utilisateur ou dans des conditions très particulières, et spécifiques à l'application pour laquelle il a été conçu initialement. De ce fait, la réutilisabilité de l'entité logicielle concernée est très limitée : elle ne peut généralement être réutilisée que dans la même situation (*i.e.* même type d'utilisateur ou mêmes conditions d'utilisation) ;

- *l'évolution des besoins des utilisateurs et indirectement, des fournisseurs*
  Tout au long du cycle de vie d'une application, les besoins de ses utilisateurs ainsi que de ses fournisseurs évoluent ; ce qui rend la réutilisation difficile dans certains cas. Tout d'abord, du point de vue de l'utilisateur, ses besoins peuvent changer en permanence (besoin de nouvelles fonctionnalités, besoin de mise à jour des fonctionnalités existantes, etc.). Ainsi, la réutilisation d'une entité logicielle qui avait été prévue, par les concepteurs, pour subvenir aux besoins des utilisateurs peut au cours du temps se révéler insuffisante.
  Du point de vue des fournisseurs, ces derniers doivent modifier en permanence leur application de manière à pouvoir répondre à de nouvelles attentes de leurs clients. De ce fait, ils doivent être capables de mettre à jour facilement et rapidement les services qu'ils se proposent de fournir ;

- *l'évolution de l'environnement de déploiement et sa grande diversité*
  L'un des principaux freins à la réutilisation d'entités logicielles existantes est dû à la multiplicité des environnements de déploiement. Tout d'abord, la grande variété d'infrastructures logicielles et matérielles existantes rend impossible la conception d'entités génériques capables d'être déployées dans n'importe quel contexte. De ce fait, les concepteurs ainsi que les développeurs doivent émettre des hypothèses lors de la création de nouvelles entités logicielles. Ces hypothèses peuvent être en contradiction avec les nouvelles conditions d'utilisation. Ainsi, une entité logicielle peut ne pas être réutilisable dans certains contextes. Par exemple, dans le cadre des environnements à ressources limitées, une entité logicielle peut ne pas pouvoir être réutilisée si les ressources qu'elle requiert pour être exécutées sont insuffisantes.
  Par ailleurs, l'environnement d'exécution d'une application peut évoluer tout au long de son cycle de vie. En effet, au cours de son exécution, de nouveaux matériels peuvent apparaître ou dis-

paraître ; ceci pouvant influencer le comportement de l'application. Ainsi, les entités logicielles doivent être capables de prendre en compte ces évolutions de l'infrastructure matérielle ;

- *une mauvaise conception des entités logicielles*
  La mauvaise conception des entités logicielles peut également être un obstacle à la réutilisation. Par exemple, une entité logicielle peut avoir été conçue comme une unité monolithique alors qu'elle fournit des ensembles de services indépendants. De ce fait, ces services ne peuvent être réutilisés de manière séparée. Or, si certaines conditions (ressources limitées, conditions d'utilisation spécifiques pour certains services, sécurité, etc.) ne permettent pas d'utiliser l'entité intégralement, la réutilisation ne peut pas être effective.

1.2.1.2 La variabilité pour faciliter la réutilisation

Pour franchir les obstacles liés à la réutilisation d'entités logicielles que nous avons définis précédemment, il est indispensable de leur introduire de nouvelles propriétés configurables permettant de les réutiliser dans de nouvelles conditions.

Toute entité logicielle est constituée d'une partie fixe et d'une partie variable [115] qui contient un ensemble de points de variabilité. Un point de variabilité correspond à une partie abstraite d'un artefact. Il s'agit d'une partie qui admet différentes représentations [60]. A titre d'exemples, nous pouvons citer les points de variabilité suivants : variabilité à la plate-forme d'exécution (*i.e.* l'entité logicielle peut être réutilisée avec différentes plates-formes d'exécution), variabilité aux besoins des utilisateurs (*i.e.* l'entité logicielle peut être réutilisée suivant différents besoins de l'utilisateur), etc.

La taille et le contenu de ces deux parties (partie fixe et partie variable) conditionnent sa réutilisation d'une entité logicielle : plus la partie variable contient de points de variabilité, plus l'entité logicielle devient réutilisable. Pour réutiliser une entité logicielle, il faut fixer les points de variabilité en fonction de son nouveau contexte d'utilisation. Ces points peuvent être fixés à tout moment dans le cycle de vie d'une entité logicielle : aussi bien avant son déploiement que pendant son exécution.

L'introduction de nouveaux points de variabilité à une entité logicielle existante va donc permettre d'agrandir sa partie variable ; ce qui va accroître sa capacité à être réutilisée : elle va pouvoir être réutilisée dans plus de situations. Les points de variabilité à introduire dans une entité logicielle existante sont généralement étroitement liés à l'application concernée et à son nouvel environnement d'exécution. Nous étudierons, par la suite (voir Section 1.5), les points de variabilités nécessaires à la réutilisation d'entités logicielles dans le cadre d'environnements ubiquitaires.

1.2.2 Techniques de mise en place de points de variabilité par l'adaptation

Afin d'accroître la réutilisabilité d'entités logicielles existantes, des points de variabilités peuvent leur être introduits à différents niveaux de leur cycle de vie. Pour cela, deux stratégies peuvent être envisagées :

- *l'introduction de la variabilité lors de processus d'ingénierie logicielle*
  La première stratégie consiste à introduire des points de variabilité au moment de la phase de création d'une entité logicielle (voir Figure 1.1, stratégie A). Elle passe généralement par la conception de modèles ou de méta-modèles dotés de points de variabilités destinés à favoriser la réutilisation des entités logicielles conçues à partir de ces modèles. Cette stratégie est la plus utilisée dans les approches existantes. Par exemple, Boinot *et al* définissent dans [28] un modèle de compo-

sants appelés « adaptive component » capables de modifier leur comportement en fonction de leur contexte d'exécution. Ce modèle met en jeu le patron de conception « Strategy » que nous présenterons par la suite (voir Section 1.4) et qui permet de définir plusieurs implémentations possibles pour un même service ; le choix de l'implémentation à utiliser est alors déterminé dynamiquement en fonction du contexte. D'autres approches telles que Safran proposé par David dans [47] ou ACEEL proposé par Chefrour dans [44] utilisent la réflexion pour adapter, en fonction du contexte, le comportement ou la structure de composants conformes aux modèles qu'ils proposent. D'autres approches telles que les solutions à base d'aspects [103], permettent d'intégrer des points de variabilités au moment de l'ingénierie de composants logiciels.
Cependant, étant donné que la variabilité ne peut être mise en œuvre au moment de la conception des entités logicielles, elle n'est utilisable pour des entités existantes que si un processus permettant d'obtenir une entité conforme au modèle proposé à partir d'une entité existante est fourni ; ce qui limite son champs d'action si tel n'est pas le cas ;

- *l'introduction de la variabilité lors de processus de ré-ingénierie logicielle*
  La deuxième stratégie consiste à introduire des points de variabilité à des entités logicielles après leur création (voir Figure 1.1, stratégie B). Cette stratégie nécessite une mise à jour de l'entité à réutiliser. Cette mise à jour passe généralement par la proposition d'un processus permettant d'intégrer à une entité logicielle existante de nouveaux points de variabilité ou bien de modifier ceux existants afin d'améliorer leur réutilisabilité. Un tel processus doit prendre en paramètres deux éléments : d'une part, l'entité logicielle existante qui doit être réutilisée et d'autre part, son nouveau contexte d'utilisation. Ainsi, le processus devra accomplir l'ensemble des tâches permettant d'obtenir, à partir de l'entité fournie, une entité capable d'être utilisée dans le contexte spécifié. Ce type de processus met en œuvre des techniques de transformation de code telles que la refactorisation [90] que nous détaillerons dans la section 1.4. En fait, ces techniques consistent à modifier l'implémentation d'un composant afin de la rendre conforme à un nouveau besoin.
  Cette stratégie peut également être utilisée en complément de la première si les points de variabilité instaurés ne sont pas suffisants pour favoriser la réutilisation.

1.2.3 Mises à jour logicielles : adaptation, évolution ou maintenance ?

Dans la littérature, la mise à jour d'une entité logicielle se décline en trois approches : la maintenance, l'évolution et l'adaptation. Selon les différentes communautés de recherche existantes, ces trois termes sont utilisés soit conjointement, l'un semblant compléter les autres, soit indépendamment, les uns pouvant remplacer ou bien englober les autres.

1.2.3.1 L'adaptation

Selon le dictionnaire Hachette [61], « adapter » est une action qui consiste à rendre un dispositif, des mesures, etc., aptes à assurer ses fonctions dans des conditions particulières ou nouvelles.

Dans le domaine des systèmes logiciels, « adapter » signifie modifier le système afin de lui permettre de se comporter correctement dans des environnements et des contextes différents [82]. En effet, une même application peut être déployée dans des environnements totalement différents (au niveau des ressources fournies par l'architecture matérielle de déploiement, au niveau des conditions d'utilisation, etc.) ; ce qui lui impose, dans certaines conditions, de s'adapter à la situation afin de garantir la disponibi-

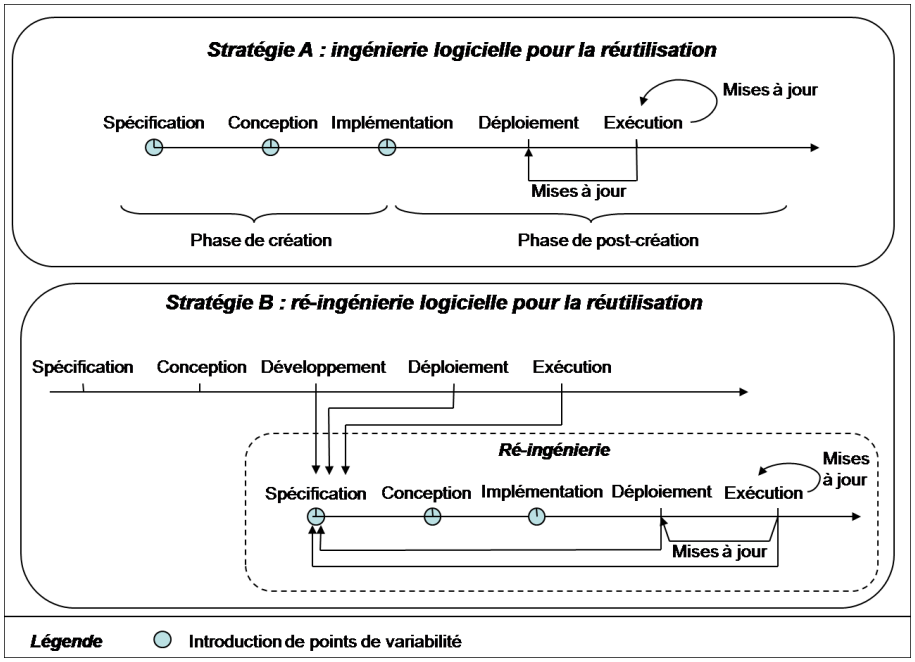

Figure 1.1 – Deux stratégies pour la mise en place de points de variabilité pour la réutilisation

lité et la performance des services qu'elle propose. De plus, de nombreux évènements peuvent intervenir tout au long du cycle de vie d'une application [64] ; certains pouvant affecter son comportement ou sa structure. Par exemple, des modifications de l'infrastructure de déploiement (*i.e.* suppression de périphériques, évolution des caractéristiques techniques des machines de déploiement, etc.) peuvent engendrer de graves conséquences sur le comportement de l'application telles qu'une rupture de la continuité de service ou bien une dégradation de la qualité de service [3]. Ces évènements peuvent être la conséquence de nombreux éléments tels que l'impossibilité de prévoir, au moment de la conception d'une application et cela, de manière exhaustive, tous ses cas d'utilisation (cette affirmation est la conséquence de la multiplication des environnements logiciels et matériels ainsi que l'évolution de la manière d'interagir entre l'application et l'utilisateur liée notamment à l'apparition des applications ubiquitaires et mobiles) ou bien l'introduction d'erreurs de conception ou de développement qui n'ont pas été détectées lors des phases de tests de l'application. Ainsi, pour lutter contre ce type de problèmes pouvant intervenir lors de l'exécution d'une application, l'adaptation apparaît comme une des solutions envisageables.

Nous pouvons noter qu'il est essentiel de différencier une phase d'adaptation d'une phase de personnalisation (en anglais « *customisation* » [64]. En effet, la personnalisation permet, uniquement à l'utilisateur, de modifier la structure ou le comportement des entités logicielles ou d'applications logicielles de par le paramétrage de certaines options proposées par l'entité ou l'application. En aucun cas, elle ne permet d'introduire des variabilités pour la réutilisation. En fait, son rôle est de permettre à l'utilisateur de paramétrer certains points de variabilité pendant l'exécution de l'application concernée.

1.2.3.2 La maintenance

Selon le dictionnaire Larousse [106], la maintenance est définie comme « l'ensemble des opérations permettant de maintenir ou de rétablir un système, un matériel, un appareil, etc. dans un état donné ou de lui restituer des caractéristiques de fonctionnement spécifiées ».

Dans le domaine du génie logiciel, la maintenance, telle qu'elle est définie dans la norme ISO 12207 (*Information Technology - Software Life Cycle Processes*), désigne « le processus mis en œuvre lorsque le logiciel subit des modifications relatives au code et à la documentation correspondante. Ces modifications peuvent être dues à un problème, ou encore à des besoins d'amélioration ou d'adaptation. L'objectif est de préserver l'intégrité du logiciel malgré cette modification. On considère en général que ce processus débute à la livraison de la première version d'un logiciel et prend fin avec son retrait ».

En d'autres termes, la maintenance désigne les actions pouvant être menées afin de prévenir ou de traiter les défaillances d'une application ou d'un système pouvant intervenir après sa mise à disposition [54]. Par exemple, il est possible qu'après la mise en service d'une application, celle-ci ne fonctionne pas correctement, des erreurs étant toujours présentes. La maintenance est alors chargée de corriger ces erreurs [43]. De plus, lorsqu'elle est utilisée une application se dégrade généralement au bout d'un laps de temps fini ; ceci pouvant être du à la mauvaise conception de certaines parties de l'application. La maintenance à également pour objectif de pallier à cette dégradation.

Ainsi, les principales actions menées dans le cadre de la maintenance ont pour principal objectif de corriger d'éventuelles défaillances d'une application, d'accroître sa robustesse ou bien de renforcer sa maintenabilité [54] (par exemple, en augmentant la lisibilité de son code source de par l'ajout de commentaires ou l'élimination de codes dupliqués).

La technique la plus couramment utilisée pour la maintenance d'applications logicielles est la restructuration que nous détaillerons par la suite (voir Section 1.4.2.7). Cette technique permet d'améliorer la qualité d'un code et ainsi, sa maintenabilité, sans en changer son comportement.

Selon la communauté « adaptation », la maintenance est considérée comme une autre forme particulière de l'adaptation qui est appelé « adaptation corrective » [75]. Cette forme d'adaptation consiste à corriger les erreurs de fonctionnement d'une application. Ainsi, elle fait référence à une vision réductrice de la maintenance.

1.2.3.3 L'évolution

Selon le dictionnaire Larousse [106], « l'évolution » désigne une transformation graduelle et continuelle.

Contrairement au terme « maintenance », dans le domaine du génie logiciel, il n'existe aucun standard destiné à définir le terme d'évolution. De ce fait, dans la littérature, différentes définitions ont été proposées. Cependant, ce terme reste relativement ambigu.

Selon le point de vue de Chapin [43], l'évolution est considérée comme un sous-ensemble d'activités de la maintenance à savoir l'ajout, la suppression et la modification de propriétés fonctionnelles. Cette définition est également utilisée dans la communauté « adaptation » qui considère l'évolution comme une forme d'adaptation appelée « l'adaptation évolutive » [75].

Cependant, cette définition parait relativement réductrice au vue des travaux qui ont été réalisés sous la thématique de l'évolution [108]. La définition la plus adoptée par la communauté de l'évolution englobe beaucoup plus d'activités que celle fournie par Chapin [43]. En fait, l'évolution est définie comme l'ensemble des activités relatives à l'analyse et à la modification d'un logiciel après sa mise en service [54, 88]. Ainsi, au vue de cette définition, le terme d'évolution est substituable à celui de maintenance.

Cependant, contrairement à la communauté « maintenance » qui s'intéresse essentiellement au code source d'une application et à sa mise à jour, la communauté « évolution » couvre non seulement l'évolution du code, mais aussi l'évolution de l'architecture, des données et des schémas, et plus généralement de tout artefact intervenant dans un système logiciel, qu'il soit patrimonial ou récent. Ainsi, selon cette vision, la maintenance serait inclue dans l'évolution.

1.2.4 Bilan sur le positionnement de l'adaptation par rapport à la réutilisation

Comme nous avons pu le constater, la réutilisation est au centre de l'ingénierie logicielle car elle permet de diminuer le coût et le temps de production. Cependant, dans la majorité des cas, la réutilisation ne peut être réalisée de manière *ad-hoc* : il est nécessaire d'introduire des points de variabilité dans les entités logicielles existantes. Leur introduction passe par la mise à jour des entités logicielles concernées. Dans la littérature, cette mise à jour se décline en trois catégories : la maintenance, l'évolution et l'adaptation.

Cependant, hormis pour la maintenance qui est définie par des standards, il n'existe pas de définition unique admise par la communauté du génie logiciel, pour les concepts d'adaptation et d'évolution. De nombreux travaux ont tentés de positionner ces techniques les unes par rapport aux autres. Par exemple, dans [64], Heineman et Ohlenbusch font la distinction entre l'évolution de logiciel, où les concepteurs des composants modifient le composant logiciel qu'ils ont conçu, et l'adaptation, où un constructeur d'applications adapte un composant pour probablement un usage différent. Si on demandait à un concepteur de composant d'adapter un composant, il choisirait probablement un ensemble minimal de changements à cause de sa connaissance directe du composant. Le constructeur d'applications n'a pas cet avantage, et il pourra acquérir cette connaissance simplement du code source et de la documentation. Le constructeur d'application a donc toujours besoin d'aide pour adapter avec succès des composants. Aussi, on parle de personnalisation lorsque la modification est réalisée exclusivement au moyen de l'API du logiciel et/ou en exploitant les diverses options fournies.

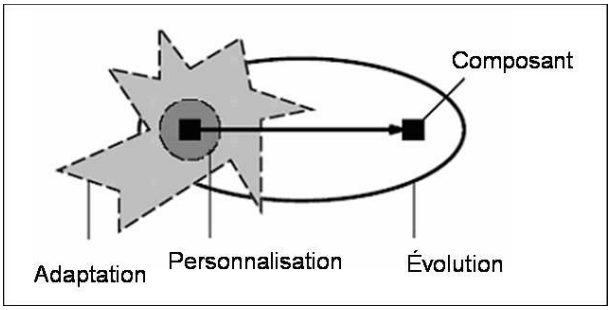

Figure 1.2 – Positionnement des concepts d'adaptation, de personnalisation et d'évolution

La figure 1.2 [64] présente une perspective sur l'adaptation d'un composant. Suivant cette dernière, étant donné un composant logiciel (représenté par un petit carré noir), le grand ovale représente l'espace des chemins possibles d'évolution pour un composant, dont un est montré par une flèche. La distance entre les deux composants est proportionnelle à la différence entre les composants. Le composant a un ensemble pré-emballé d'options qui permet la personnalisation, elle est représentée par le petit cercle gris-foncé. Ainsi, la différence apparente entre un composant adapté aux besoins du client et son original est très petite. La région gris-clair curieusement formée représente les adaptations possibles qui peuvent

être effectuées par un constructeur d'application. Le secteur pour chaque région est proportionnel aux situations dans lesquelles le composant peut être réutilisé.

Or, compte tenu des définitions présentées précédemment, nous pouvons considérer, comme nous allons le constater par la suite, que, dans notre cas, les termes d'adaptation, d'évolution et de maintenance sont substituables. Ainsi, dans la suite de ce manuscrit, nous avons opté pour l'utilisation de la terminologie de l'adaptation pour désigner une activité de mise à jour.

1.3 L'adaptation dans l'ingénierie des logiciels à base de composants : présentation et concepts

Dans cette section, nous allons étudier, en détails, le principe de l'adaptation d'une application logicielle. Puis, nous nous somme focalisés plus particulièrement sur les composants logiciels et les applications conçues à base de composants car ils constituent notre cadre de travail.

1.3.1 Caractérisation de l'adaptation d'une application logicielle

Pour être adaptable une entité logicielle (constituant ou constitué) doit respecter un certain nombre de propriétés tels qu'un schéma défini pour son processus ou un certain nombre d'invariants que ce processus doit vérifier. Nous décrivons dans la suite de cette section ces différentes propriétés.

1.3.1.1 Caractéristiques d'une application adaptable

En théorie, toute application ou système peut être adapté. Cependant, le résultat de l'adaptation et sa facilité de mise en œuvre sont conditionnés par les propriétés de l'entité logicielle que l'on souhaite adapter. Ainsi, selon Ledoux [82], pour être facilement adaptable, une application ou un système doit posséder au minimum trois caractéristiques qui sont la modularité, la composabilité et la capacité à s'observer.

1. *La modularité*
   Tout d'abord, une application ou un système doit être construit de façon modulaire afin de pouvoir facilement l'adapter et le configurer à tout moment [104]. La modularité est une approche structurante qui sépare un logiciel en petites unités qui, rassemblées, composeront l'ensemble du logiciel.
   Cette propriété permet de le modifier ou de remplacer certaines parties avec un minimum d'interférences sur les autres parties qui le composent ; ce qui rend l'adaptation plus aisée.

2. *La composabilité*
   Les constituants d'une application ou d'un système doivent être composables, c'est-à-dire, ils doivent présenter la capacité de s'assembler. La composabilité est une mesure des capacités d'assemblage entre les éléments constituants d'une application ou d'un système [82].
   Cette propriété permet de le découper puis de le recomposer tout en réalisant l'adaptation. De ce fait, plus les constituants d'un système sont composables, plus l'application ou le système sera adaptable.

3. *L'introspection*
   Une autre caractéristique que doivent posséder les constituants d'une application ou d'un système adaptable est relative à leur capacité à s'observer et à raisonner sur leur propre état (*i.e.* introspection). En effet, l'analyse de son comportement constitue une étape indispensable de l'adaptation. L'observation va permettre au concepteur de prendre les mesures nécessaires pour réaliser l'adaptation, ou bien de proposer à l'administrateur des méthodes pour réaliser l'adaptation ou encore prendre automatiquement des mesures (*i.e.* auto-adaptation) à condition que l'application possède des capacités d'intercession.

Toujours selon Ledoux [82], une autre caractéristique peut être requise dans certains cas : *l'intercession*. Il s'agit de la capacité que peut posséder une application ou un système pour agir sur lui-même. Cette propriété est très importante pour adapter une application ou un système de manière dynamique. En fait, dans ce cas, l'application ou le système doit pouvoir modifier son comportement ou sa structure sans arrêter son exécution.

1.3.1.2 Caractérisation d'un processus d'adaptation d'une application

Un processus d'adaptation est régi au travers d'un ensemble de tâches à réaliser qui sont regroupées dans quelques étapes principales. Pour préserver un fonctionnement sain du système à adapter, ces étapes sont contraintes à respecter un certain nombre d'invariants. Par ailleurs, des propriétés permettent d'évaluer la qualité du processus d'adaptation. Nous présentons toutes ces facettes d'un processus d'adaptation dans cette section.

Les invariants d'un processus d'adaptation Pour être valide, tout processus d'adaptation d'une application ou d'un système doit préserver un certain nombre de propriétés qui ont été étudiées par Occello dans [97] et qui lui permettent d'assurer que l'adaptation s'exécute correctement :

- *la stabilité du système*
  L'adaptation ne doit pas entraîner une dégradation de la stabilité du système, c'est-à-dire, il doit rester capable d'éviter les effets inattendus d'une modification du système ;

- *la cohérence*
  L'adaptation étant provoquée par une évolution du contexte d'exécution de l'application (*i.e.* interaction entre l'application et le contexte), le comportement de l'application et sa structure doivent pouvoir évoluer en permanence avec le contexte. Or, certains contrôles peuvent se révéler incompatibles donc, après adaptation, des incohérences peuvent apparaître. Il est donc indispensable d'assurer la cohérence de l'adaptation. Une phase d'adaptation ne peut être lancée que lorsque l'application se trouve dans un état cohérent et le résultat d'une phase d'adaptation doit être une application se trouvant dans un état cohérent.
  La cohérence des modifications doit être gérée à tous les niveaux de granularité de l'application. Par exemple, dans l'ingénierie des composants logiciels, la cohérence doit être gérée sur deux niveaux : le niveau local aux composants (*i.e.* niveau micro) et le niveau global à l'application (*i.e.* gestion de la cohérence au niveau de l'architecture de l'application : niveau macro) ;

- *la sûreté de fonctionnement*
  Une phase d'adaptation ne doit pas entraîner de disfonctionnement de l'application ;

- *la transparence*
  Les services offerts par l'application doivent pouvoir être invoqués de la même manière avant et après son adaptation. De plus, les services non fonctionnels issus de l'adaptation ne doivent pas être accessibles par les autres entités logicielles qui composent l'application. Par exemple, si l'adaptation entraîne un partage des ressources dans un environnement distribué, les services de maintien de la cohérence doivent être cachés au reste de l'application et donc inaccessibles pour les éléments non concernés par le partage ;

- *l'interopérabilité*
  Une phase d'adaptation ne doit pas entraîner de pertes au niveau de l'interopérabilité des entités logicielles qui composent l'application. Par exemple, si l'entité logicielle adaptée pouvait, avant son adaptation, être assemblée avec un ensemble d'autres entités logicielles, après son adaptation, elle doit pouvoir être assemblée au minimum avec les mêmes entités ;

- *l'extensibilité*
  Plusieurs processus d'adaptation différents peuvent être mis en place successivement sans que cela n'entraîne de disfonctionnement de l'application.

Les différentes étapes d'un processus d'adaptation Tout processus d'adaptation d'une application comporte trois grandes étapes [45] ; chacune pouvant être réalisée manuellement (*i.e.* avec l'intervention du concepteur ou de l'administrateur) ou bien automatiquement par l'application elle-même ou par un outil dédié (voir Figure 1.3). Ces étapes sont les suivantes :

- *le déclenchement d'une phase d'adaptation*
  La première étape d'un processus d'adaptation consiste à déterminer quels sont les évènements susceptibles d'entraîner une phase d'adaptation de l'application. Pour cela, il est nécessaire dans un premier temps de prendre en compte tous les éléments du contexte pouvant influer sur le cycle de vie de l'application puis de détecter toutes modifications susceptibles d'avoir des répercutions sur le comportement de l'application. Il faut donc définir au cours de cette étape quels sont les évènements déclencheurs d'une phase d'adaptation ;

- *la prise de décisions*
  Cette étape consiste à déterminer les modifications qui doivent être réalisées pour réagir à la première étape. Ainsi, il s'agit de déterminer quelles sont les actions à effectuer pour réaliser l'adaptation de l'application et établir une stratégie pour appliquer ces opérations (*i.e.* choix de la stratégie d'adaptation). Cette étape doit s'appuyer sur les éléments du contexte qui ont déclenché l'adaptation afin de prendre en compte les évolutions et de proposer un nouveau comportement ou une nouvelle structure de l'application adapté à la situation courante ;

- *la réalisation de l'adaptation*
  La dernière étape d'un processus d'adaptation consiste à exécuter les instructions qui vont permettre d'adapter l'application à son contexte d'exécution en fonction de la technique d'adaptation. Cette étape est étroitement liée aux éléments dont peut disposer les mécanismes de réalisation de l'adaptation ainsi qu'aux modèles et aux langages utilisés. Les mécanismes réalisation de l'adap-

tation peuvent être statiques de par la manipulation du code de l'application afin de l'adapter à de nouveaux besoins ou bien dynamiques. La mise en œuvre des mécanismes dynamiques requiert généralement certaines propriétés sur les modèles utilisés telles que l'introspection et l'intercession. De plus, ils utilisent des opérations standard qui sont généralement fournies avec l'environnement de base.

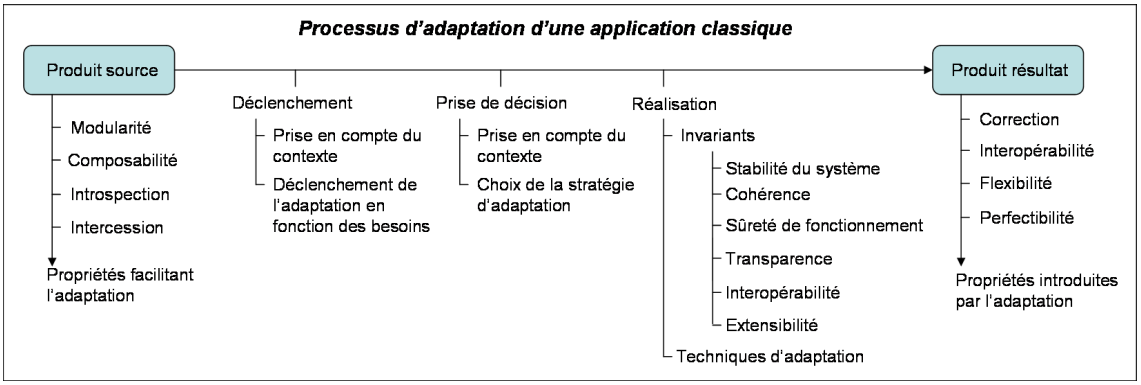

Figure 1.3 – Processus d'adaptation d'une application logicielle

La qualité d'un processus d'adaptation La qualité d'un processus d'adaptation peut être mesurée suivant trois critères que nous définissons ci-dessous :

- *le degré d'automatisation de l'adaptation*
  La qualité d'un processus d'adaptation se mesure généralement par son degré d'automatisation, c'est-à-dire la capacité de l'application à réaliser sa propre adaptation en fonction de son contexte d'exécution. En fait, l'intervention de l'utilisateur doit être minimale. Cependant, elle ne doit pas être nulle car des études [11] ont prouvé que si l'utilisateur n'intervient pas lors de l'exécution d'une application, sa satisfaction est moindre car il a l'impression de ne pas avoir le contrôle sur l'application. En fait, pour déterminer le degré d'automatisation d'une application, il est utile de poser des questions telles que : quels sont les acteurs de l'adaptation ? Quelle est la place de l'utilisateur et de l'administrateur dans l'adaptation de l'application ?

- *la variété des opérations d'adaptation*
  Par ailleurs, la qualité d'un processus d'adaptation se mesure également par la variété des opérations d'adaptation disponibles et leurs impacts sur le résultat de l'adaptation. En fait, ces opérations vont conditionner les réactions de l'application suivant le contexte dans lequel elle est exécutée. Il est alors possible de tester la qualité de l'adaptation d'une application en modifiant certaines caractéristiques de l'architecture matérielle ou logicielle de déploiement, et en observant le comportement de l'application dans ces nouvelles conditions ;

- *la capacité de prise de décisions*
  De plus, la qualité de l'adaptation est étroitement liée au type d'application que l'on cherche à adapter. En effet, dans la majorité des cas, il est impossible de réaliser une adaptation « parfaite » de l'application à son contexte. Le processus d'adaptation doit donc faire des choix afin de

privilégier telles ou telles fonctionnalités par rapport à d'autres tout en tenant compte des besoins de l'utilisateur et de l'application. En conséquent, la qualité de l'adaptation d'une application se mesure également par sa capacité de prise de décisions (*i.e.* prendre les bonnes mesures afin de tenir compte de tous les éléments qu'elle a en sa possession).

1.3.1.3 Caractérisation du résultat de l'adaptation

Comme nous l'avons évoqué précédemment, le résultat de l'adaptation dépend des propriétés du produit à adapter (*i.e.* source de l'adaptation) mais aussi des techniques utilisées pour réaliser l'adaptation (voir Section 1.4) qui peuvent agir à différents niveaux (par exemple, adaptation de comportement, de la structure, etc.) et ainsi avoir des conséquences différentes sur le résultat de l'adaptation.

La qualité du résultat d'une adaptation dépend du point de vue dans lequel on se positionne. Par exemple, pour un développeur, la qualité de l'adaptation peut se mesurer en fonction du code produit : plus le code généré par l'adaptation sera lisible, de par l'introduction de commentaires, la mise en œuvre d'un nommage sémantique (*i.e.* les variables, méthodes, classes ou autres doivent être nommées en fonction de leur sémantique), la factorisation du code (*i.e.* élimination de code dupliqué), etc., plus la satisfaction du développeur sera importante.

Du point de vue de l'utilisateur, la qualité du résultat de l'adaptation se mesure généralement par sa satisfaction. Celle-ci peut être la conséquence d'éléments divers tels que l'assurance de la continuité de service (*i.e.* les services fournis par l'application sont disponibles en continu) et de la qualité de service (*i.e.* performance des services, adéquation à la situation dans laquelle se trouve l'utilisateur).

1.3.2 Les composants logiciels : un paradigme qui supporte l'adaptation

Dans la section précédente, nous avons étudié l'adaptation de manière générale. Maintenant, nous allons nous focaliser sur l'adaptation par rapport à l'ingénierie des composants logiciels.

1.3.2.1 Les principes d'une approche à base de composants

L'ingénierie des composants logiciels [123] définit une application comme étant l'assemblage d'unités logicielles indépendantes appelées « composants logiciels ».

Définitions Étant donné que le paradigme « composant logiciel » est relativement récent, ses concepts ne sont pas souvent exprimés clairement. Actuellement, il n'existe aucun standard. De ce fait, le terme de « composant logiciel » ne possède pas de définition unique. Voici quelques exemples de définitions qui ont été proposées dans la littérature :

Une des premières définitions du mot « composant » a été donnée en 1995 par Jed Harris, Président du CI Lab :

> *« un composant est un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support. De plus, il est doté d'interfaces standard pour pouvoir interopérer. »*

D'autres définitions proposées dans la littérature insistent plus sur les notions de services fournis et services requis par le composant. Par exemple, celle proposée par Chefrour dans [45] est la suivante :

> *« Un composant est une entité logicielle qui fournit un service particulier via une interface séparée de l'implantation mettant en œuvre ce service. »*

Une des définitions d'un composant logiciel, les plus citées dans la littérature, est celle proposée par Szyperski *et al.* dans [123] :

> *« un composant est une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites sur son contexte. Un composant peut être déployé de manière indépendante et il est sujet à compositions par des parties tierces. »*

Cette dernière définition met en avant les notions d'assemblage et d'autonomie des composants. En d'autres termes, un composant logiciel est une entité logicielle indépendante pouvant être installée sur différentes plates-formes, configurée et capable de s'auto-décrire. Tout d'abord, le caractère d'indépendance des composants est une propriété essentielle de cette technologie car elle procure aux composants une autonomie leurs permettant d'être réutilisés facilement ; ce qui est l'objectif premier de ce paradigme. Concernant la propriété d'auto-description, celle-ci procure aux composants des propriétés d'introspections qui sont indispensables à la mise en œuvre de leurs adaptations. La propriété de configurabilité offre, quand à elle, à un acteur de l'adaptation (voir Section 1.4.2.5) la possibilité de modifier facilement le comportement d'un composant (en modifiant les valeurs de ses attributs configurables).

Structure d'un composant La tâche d'un composant est de fournir des services et pour cela ils peuvent avoir besoin de services fournis par d'autres composants pour assurer son fonctionnement (voir Figure 1.4). Ainsi, un composant doit pouvoir fournir une description de ses services fournis et requis ainsi que les règles d'interconnexion. De ce fait, un composant logiciel doit être construit sur deux niveaux : un niveau implémentatoire et un niveau architectural.

Le niveau implémentatoire contient le code source correspondant à l'implémentation des services fournis par le composant (*i.e.* hiérarchie de classes représentant le code de l'implémentation de ses services et de ses interfaces).

Quant au niveau architectural, il contient une description de la structure externe du composant (*i.e.* description des services fournis et requis par le composant) ainsi qu'une description de sa structure interne (*i.e.* description de chaque sous-composant, de leur configuration et des connexions entre ces derniers). Ces descriptions sont généralement réalisées par l'intermédiaire de langages dédiés pour la plupart basées sur XML. Ces langages permettent également de décrire soit le contenu de ses interfaces (*i.e.* description des services : paramètres, comportement, etc.) soit l'architecture de l'application (*i.e.* instanciation des composants, assemblage des composants, paramétrisation des services techniques, etc.). Les langages de description d'interfaces sont appelés IDL (*Interface Description Language*) et ceux de description d'architecture sont appelés ADL (*Architecture Description Language*) [87].

La structure d'un composant et ses propriétés dépendent du modèle utilisé pour le construire. Ces modèles peuvent être hiérarchiques ou non. Dans le cadre de modèles hiérarchiques, les composants sont de deux types : des composants composites et des composants monolithiques. Les composants composites disposent d'une structure externe définissant l'ensemble des services fournis et requis et d'une structure interne définissant l'ensemble des composants qu'ils encapsulent (appelés sous-composants) ainsi que leur configuration d'assemblage (voir Figure 1.4). Les composants monolithiques sont quant à eux définis par leur structure externe : ils sont construits comme un seul bloc et donc ne contiennent pas de sous-composants. Les modèles de composants non hiérarchiques ne définissent quant à eux que des composants monolithiques.

Cycle de vie d'un composant Chaque composant est doté d'un cycle de vie qui part de la spécification jusqu'à l'exécution en passant par le « *packaging* » et le déploiement. La prise en considération de la totalité du cycle de vie d'un composant lui permet d'accroître son autonomie et de faciliter son adaptation.

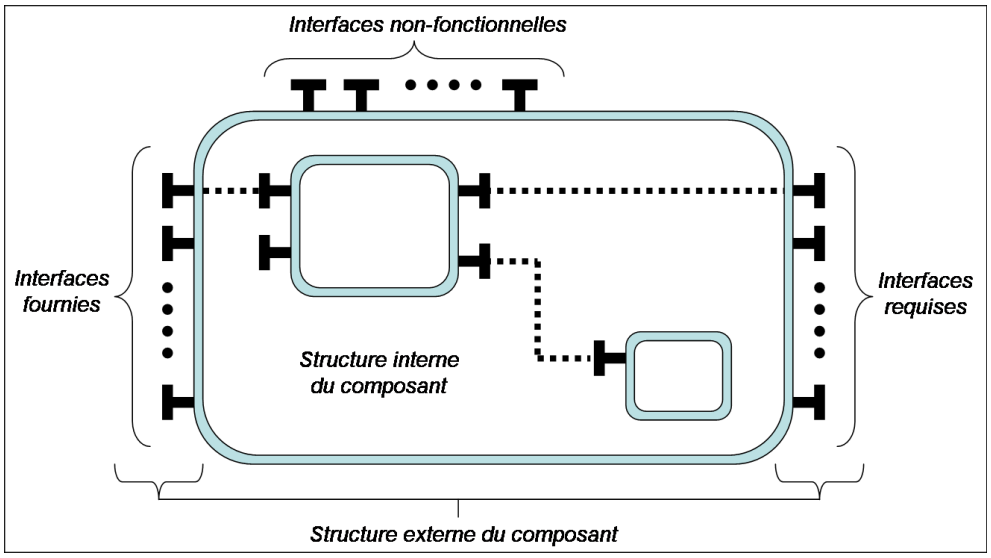

Figure 1.4 – Architecture d'un composant logiciel de type composite

En effet, l'adaptation peut être mise en œuvre au niveau des composants à tout moment dans leur cycle de vie. De plus, la majorité des modèles composants existants offrent la possibilité, durant l'exécution des composants, de contrôler leur cycle de vie. En fait, des mécanismes permettent de démarrer, de stopper ou de mettre en pause un composant. De tels mécanismes facilitent donc l'adaptation statique. En effet, un composant peut être arrêté, adapté puis redémarré.

Modèles de composants Dans la littérature, de nombreux modèles de composants logiciels ont été proposés. Ces derniers peuvent être classifiés en trois catégories [85] : les modèles académiques tels que Fractal [39] et Darwin [83]. Leur objectif premier est de mener des études sur un ou plusieurs aspects particuliers du paradigme de « composant logiciel » ; les modèles industriels tels que JavaBean [62], Enterprise Java Beans (EJB) [86] ou COM/COM+ [31]. Ces modèles de composants répondent plus aux besoins des concepteurs d'applications industrielles ; et enfin, les modèles de références dont l'objectif principal est de définir des normes tout en tenant compte des résultats académiques tels que CCM [59]. Pour illustrer notre état de l'art sur l'adaptation de composants logiciels et d'applications conçues à base de composants, nous avons étudié un panel représentatif de ces différentes catégories de modèle de composants. L'ensemble des modèles de ce panel est présenté dans l'annexe C.

1.3.2.2 Composants, variabilité, adaptation et réutilisation

L'objectif de l'ingénierie des composants logiciels est de donner la possibilité aux concepteurs d'applications (ainsi qu'aux personnes chargés de la maintenance d'applications existantes) de manipuler des entités logicielles de haut niveau (*i.e.* notamment de plus haut niveau par rapport aux objets où la granularité est assez faible). Cette approche permet donc de gagner en réutilisabilité à grande échelle.

Le paradigme de « composants logiciels » introduits de nouveaux points de variabilité. A titre d'exemple, nous pouvons citer les points suivants :

- *la variabilité pour le comportement*
  Cette variabilité se manifeste au travers de deux points qui sont les suivants :

  – *la variabilité des connexions*
    Cette variabilité exploite la propriété de composabilité des composants. En effet, l'intérêt principal des composants logiciels réside dans le fait qu'il soit possible de les utiliser comme des briques de base configurables dans le but de construire des applications par composition. De ce fait, il est possible de changer facilement la configuration et les liens entre les composants et ainsi de pouvoir modifier le comportement de l'application ;

  – *la variabilité des traitements*
    Cette variabilité exploite les propriétés de modularité et d'autonomie des composants. En effet, les traitements sont regroupés dans des entités logicielles ; chacune étant destinée à répondre à un besoin spécifique. De ce fait, il est possible de changer facilement le comportement d'une application en modifiant ou en remplaçant certains de ses composants. La variabilité des traitements permet ainsi de pouvoir changer les traitements proposés ou la manière dont ils sont réalisés.

- *la variabilité du déploiement*
  Cette variabilité exploite la propriété d'autonomie des composants. En fait, un composant est considéré comme une unité de déploiement autonome et auto-descriptive. Les composants logiciels se proposent d'améliorer la description de codes à savoir donner une description la plus précise possible de l'utilisation de chaque composant (et de chacun des services qu'il propose ou requiert). Pour cela, il a été introduit la notion de contrat [25] de manière à contrôler le comportement des services qu'il propose. Grâce à ces descriptions, la réutilisation de composants est facilitée ;

- *la variabilité de structure*
  Cette variabilité exploite également les propriétés de composabilité et de modularité des composants logiciels. La structure d'une application conçue à base de composants peut être modifiée facilement par le remplacement de composants ou la reconfiguration des connexions entre ces composants. De plus, certains modèles de composants dit « hiérarchiques » offrent la possibilité de modifier la structure interne d'un composant (*i.e.* modification de ses sous-composants, des connexions entre ses sous-composants, etc.) ;

- *la variabilité de code*
  La variabilité de code exploite le concept d'architectures logicielles [101]. En fait, les architectures logicielles offrent des mécanismes permettant de décrire des applications ou des composants à de hauts niveaux d'abstraction. Cette variabilité dite « par l'abstraction » donne alors la possibilité de générer différentes implémentations suivant les conditions d'utilisation et les plates-formes de déploiement utilisées.

Tous ces points de variabilité font des composants logiciels, des entités facilement adaptables. De ce fait, leur capacité à être réutilisés en est accrue.

1.3.2.3 Étude des différences entre une approche d'adaptation d'applications classiques et une approche d'adaptation d'applications à base de composants

Différences des approches L'adaptation d'une application conçue à base de composants logiciels présente des spécificités liées au paradigme composant. Parmi ces spécificités, nous pouvons citer, à titre d'exemple :

- *l'adaptation sur deux niveaux*
  Dans le domaine des composants logiciels, un processus d'adaptation peut agir sur deux niveaux : l'adaptation au niveau de l'application en elle-même (par exemple, par la reconfiguration des connexions entre les éléments qui la composent) et l'adaptation au niveau des composants et/ou de leurs interactions qu'elle contient (par exemple, par le paramétrage des attributs des composants en fonction du contexte). Ainsi, au travers de l'adaptation, l'on peut parler de la création d'un nouveau cycle de développement pour une application. Ce cycle nécessite la vérification des effets des modifications sur ces deux niveaux d'adaptation. Cette vérification doit être réalisée aussi bien au niveau local au composant (*i.e.* plusieurs adaptations sur une même fonctionnalité d'un composant) qu'au niveau global à l'application (*i.e.* adaptation sur les réseaux de composants interconnectés). Les problèmes rencontrés pendant une phase d'adaptation sont la création de boucles infinies (*i.e.* cycles) et la présence de choix (*i.e.* points non déterministes) [98]. L'application doit être capable de résoudre ces problèmes car l'adaptation ne doit pas entraîner d'erreur provoquant une interruption brutale de son exécution ;

- *l'adaptation de composants existants*
  Par ailleurs, l'un des principaux verrous de la programmation orientée composants réside dans le fait que l'on travaille généralement sur des composants logiciels qui ont été conçus et développés par des personnes différentes de celles qui créent les applications. De ce fait, les composants ne correspondent pas toujours aux réels besoins liés à leur utilisation. Par exemple, un composant peut avoir été conçu pour répondre à un besoin trop spécifique ou trop générique, si bien que certains de ses services se révèlent inutilisables ou non utilisés après la création d'une nouvelle application, bien que leur déploiement soit nécessaire car ils ne peuvent être dissociés de l'application. Cette situation peut se révéler très problématique dans des environnements à ressources contraintes dans lesquels le coût de déploiement (en espace mémoire, etc.) de chaque service doit être pris en considération ; il doit généralement être le plus faible possible. Ainsi, la réutilisation à grande échelle engendre de nouveaux problèmes qui peuvent être résolus par l'adaptation des composants ou des architectures logicielles mis en jeu.

Différences de processus Les étapes d'un processus d'adaptation d'une application à base de composants sont identiques à celles relatives à un processus d'adaptation classique : détection - prise de décisions - réalisation. Seules les techniques de réalisation de l'adaptation (troisième phase du processus) sont spécifiques aux composants logiciels (voir Figure 1.5). Ces techniques que nous présenterons dans la section 1.4 peuvent requérir d'une part, des propriétés spécifiques à l'application source telles que la disponibilité du code source, l'introspection et l'intercession fournies par le modèle de composants utilisé, etc. et d'autre part, des opérations de base pour l'adaptation fournies généralement par l'infrastructure logicielle de déploiement de l'application (*i.e.* modèle de composants utilisé et sa plate-forme d'implémentation).

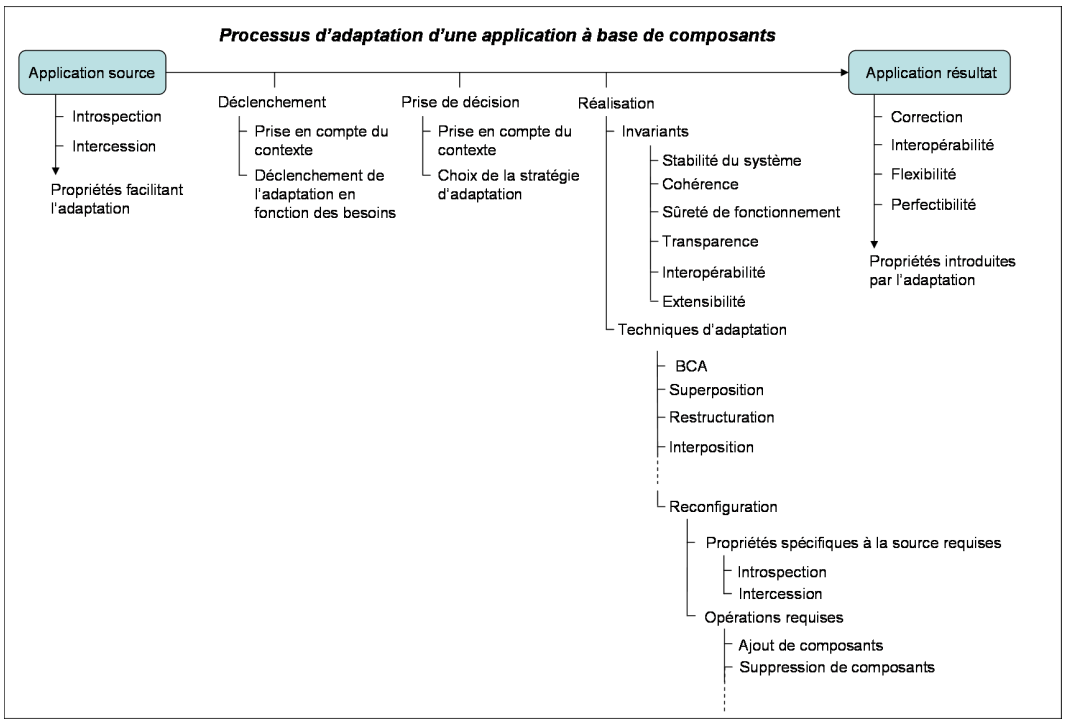

Figure 1.5 – Processus d'adaptation d'une application conçue à base de composants

1.3.2.4 Composants adaptables ou auto-adaptatifs

Un composant adaptable est un composant qui possède la capacité d'être adapté soit par l'intervention d'une entité[1] extérieure à celui-ci ; soit par le composant lui-même qui peut modifier son comportement ou sa structure en fonction des éléments qu'il connaît. Dans le second cas, on parle alors de composants auto-adaptatifs.

Si l'on se base sur les conditions requises pour qu'un système ou une application soit adaptable, citées précédemment, un composant logiciel satisfait par définition [123] les deux premières propriétés à savoir la modularité et la composabilité. L'introspection et l'intercession sont deux propriétés, quant à elles, dépendantes du modèle de composants utilisé et de son implémentation. Cependant, la plupart des modèles proposés dans la littérature proposent au minimum des fonctionnalités d'introspection. Dans le panel de modèle de composants que nous avons établi dans l'annexe C, seul le modèle JavaBean [62] ne dispose pas de la propriété d'intercession. Si l'on prend l'exemple du modèle de composants Fractal [39], ces deux propriétés sont garanties par la présence d'interfaces de contrôle, fournies par chaque composant, qui permettent d'observer sa structure et son comportement ainsi que d'agir sur lui-même (par exemple, chaque composant Fractal est doté d'une interface de contrôle appelée *BindingController* permet de réaliser de l'introspection sur les connexions entre les composants et de les modifier).

Par ailleurs, dans de nombreux environnements tels que les environnements ubiquitaires ou mobiles, le contexte d'exécution d'une application est en perpétuelle évolution. De ce fait, la continuité de service peut être remise en question à tout moment lors de l'exécution d'une application. Par exemple, dans le

[1] Une entité peut-être une personne physique, un outil dédié, un *framework*, un autre composant, etc.

cadre d'environnement à ressources limitées, une diminution de la valeur d'une propriété de l'une des machines de déploiement de l'application (par exemple, diminution de la taille mémoire disponible sur la machine de l'utilisateur) peut avoir de lourdes conséquences sur le fonctionnement de l'application (par exemple, la continuité de service peut être interrompue ou bien la qualité de service peut être altérée). Ainsi, une application doit être capable de tenir compte des évolutions de son contexte d'exécution afin de s'adapter aux nouvelles situations rencontrées pour lui garantir une continuité de service et maintenir une certaine qualité de service. Dans ce cas, un composant doit être auto-adaptatif.

Un composant auto-adaptatif est un composant capable d'adapter automatiquement sa structure et son comportement en réponse à des variations de son environnement d'exécution ayant des conséquences sur son comportement.

1.4 Étude des approches existantes d'adaptation dans l'ingénierie des composants logiciels

Afin de montrer les avantages, les insuffisances et les différences des approches existantes par rapport à leur capacité de prendre en compte l'adaptation, nous avons dans un premier temps étudié les possibilités d'adaptation au niveau des infrastructures logicielles de déploiement disponibles. Ensuite, nous avons réalisé une étude comparative des approches d'adaptation qui ont été proposées dans la littérature de l'ingénierie des composants logiciels.

1.4.1 Prise en charge de l'adaptation au niveau de l'infrastructure logicielle de déploiement

L'infrastructure logicielle de déploiement d'une application fait référence au modèle de composants utilisé et la plate-forme d'implémentation de ce modèle. Comme nous l'avons évoqué précédemment, son aptitude à permettre l'adaptation dépend de ses propriétés d'introspection et surtout d'intercession.

1.4.1.1 Disponibilité des mécanismes de base support à l'adaptation dans les infrastructures à base de composants

Nous avons répertorié, ci-dessous, les opérations d'adaptation d'une application conçue à base de composants logiciels et nous avons recherché leur disponibilité[2] dans le panel de modèle de composants présenté dans l'annexe C.

1. *Adaptation au niveau des interfaces (voir Tableau 1.1)* :
   - ajouter un point d'entrée à une interface ($Op_{Interface}1$)
   - modifier un point d'entrée à une interface ($Op_{Interface}2$)
   - masquer[3]/supprimer un point d'entrée ($Op_{Interface}3$)

[2] $+$ indique qu'il est possible de réaliser l'opération concernée dans le modèle spécifié. $-$ indique l'indisponibilité de l'opération concernée dans le modèle spécifié. α/β : α dans le cas d'une adaptation statique et β dans le cas d'une adaptation dynamique.

[3] « Masquer » signifie rendre inopérant un point d'entrée alors que supprimer est une fonction irréversible qui efface le point d'entrée de l'interface.

Opérations	($Op_{Interface}1$)	($Op_{Interface}2$)	($Op_{Interface}3$)
JavaBean	$-/-$	$-/-$	$-/-$
EJB	$+/-$	$+/-$	$+/-$
CCM	$+/-$	$+/-$	$+/-$
COM et COM+	$+/+$	$+/+$	$+/-$
Fractal	$+/-$	$+/-$	$+/-$

Table 1.1 – Disponibilité[3] des opérations d'adaptation au niveau des interfaces dans les modèles de composants de notre panel

2. *Adaptation au niveau des ports (voir Tableau 1.2)* :
   - ajouter une interface à un port ($Op_{Port}1$)
   - remplacer une interface par une ou plusieurs autres ($Op_{Port}2$)
   - masquer/supprimer une interface dans un port ($Op_{Port}3$)
   - restructurer les interfaces d'un port ($Op_{Port}4$)

Opérations	($Op_{Port}1$)	($Op_{Port}2$)	($Op_{Port}3$)	($Op_{Port}4$)
JavaBean	$-/-$	$-/-$	$-/-$	$-/-$
EJB	$-/-$	$-/-$	$-/-$	$-/-$
CCM	$+/-$	$+/-$	$+/-$	$+/-$
COM et COM+	$-/-$	$-/-$	$-/-$	$-/-$
Fractal	$-/-$	$-/-$	$-/-$	$-/-$

Table 1.2 – Disponibilité[3] des opérations d'adaptation au niveau des ports dans les modèles de composants de notre panel

3. *Adaptation au niveau d'un composant (voir Tableaux 1.3 et 1.4)* :
   - ajouter un port/une interface ($Op_{Comp}1$)
   - masquer/supprimer un port/une interface ($Op_{Comp}2$)
   - restructurer les ports/les interfaces ($Op_{Comp}3$)
   - ajouter un sous-composant (dans le cas d'un modèle de composants hiérarchiques) ($Op_{Comp}4$)
   - masquer/supprimer un sous-composant (dans le cas d'un modèle de composants hiérarchiques) ($Op_{Comp}5$)
   - fusionner deux sous-composants (*i.e.* remplacer deux sous-composants par un seul) ($Op_{Comp}6$)
   - fragmenter un sous-composant (*i.e.* remplacer un sous-composant par plusieurs) ($Op_{Comp}7$)
   - configurer le composant (*i.e.* modifier ses attributs) ($Op_{Comp}8$)
   - configurer un sous-composant ($Op_{Comp}9$)
   - associer une nouvelle implémentation ($Op_{Comp}10$)

4. *Adaptation au niveau des connexions entre composants (voir Tableau 1.5)* :
   - configurer des interconnexions entre les composants ($Op_{Connexion}1$)
   - rediriger des messages échangés entre les composants ($Op_{Connexion}2$)
   - insérer des pré-conditions et des post-conditions ($Op_{Connexion}3$)

Opérations	($Op_{Comp}1$)	($Op_{Comp}2$)	($Op_{Comp}3$)	($Op_{Comp}4$)	($Op_{Comp}5$)
JavaBean	$-/-$	$-/-$	$-/-$	$-/-$	$-/-$
EJB	$+/-$	$+/-$	$+/-$	$-/-$	$-/-$
CCM	$+/-$	$+/-$	$+/-$	$-/-$	$-/-$
COM et COM+	$+/-$	$+/-$	$+/-$	$-/-$	$-/-$
Fractal	$+/-$	$+/-$	$+/-$	$+/+$	$+/+$

Table 1.3 – Disponibilité[3] des opérations d'adaptation au niveau d'un composant dans les modèles de composants de notre panel (a)

Opérations	($Op_{Comp}6$)	($Op_{Comp}7$)	($Op_{Comp}8$)	($Op_{Comp}9$)	($Op_{Comp}10$)
JavaBean	$-/-$	$-/-$	$+/+$	$-/-$	$+/-$
EJB	$-/-$	$-/-$	$+/+$	$-/-$	$+/-$
CCM	$-/-$	$-/-$	$+/+$	$-/-$	$+/-$
COM et COM+	$-/-$	$-/-$	$+/+$	$-/-$	$+/+$
Fractal	$+/-$	$+/-$	$+/+$	$+/+$	$+/-$

Table 1.4 – Disponibilité[3] des opérations d'adaptation au niveau d'un composant dans les modèles de composants de notre panel (b)

Opérations	($Op_{Connexion}1$)	($Op_{Connexion}2$)	($Op_{Connexion}3$)
JavaBean	$-/-$	$+/-$	$+/-$
EJB	$-/-$	$+/-$	$+/+$
CCM	$-/-$	$+/-$	$+/-$
COM et COM+	$-/-$	$+/-$	$+/-$
Fractal	$-/-$	$+/-$	$+/-$

Table 1.5 – Disponibilité[3] des opérations d'adaptation au niveau des connexions entre composants dans les modèles de composants de notre panel

5. *Adaptation au niveau de l'architecture de l'application (voir Tableau 1.6)* :
   - ajouter un composant ($Op_{Archi}1$)
   - masquer/supprimer un composant existant ($Op_{Archi}2$)
   - modifier les interconnexions entre les composants ($Op_{Archi}3$)
   - fusionner deux composants ($Op_{Archi}4$)
   - fragmenter un composant ($Op_{Archi}5$)
   - transférer un composant dans un autre conteneur ($Op_{Archi}6$) (*i.e.* migration de composants)

Opérations	($Op_{Archi}1$)	($Op_{Archi}2$)	($Op_{Archi}3$)	($Op_{Archi}4$)	($Op_{Archi}5$)	($Op_{Archi}6$)
JavaBean	$+/+$	$+/+$	$-/-$	$+/-$	$+/-$	$+/-$
EJB	$+/+$	$+/+$	$+/+$	$+/-$	$+/-$	$+/-$
CCM	$+/+$	$+/+$	$+/+$	$+/-$	$+/-$	$+/-$
COM et COM+	$+/+$	$+/+$	$+/+$	$+/-$	$+/-$	$+/-$
Fractal	$+/+$	$+/+$	$+/+$	$+/-$	$+/-$	$+/-$

Table 1.6 – Disponibilité[3] des opérations d'adaptation au niveau de l'architecture de l'application dans les modèles de composants de notre panel

1.4.1.2 L'adaptation par l'utilisation des opérations offertes par l'infrastructure

Les opérations d'ajout (de points d'entrées dans une interface, d'interfaces dans un port, de ports/interfaces dans un composant ou bien de composants dans une application) sont généralement utilisées pour faire évoluer un composant ou une application de par l'ajout de nouvelles fonctionnalités (*i.e.* adaptation évolutive).

Concernant les opérations de modification et de remplacement (d'interfaces, de ports, de composants, ou d'architectures), elles permettent également de faire évoluer le comportement d'un composant ou d'une application mais aussi de corriger d'éventuels défauts détectés soit par l'administrateur de l'application soit par l'application elle-même (*i.e.* adaptation corrective).

Les opérations de masquage/suppression sont, quant à elles, utilisées dans le cadre de ressources limitées afin de réduire les besoins en ressources d'un composant ou d'une application, ou bien rendre inactives des interfaces susceptibles de consommer trop de ressources (ou bien susceptibles de ne pouvoir être exécutées correctement).

Les opérations d'ajout, de modification et de suppression de point d'entrées, d'interfaces et de ports peuvent également être utilisées pour assurer l'interopérabilité des composants au moment de leur assemblage. Les opérations de restructuration des ports/interfaces d'un composant partage le même objectif. Elles permettent à deux composants - l'un fournissant les services requis par l'autre - d'être assemblés même si leurs ports/interfaces ne sont pas initialement structurés de la même manière.

Les autres opérations (fragmentation de composants et fusion de composants) peuvent être utilisées pour modifier la structure d'un composant afin de l'adapter à son environnement d'exécution (*i.e.* adaptation à l'architecture matérielle et aux besoins de l'utilisateur).

Les opérations de configuration (de connexions ou de composants) sont utilisées afin d'adapter le comportement d'une application en fonction de contexte d'exécution (*i.e.* adaptation au contexte d'utilisation).

1.4.1.3 Mise en œuvre de ces opérations

De manière statique, la plupart des opérations d'adaptation décrites précédemment sont réalisées par génération ou par modification de code, et ce, pour la plupart de manière manuelle ; ce qui rend ces opérations réalisables dans la majorité des modèles recensés. Cependant, certaines opérations qui ne nécessitent pas l'intégration de nouveau code métier spécifique à l'application, telles que la fragmentation ou la fusion de composants peuvent être entièrement automatisées. Or, actuellement, aucune approche ne propose de les automatiser. Ainsi, dans le cadre de cette thèse nous nous sommes attachés à proposer une approche permettant d'automatiser ces opérations d'adaptation (voir Chapitre 2).

De manière dynamique, ces opérations sont étroitement liées au modèle de composants utilisé qui peut ou non les proposer. En fait, les modèles permettant de réaliser ce type d'opération doivent être dotés de propriétés d'introspection et d'intercession comme c'est le cas pour les modèles Ejb, CCM, COM et Fractal. Cependant, les opérations proposées par les modèles sont, pour la plupart, basiques. En fait, ils proposent généralement uniquement des opérations standard tels que l'ajout de composants, la suppression de composants, la modification de liaison entre les composants, etc. Les opérations plus complexes telles que la fragmentation ou la fusion dynamique de composants ne sont pas offertes par les modèles existants. De ce fait, nous proposons dans cette thèse une approche permettant de fragmenter dynamiquement un composant (voir Chapitre 4).

1.4.2 Un canevas pour l'étude de l'adaptation au niveau des applications à base de composants

Pour étudier l'adaptation de composants logiciels et d'applications conçues à base de composants, nous avons défini un certain nombre de critères de comparaison permettant de classifier les travaux existants, à savoir les raisons de l'adaptation (pourquoi adapter ?), sa cible (sur quoi porte l'adaptation ?), son moment d'exécution (quand est réalisée l'adaptation ?), son niveau d'automatisation (qui réalise l'adaptation ?), son environnement d'exécution (dans quel environnement est réalisée l'adaptation ?), et enfin la technique utilisée pour sa dynamicité et sa mise en œuvre (comment est réalisée l'adaptation ?).

1.4.2.1 Les raisons de l'adaptation comme critère de classification

L'adaptation d'un composant logiciel ou d'une application conçue à base de composants a trois objectifs majeurs qui sont : l'accroissement de la flexibilité (*i.e.* le composant ou l'application devient alors capable d'être exécuté dans n'importe quel environnement et peuvent s'adapter afin de garantir sa continuité de service) [3], l'augmentation ou le maintien des performances (*i.e.* augmentation ou préservation de la qualité de service) [75] et enfin l'évolution des fonctionnalités (*i.e.* ajout ou modification de services fournis par les composants ou l'application) [44, 75].

Tout d'abord, un composant ou une application peut être adapté de manière à accroître sa flexibilité. Dans ce cas, nous pouvons distinguer trois types d'adaptation :

- *l'adaptation adaptative*
  L'adaptation adaptative telle qu'elle est définie par Ketfi *et al* dans [75] consiste à faire modifier le comportement ou la structure d'un composant ou d'un assemblage de composants en fonction de son environnement d'exécution. Par exemple, l'adaptation adaptative peut consister à configurer les propriétés d'un composant en fonction de son contexte d'exécution à savoir le profil de l'utilisateur, son environnement proche, etc. ;

- *l'adaptation pour la flexibilité*
  Adapter un composant ou une application à base de composants pour accroître sa flexibilité consiste à le rendre capable de s'exécuter sur n'importe quel système et d'exploiter au mieux ses capacités. Par exemple, le déploiement d'un composant peut être qualifié de flexible lorsque la machine de déploiement ne dispose pas de ressources nécessaires mais que le déploiement peut quand même être réalisé (par exemple, en distribuant certains sous-composants en fonction du contexte ou en ne chargeant que les services susceptibles d'être utilisés) ;

- *l'adaptation pour l'interopérabilité*
  Ce type d'adaptation est généralement utilisé pour la création de nouvelles applications ou bien lors de la reconfiguration d'une architecture logicielle. Elle réside dans l'utilisation d'outils ou de mécanismes permettant d'assurer la compatibilité entre les services, interfaces ou ports requis et fournis par les composants. Par exemple, si un composant fournit les services requis par un autre composant mais que la structure de leur port ne permet pas de les assembler, une adaptation par restructuration est alors nécessaire afin de rendre compatible les deux composants. Cette adaptation permet donc d'assurer l'interopérabilité entre ces deux composants ;

Par ailleurs, l'adaptation peut être réalisée dans l'optique de maintenir les performances de l'application ou de les augmenter. Dans ce cas, nous distinguons trois types d'adaptation :

- *l'adaptation perfective*
  Ce type d'adaptation est utilisé pour accroître les performances d'un composant ou d'un assemblage de composant. Ainsi, par exemple, si un composant est jugé inefficace vis-à-vis de ses performances, il peut être remplacé par un autre de manière à augmenter l'efficacité de l'application (*i.e.* éviter une dégradation des performances de l'application). Par exemple, diminuer la complexité d'un algorithme permet d'augmenter les performances. L'objectif de l'adaptation perfective est d'optimiser le comportement des composants ;

- *l'adaptation correctionnelle*
  Son objectif est de corriger les erreurs de fonctionnement d'une entité au sein de l'application : si l'exécution ne se déroule pas correctement, il faut identifier le ou les éléments (composants, connecteurs, etc.) qui posent problèmes et les remplacer par d'autres supposés ayant un fonctionnement correct et fournissant les mêmes fonctionnalités que les composants défaillants ;

- *l'adaptation pour l'accroissement de la réutilisabilité*
  L'objectif d'une telle adaptation est d'accroître la réutilisabilité d'un composant ou d'un assemblage de composants. Par exemple, un composant peut être utilisé dans des contextes différents de ceux envisagés durant sa conception. De ce fait, pour être réutilisé, ce composant devra être adapté afin qu'il réponde à de nouveaux besoins.

Le dernier besoin de l'adaptation repose sur l'évolution des fonctionnalités de l'application que l'on veut adapter [44, 75]. Cette stratégie appelée *adaptation évolutive* consiste à ajouter de nouveaux composants ou services à l'application de manière à proposer à l'utilisateur de nouvelles fonctionnalités (*i.e.* possibilité d'ajouts de nouveaux composants ou étendre les composants existants). Elle est liée aux besoins de l'utilisateur. Cependant, les fonctionnalités existantes (*i.e.* services fournis par l'application avant son adaptation) ne sont pas modifiées.

1.4.2.2 La cible de l'adaptation comme critère de classification

La première étape d'un processus d'adaptation d'une application consiste généralement à déterminer la cible de l'adaptation (*i.e.* sur quoi porte l'adaptation ? que doit-t-on adapter ?).

La facette cible de l'adaptation Tout d'abord, l'adaptation peut porter sur le comportement ou la structure de l'entité cible ; la cible pouvant être n'importe quelle entité définie dans l'application à adapter. Par exemple, la cible peut être les services métiers (*i.e.* services fournis par l'application) comme sur les services techniques relatifs à la persistance, aux transactions ou à la sécurité, qui sont des services généralement fournis par la plate-forme de déploiement de l'application.

L'entité cible de l'adaptation L'adaptation d'une application peut agir sur différents niveaux de granularité (*i.e.* l'application est composée d'éléments mis en relation par des liaisons) :

- *au niveau des éléments qui composent l'application : les composants*
  Adapter un élément d'une application peut consister par exemple à corriger son comportement si ce dernier n'est pas conforme à la situation dans laquelle l'application se trouve et s'il présente des risques d'erreur. Les éléments à adapter peuvent être vus comme des boîtes blanches (*i.e.* un élément est vu comme une boîte blanche lorsque sa spécification interne est visible par le reste des composants de l'application), des boîtes grises (*i.e.* un élément est vu comme une boîte grise lorsque seulement une partie de sa spécification interne est visible par le reste des composants de l'application, l'autre n'est pas visible) ou bien des boîtes noires (*i.e.* un élément est vu comme une boîte noire lorsque sa spécification interne n'est pas visible par le reste des composants de l'application). Il peut s'agir d'un simple paramétrage comme d'une modification complète de l'élément en question. Ces opérations permettent d'optimiser l'application au niveau local à l'élément que l'on doit adapter ;

- *au niveau des liaisons entre les éléments qui composent l'application : les connecteurs*
  Adapter les liaisons entre les différents éléments d'une application peut consister à modifier leur type (synchrone ou asynchrone) ou bien à analyser et à traiter les messages échangés de manière à effectuer des post-traitements (*i.e.* après envoi du message), des prétraitements (*i.e.* avant réception du message) ou encore des redirections (*i.e.* changement de cible) de manière à optimiser le comportement général de l'application ;

- *au niveau de l'architecture de l'application*
  L'adaptation peut engendrer une restructuration complète de l'architecture de l'application. Cette opération consiste en fait à réorganiser l'assemblage des différents éléments qui composent l'application. Par exemple, pour des raisons d'optimisation de comportement, l'architecture d'une application peut être modifiée de manière à pouvoir exécuter certains services en parallèle. Ces opérations ont pour objectif d'optimiser le comportement global de l'application.

Il est à noter que la facette de l'adaptation est étroitement liée à l'entité cible. En fait, les entités cibles d'une adaptation sont différentes que l'on choisisse d'adapter la structure ou le comportement (voir Figure 1.6). Par exemple, l'adaptation du comportement agit essentiellement au niveau des services (fonctionnels ou non) alors que l'adaptation de la structure est réalisée sur les entités qui forment un composant (interfaces, ports, composants, etc.).

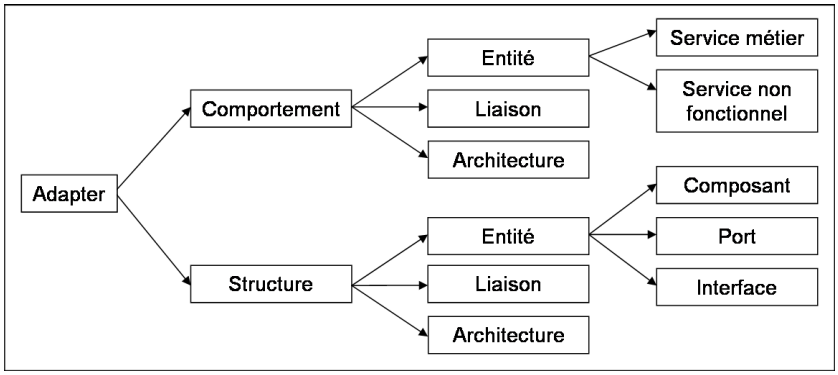

Figure 1.6 – Les différentes cibles de l'adaptation

1.4.2.3 Le moment de l'adaptation comme critère de classification

Une phase d'adaptation peut être provoquée par la découverte ou l'évolution du contexte d'exécution d'une application. Elle peut être également déclenchée par l'intervention d'une personne physique telle que l'administrateur de l'application ou l'utilisateur, à tout moment dans le cycle de vie d'une application. Ainsi, on peut distinguer différents moments clefs au cours desquels une application peut être adaptée :

- *avant le déploiement*
  L'adaptation de l'application peut être réalisée *au moment de sa conception et de son développement*. En effet, si le ou les concepteurs possèdent des informations sur le contexte d'utilisation (*i.e.* contexte d'exécution), il peut adapter l'application en fonction de ces données. Plus généralement, pendant cette étape du cycle de vie de l'application, les concepteurs construisent l'application de manière à autoriser des adaptations ultérieures. Ils ont pour tâche de prévoir les possibilités d'adaptation afin d'agir en conséquence.
  De plus, l'adaptation de l'application peut également être effectuée *avant la compilation de son code source*. L'entité physique ou logicielle chargée d'adapter l'application peut alors modifier le code de manière à prendre en compte les données connues sur le contexte d'exécution.
  Enfin, l'adaptation peut être réalisée *après la compilation de son code source*. Deux cas sont possibles : soit le code source est disponible ; dans ce cas l'adaptateur pourra manipuler ce dernier (*i.e.* techniques de refactorisation et de transformation de code), puis recompiler les nouveaux fichiers sources de l'application adaptée. Soit, le code source de l'application n'est pas disponible ; dans ce cas, il faut appliquer des méthodes de « *reengineering* » ;

- *pendant le déploiement*
  Le déploiement est une phase importante d'un cycle de vie d'une application. Le déploiement comporte un ensemble de tâches qui doivent être réalisées avant l'exécution de l'application afin d'assurer son fonctionnement. Parmi ces activités, nous pouvons citer la copie des fichiers binaires sur la machine sur laquelle va être exécutée l'application, la mise en place des connexions entre les différents éléments qui composent l'application, etc.
  L'adaptation de l'application au moment de son déploiement revêt une importance cruciale car à cet instant précis de son cycle de vie, l'application découvre l'environnement dans lequel elle

va être exécutée. De nombreux travaux sont consacrés à l'acquisition et à la prise en compte du contexte notamment au moment du déploiement. Ces derniers ont pour objectif majeur de concevoir des outils permettant à l'application de prendre en compte son contexte afin de s'y adapter ;

- *pendant l'exécution*
  Le contexte d'exécution d'une application peut changer en permanence et c'est notamment le cas pour des applications destinées à fonctionner dans des environnements ubiquitaires et mobiles. Ainsi, l'adaptation pendant l'exécution peut se révéler obligatoire pour certains types d'application. Comme pour le déploiement, il existe de nombreux outils permettant à l'application d'acquérir en continu (*i.e.* tout au long de son exécution) des connaissances sur son environnement ainsi que sur son comportement.

1.4.2.4 La dynamicité de l'adaptation comme critère de classification

L'adaptation d'une application peut être réalisée à la volée ou bien nécessiter l'arrêt de l'application. On parle *d'adaptation dynamique* lorsque celle-ci est réalisée sans arrêter l'application et *d'adaptation statique* lorsqu'il est nécessaire de stopper l'application pour procéder à l'adaptation :

- *l'adaptation statique*
  Les modifications opérées pour réaliser l'adaptation sont effectuées lorsque l'application est à l'arrêt. Généralement, elles sont effectuées avant la phase de déploiement de l'application sur sa plate-forme d'exécution. Après son déploiement, il est également possible de réaliser une phase d'adaptation statique : pour cela, il suffit de stopper l'exécution de l'application, puis opérer les transformations adéquates et enfin redémarrer l'application ;

- *l'adaptation dynamique*
  Adapter dynamiquement une application consiste à introduire des modifications pendant l'exécution d'une application. Cette activité est également appelée reconfiguration dynamique. L'adaptation est donc réalisée après la phase de déploiement dans le cycle de vie de l'application. Cette approche ne nécessite pas l'arrêt de l'application contrairement à l'approche statique. Pour qu'une application soit adaptable dynamiquement, il faut que les éléments qui la composent et les modèles utilisés dans sa conception supportent la dynamicité.

1.4.2.5 Le niveau d'automatisation de l'adaptation comme critère de classification

Ce critère fait référence au degré d'automatisation d'un processus d'adaptation à savoir la place de l'utilisateur de l'adaptation dans un processus d'adaptation ; l'utilisateur de l'adaptation pouvant être le concepteur/programmeur de l'application, l'administrateur de l'application ou bien l'utilisateur de l'application. Une adaptation peut être désignée comme manuelle (*i.e.* le processus d'adaptation est entièrement réalisé par l'utilisateur de l'adaptation), semi-automatique (*i.e.* le processus d'adaptation nécessite l'intervention de l'utilisateur de l'adaptation) ou automatique (*i.e.* le processus d'adaptation ne nécessite pas d'intervention de l'utilisateur de l'adaptation).

Il dépend notamment des acteurs de l'adaptation. En fait, un acteur de l'adaptation est une entité physique ou logicielle capable de démarrer, de modifier, de stopper, d'annuler ou bien de superviser une phase d'adaptation. Elle peut intervenir à différents niveaux dans le cycle de vie d'une application (voir Figure 1.7). Nous distinguons cinq acteurs de l'adaptation :

- *l'utilisateur*
  Dans certains cas, l'utilisateur peut être un acteur de l'adaptation ; notamment dès lors qu'il est inclus dans un processus décisionnel (déclenchement d'une phase d'adaptation, spécification du résultat de l'adaptation, etc.) ;

- *le concepteur / programmeur*
  Il est chargé de concevoir la manière dont l'application va s'adapter dans un contexte particulier. En fait, il doit définir les règles et les mécanismes qui vont permettre à l'application de s'adapter. De plus, il a la possibilité de figer certaines étapes de l'adaptation (*i.e.* de par la création des règles). Le problème, généralement rencontré dans la création ou la maintenance d'application, réside dans le fait que dans la majorité des cas, ce n'est pas la même personne qui a écrit le code de l'application ou du composant et celle qui souhaite adapter ce code, d'où une certaine difficulté pour réaliser l'adaptation ;

- *l'administrateur*
  Il s'agit de la personne qui a la possibilité de déclencher une phase d'adaptation de l'application. Cette opération est généralement réalisée par l'appel à un ou plusieurs services fournis par l'application elle-même et dédiés à cet effet. Par exemple, il peut détecter une anomalie et prendre les mesures nécessaires pour y remédier en adaptant le code de l'application ou bien en démarrant une procédure d'adaptation ;

- *un adaptateur*
  Un adaptateur est un outil chargé de réaliser toutes les tâches contenues dans un processus d'adaptation. Il peut être autonome ou bien contrôlé par l'un des autres acteurs de l'adaptation ;

- *l'application*
  L'application est l'acteur qui subit l'adaptation généralement déclenchée et réalisée par ceux cités précédemment. Par ailleurs, elle a la possibilité de s'auto-adapter par exemple par le déclenchement de capteurs ou d'évènements. Cette procédure passe obligatoirement par une phase d'acquisition et d'analyse de son contexte d'exécution. Ainsi, une application peut modifier son propre comportement en observant ses ressources ainsi que les changements de son environnement d'utilisation.

Une application est qualifiée d'adaptative lorsqu'elle fournit les moyens à un acteur externe (*i.e.* concepteur, administrateur, etc.) de réaliser des adaptations, alors qu'elle est qualifiée d'auto-adaptative lorsque l'adaptation est réalisée de façon interne et qu'elle est déclenchée par des variations de son contexte d'exécution.

Le niveau d'automatisation de l'adaptation dépend également de la profondeur (quels sont les étapes pour chaque travail) et de la pertinence de la décision (quels sont les éléments de décisions).

1.4.2.6 L'environnement d'exécution de l'application comme critère de classification

Les travaux existants se positionnent généralement dans le cadre d'environnement particulier de manière à répondre à une problématique spécifique. En fait, l'objectif et la réalisation de l'adaptation peuvent être très différents selon l'environnement d'exécution dans lequel elle est positionnée. Nous

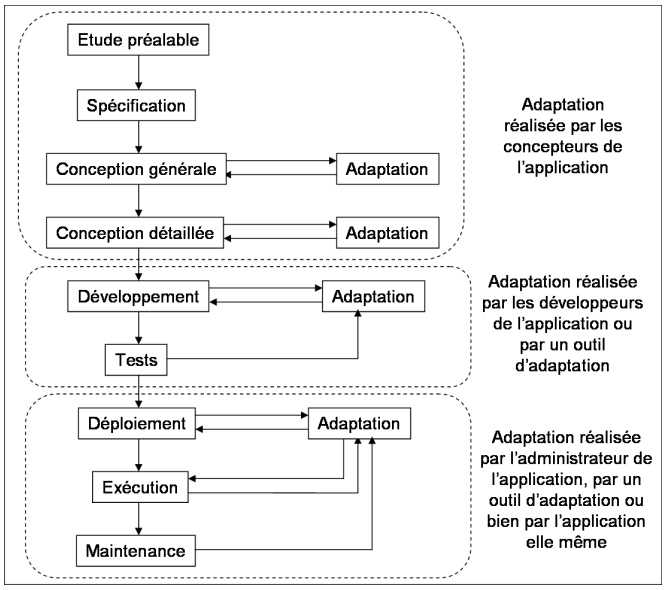

Figure 1.7 – Positionnement des acteurs de l'adaptation dans le cycle de vie d'une application

avons recensé quatre types d'environnements sur lesquels se sont focalisées les approches existantes ; chaque type d'environnement est doté de caractéristiques spécifiques sur lesquelles doit se baser l'adaptation :

- *les environnements mono-machine*
  Dans le cadre d'un environnement mono-machine, l'application est déployée et exécutée sur une seule machine (*i.e.* l'application utilise uniquement les services et les données fournies par la machine sur laquelle elle est installée). Dans ce cadre précis, l'adaptation peut consister par exemple à charger ou à décharger certaines parties d'un composant si les ressources nécessaires à son déploiement ne sont pas suffisantes pour permettre un déploiement intégral du composant. Dans ce cas, on parle d'adaptation au support ;

- *les systèmes distribués standard, les grilles de calculs et les clusters*
  Dans un environnement réparti, une application peut être déployée et exécutée sur plusieurs machines d'un réseau qui peut être filaire (*i.e.* réseaux physiques) ou sans fil (*Wifi*, *Bluetooth*, IR, etc.). Elle peut donc utiliser des données ou des services qui sont disponibles sur des sites (*i.e.* machines) différents par l'intermédiaire du réseau. Les principaux problèmes à résoudre lors de la création d'applications destinées à être exécutées dans ce type d'environnement sont la gestion des communications distantes (*i.e.* appel à des services distants par l'intermédiaire du réseau), la tolérance aux fautes (*i.e.* gestion des transactions, tolérance aux pannes, etc.), la disponibilité des ressources et services (*i.e.* éviter les problèmes de privation liés à la gestion de la cohérence et de la concurrence des transactions), la gestion des ressources distantes (*i.e.* gestion des caches, bases de données distribuées) et enfin la sécurité (*i.e.* authentification, transactions sécurisées, etc.).
  Une grille informatique ou *grid* est une infrastructure matérielle constituée d'un ensemble coordonné de ressources potentiellement partagées, distribuées, hétérogènes, externalisées et sans

administration centralisée.
Dans ce type d'environnement, l'adaptation est généralement utilisée afin d'améliorer les performances de l'application ou de certains composants. Ce résultat peut être obtenu en permettant l'exécution parallèle de certains morceaux de code en fonction du contexte ;

- *les environnements mobiles*
  L'informatique mobile (en anglais « *Mobile Computing* ») est un domaine qui concerne la fabrication et l'utilisation d'ordinateurs de taille réduite que l'on peut transporter avec soi dans ses déplacements et qui, le plus souvent, peuvent être reliés sans fil à un réseau ou à d'autres ordinateurs. Ainsi, la grande problématique des approches mettant en jeu ce type de réseau consiste à garantir à l'utilisateur qu'il puisse continuer d'accéder à l'information fournie par une infrastructure distribuée sans tenir compte de son emplacement.
  L'informatique mobile se focalise sur la mobilité du matériel, des données et du logiciel dans des applications informatiques et leur possible apparition et disparition (*i.e.* liée aux connexion/déconnexions des machines) durant toute la durée de vie de l'application.
  De ce fait, elle peut être considérée comme une classe spécialisée des systèmes répartis où quelques nœuds peuvent désengager des opérations distribuées communes, se déplacer librement dans l'espace physique et se reconnecter ultérieurement à un autre segment probablement différent d'un réseau informatique afin de reprendre des activités suspendues ;

- *les environnements ubiquitaires*
  Un environnement ubiquitaire (voir Section 1.5) correspond à un réseau ambiant se caractérisant par des entités mobiles communicantes et parfois de très petite taille et possédant des caractéristiques techniques très différentes les unes des autres allant du téléphone portable aux capacités réduites (*i.e.* puissance de calcul faible, batterie à faible autonomie, taille de l'écran limitée, capacité de stockage faible, etc.) à l'ordinateur de bureau. L'informatique ubiquitaire consiste à faire fonctionner une application dans n'importe quel environnement physique tout en réalisant l'adaptation de manière totalement transparente aux yeux de l'utilisateur [132].

1.4.2.7 Les techniques de mise en œuvre des approches d'adaptation comme critère de classification

Les techniques utilisées pour mettre en œuvre l'adaptation d'une application conçue à base de composants logiciels varient en fonction de nombreux paramètres tels que la cible de l'adaptation, le moment de l'adaptation, etc.

Présentation des techniques existantes Afin de présenter les techniques d'adaptation existantes, nous les avons classifiées en fonction de leur cible à savoir les composants, les connecteurs ou l'architecture logicielle.

Adaptation des composants

1. *BCA (Binary Component Adaptation)*
   BCA [73] est une technique d'adaptation qui agit sur le code binaire des composants. Elle ne nécessite pas le code source. Le système BCA a été implémenté en Java. L'adaptation peut être réalisée à tout moment dans le cycle de vie du composant. Pour effectuer l'adaptation d'un composant après le déploiement, l'application doit construire un fichier de spécification contenant les

éléments que l'on souhaite modifier tels que les méthodes, interfaces, champs, etc. Un compilateur fournit alors un fichier binaire contenant le code binaire modifié qui est nécessaire pour que le composant soit adapté. Ensuite, les autres classes qui font références au composant adapté doivent être recompilées. L'assemblage des fichiers de type *.class* peut alors être effectué puis exécuté.

2. *La paramétrisation*
   Cette technique d'adaptation réside dans la modification des paramètres d'exécutions d'une instance de composant (*i.e.* changer les valeurs des propriétés des composants). Elle nécessite la création d'interfaces dédiées.

3. *Les interfaces actives*
   Une interface active [63] est une interface qui peut ajouter des opérations à effectuer quand une méthode est invoquée (*i.e.* avant ou après l'appel). Par exemple, lorsqu'une méthode est appelée avec une série de paramètres, ces derniers peuvent subir des prétraitements ou être renvoyés vers d'autres méthodes. Chaque composant est associé à un composant chargé d'assurer l'appel aux méthodes définies par les interfaces actives. Ces deux composants communiquent au travers d'une interface spéciale.

4. *La transformation*
   La transformation consiste à modifier l'implémentation d'un composant (*i.e.* son code source). Cette opération peut être réalisée par l'intermédiaire d'un outil d'analyse et de transformation du code, séparé de l'application. Elle peut également être mise en œuvre au travers d'un service fourni par le composant à adapter. Il existe de multiples techniques permettant de réaliser la transformation de code :

   - *la modification sur place « open-source »*
     Cette technique réside dans le fait que le constructeur de l'application puisse si nécessaire modifier directement le code source des composants. Aussi, le constructeur doit être capable d'analyser le code d'un composant et de l'interpréter de manière à opérer des changements ;

   - *l'utilisation de langages de script*
     Les langages de script tels que Perl ou Tcl/Tk permettent de générer de nouveaux fichiers sources qui vont contenir la nouvelle implémentation du composant après adaptation. Le code va être modifié en fonction de règles prédéfinies par le programmeur. Ils permettent de réorganiser les composants en agissant directement sur le code source. Ils sont généralement utilisés pour spécifier comment un composant interagit avec un autre composant et comment les structures de données échangées doivent être transformées. Ils peuvent être utilisés pour générer les liens entre les composants pendant la phase d'assemblage d'une application ;

   - *l'utilisation de langage de composition*
     Ce sont des langages qui possèdent un plus haut niveau d'abstraction qui permet de travailler sur la composition des composants. Ils doivent être capable de manipuler à la fois des objets et des composants dont le contenu peut être distribué. Par exemple, PICT [113] et BML (*Bean Markup Language*) [129] sont des langages de composition ;

- *la restructuration*
  La restructuration de logiciels, aussi appelée refactorisation (*refactoring*) quand elle est utilisée dans le contexte d'applications orientées objets, est une transformation d'une représentation existante en une représentation sémantiquement équivalente sans changement de niveau d'abstraction [53]. Cette technique est généralement utilisée pour améliorer la structure d'un système [99] afin d'augmenter sa maintenabilité. De ce fait, le système devient plus facile à comprendre et donc certaines parties deviennent plus réutilisables [55]. De même, l'ajout de nouvelles fonctionnalités est alors plus aisé. Parmi les différentes applications à la restructuration, nous pouvons citer l'élimination du code dupliqué en vue d'améliorer sa lisibilité ou bien le partitionnement de code afin de permettre une exécution parallèle. Cette dernière opération consiste à fragmenter une application en différentes portions tout en préservant la sémantique de l'application initiale.

5. *L'héritage*
   Une technique d'adaptation pour la réutilisation est l'héritage. Cette technique, issue du monde objet, [130] permet à un composant d'acquérir les caractéristiques d'autres composants et ainsi d'obtenir des composants dotés de comportements analogues. Cependant, l'héritage est très peu utilisé par la communauté de « composants logiciels » au profit de la composition pour faciliter la réutilisation.

6. *Le « wrapping »*
   Un *wrappeur* [56, 70] est un conteneur d'objets qui encapsule le composant et qui fournit des interfaces qui permettent d'étendre les services proposés par le composant qu'il contient. Il peut donc contenir d'autres interfaces destinées à contrôler le composant qu'il contient ou à fournir de nouvelles fonctionnalités. Il n'y a aucune frontière claire entre l'emballage et l'agrégation, mais l'emballage est employé pour adapter le comportement du composant inclus tandis que l'agrégation est employée pour composer la nouvelle fonctionnalité hors des composants existants fournissant la fonctionnalité appropriée. Généralement, cette technique est utilisée pour étendre une classe ou bien transformer une interface d'une classe par une autre. Il existe plusieurs approches comme l'héritage pour les composants, l'agrégation, les filtres de composition ou les adaptateurs.

7. *La superposition*
   La superposition [29] consiste à permettre au constructeur d'application d'adapter un composant en utilisant des types d'adaptation prédéfinis et configurables de la même manière que BCA. Le principe de superposition est qu'un composant et la fonctionnalité adaptant le composant sont deux entités séparées mais qui ont besoin d'être intégrées. L'objectif est de fournir des composants adaptés dont la combinaison est réutilisable. Cette propriété exige un ensemble de types composants réutilisables d'adaptation. Ces types d'adaptation devraient être configurables et composables l'un avec l'autre de manière à tenir compte des adaptations composantes complexes.

8. *Les patrons de conceptions*
   Les patrons de conceptions (en anglais *design patterns*) [56] sont des techniques permettant de proposer des solutions à des problèmes récurrents. En fait, un patron de conceptions définit, explique et évalue de manière systématique une conception importante qui se reproduit dans les systèmes orientés objets. De ce fait, ils sont basés sur des expériences passées avec les mêmes structures.

Les patrons de conceptions permettent d'augmenter la productivité en adoptant certaines structures établies et réutilisables. En fait, ils proposent un niveau d'abstraction plus élevé permettant d'élaborer des applications de meilleure qualité. Les patrons de conceptions les plus utilisés pour mettre en place l'adaptation d'une application sont :

(a) le patron de conception « *Strategy* »
Ce patron de conception permet de définir des familles d'algorithmes encapsulés, interchangeables et associés à des contextes d'exécution. Ce patron de conception est très utilisé pour adapter le comportement d'une application lors de son exécution. En effet, il permet par exemple de définir, pour chaque composant contenu dans une application, plusieurs comportements différents (*i.e.* plusieurs implémentations pour un même service) qui vont être choisis en fonction du contexte au moment de l'exécution ;

(b) le patron de conception « *Adapter* » également appelé sous le nom de « *wrapper pattern* » ou plus simplement « *wrapper* »
Ce patron de conception permet de convertir une interface d'un composant ou d'un objet afin de la rendre conforme à celle attendue par le client (*i.e.* composant requérant cette interface pour fonctionner). Ce patron est utile lorsque l'on désire utiliser un composant existant mais que l'interface proposée n'est pas satisfaisante. Dans ce cas, le pattern propose de définir par héritage multiple un composant effectuant les actions d'adaptation nécessaires ;

(c) le patron de conception « *Observer* »
Ce patron est généralement utilisé pour envoyer des événements et des informations vers d'autres objets (les Observateurs ou « *Observer* »). Les objets *Observer* vont donc exécuter certaines actions en fonction des indications envoyées par l'objet Observable ;

(d) le patron de conception « *Memento* »
Ce patron permet d'externaliser et de sauvegarder l'état interne d'un objet de manière à pouvoir le restaurer. Ce pattern est très utilisé pour mettre en place l'adaptation dynamique d'une application.

9. *La réflexion*
La réflexion [119] est une technique utilisée pour raisonner et agir sur soi-même. Ainsi, un système réflexif peut observer son propre comportement (*i.e.* introspection) tout au long de son exécution et même agir sur lui même (*i.e.* intercession) en fonction de ses besoins. Un système réflexif est composé de deux niveaux distincts : d'une part, le niveau de base contenant le code métier du système et d'autre part, le niveau méta qui contient une représentation abstraite du système. L'accès au niveau méta à partir du niveau de base est réalisé via le concept de réification.

La réflexion est généralement utilisée pour réaliser l'adaptation dynamique d'une application. Par exemple, un système peut déceler une faille dans son comportement grâce à l'introspection et la corriger immédiatement par intercession sans stopper son exécution. La réflexion permet à la fois de modifier la structure et le comportement d'une application. Ainsi, on distingue deux types de réflexion :

(a) la réflexion comportementale
Elle réside dans la capacité du système à fournir une représentation complète de sa propre sémantique en termes d'aspects internes de son environnement d'exécution. Elle permet de modifier le comportement des processus de l'environnement sous-jacent, en prenant en compte les propriétés non fonctionnelles et la gestion des ressources. Par exemple, une application distribuée réflexive peut choisir d'utiliser un protocole de communication qui est bien adapté à son environnement d'exécution ;

(b) la réflexion structurelle
Elle réside dans la capacité du système à fournir une réification complète de lui-même en termes de structure : hiérarchie des objets, connexion des objets, état, type, etc. Grâce à la réflexion structurelle, il est possible de modifier les fonctionnalités du programme. Par exemple, un objet réflexif peut donner des renseignements sur les méthodes qu'il fournit.

L'utilisation de la réflexion pour l'adaptation a pour avantage majeur la séparation entre le code métier du système et le code qui permet de réaliser l'adaptation. De plus, la réflexion permet de définir des stratégies d'adaptation dynamique et générique car elle s'applique au niveau du méta-modèle de l'application. Cependant, la réification engendre un surcoût en temps d'exécution.

10. *Le tissage d'aspects*
La programmation par aspects (AOP) [77, 103] permet de réaliser une séparation des différentes préoccupations des programmeurs liées au développement d'applications. Généralement, ce sont les parties fonctionnelles qui sont dissociées de parties techniques de l'application. La fusion de ces deux parties est réalisée au moyen d'un tisseur permettant d'obtenir l'application finale. En fait, l'AOP a pour objectif d'isoler les définitions des propriétés transversales des applications (*i.e.* persistance, sécurité, transaction, duplication, cohérence, etc.). Ainsi, l'AOP propose des solutions permettant d'isoler chacune de ces propriétés, appelées aspects, dans des modules spécifiques et indépendants. Cette opération permet de faciliter leur définition et leur manipulation.
De ce fait, la programmation par aspects peut être utilisée pour introduire des mécanismes d'adaptation, aux composants par l'intermédiaire d'aspects tissés sur le code métier. Ce tissage peut également être réalisé au cours d'exécution de l'application (pour l'adaptation dynamique).

11. *L'utilisation de « Plugins »*
L'application utilise des outils qui permettent de découvrir les nouveaux services fournis par les « *plugins* ». Les protocoles mis en œuvre sont prédéfinis. Cette technique est généralement utilisée pour gérer l'évolution d'une application de par l'ajout de nouvelles fonctionnalités.

12. *Les cadres de conception*
Les cadres de conception (en anglais *frameworks*) sont des ensembles de composants qui définissent un plan abstrait orienté à résoudre des problèmes liés à un domaine particulier. Les *frameworks* sont divisés en deux catégories : les « boîtes blanches » et les « boîtes noires ». Dans le cas d'un *framework* « boîte blanche », il est nécessaire de connaître sa structure interne pour pouvoir l'utiliser. Un *framework* est catégorisé comme « boîte noire » dès lors que l'héritage est remplacé par la composition d'instances obtenues à partir des composants du *framework* [41].

Adaptation des liaisons entre composants (*i.e.* adaptation des connecteurs)

1. *L'interposition*
   Cette stratégie réside dans l'utilisation d'adaptateurs qui s'insèrent en amont des composants et qui interceptent les messages reçus [26]. Ces adaptateurs sont généralement implémentés sous la forme de composants ou de connecteurs [8].

2. *La délégation*
   La délégation consiste à déléguer le travail d'un composant vers un autre composant. Elle passe par la création de composants uniquement destinés à l'adaptation de l'application. Ces derniers sont chargés de répartir les tâches en désignant quel composant devra réaliser tel ou tel service en fonction du contexte [75]. Des outils de « *mapping* » permettant d'établir les correspondances entre les services sont utilisés pour mettre en place cette solution.

Adaptation de l'architecture de l'application

La technique utilisée pour adapter l'architecture d'une application [87] est appelée *reconfiguration* [47]. La reconfiguration est généralement réalisée de manière dynamique (*i.e.* lors de l'exécution de l'application) afin d'adapter l'application à son contexte d'exécution. Elle soulève trois problèmes majeurs :

1. *la définition des briques de bases utilisées pour la reconfiguration*
   La définition des briques de bases (*i.e.* unité de reconfiguration) est crucial dans la mise en œuvre d'une stratégie d'adaptation par reconfiguration. En effet, ces éléments vont conditionner les possibilités d'adaptation de l'application : plus le nombre de briques de bases est important, plus le nombre de combinaisons possibles par reconfiguration sera grand. Cependant, une trop grande quantité de briques de bases peut accroître fortement la complexité des mécanismes de prise de décisions de l'adaptation (choix des opérations de reconfiguration à réaliser pour adapter l'application/le composant au contexte) ;

2. *la définition d'un modèle de décisions permettant de reconnaître un état stable de l'unité à reconfigurer*
   Pour mettre en œuvre une opération reconfiguration, le composant doit se trouver dans un état stable (aucune de ses ressources ne doit être en cours d'utilisation). Aussi, il est nécessaire de définir des mécanismes permettant de stabiliser l'état d'un composant en cours d'exécution ;

3. *la mise en œuvre de la reconfiguration*
   La reconfiguration consiste à modifier l'architecture de l'application. Le mot architecture signifie ici, la description de la façon suivant laquelle les composants sont assemblés pour former l'application. Ainsi, la reconfiguration permet de modifier les liaisons entre les composants, de remplacer des composants, etc.

   La mise en œuvre de la reconfiguration fait généralement appel à des techniques utilisées pour l'adaptation d'un composant telles que la réflexion [119], etc.

Évaluation des techniques d'adaptation existantes Afin d'évaluer les techniques d'adaptation existantes (voir Tableau 1.7), nous avons établi les critères de comparaison suivant :

- *le type d'approche : « boîte noire » vs. « boîte blanche »* (C_1)
  Tout d'abord, les techniques d'adaptation existantes peuvent être classées en deux catégories : celles orientées « boîtes blanches » et celles orientées « boîtes noires ». Les techniques « boîtes blanches » exigent typiquement la connaissance de l'implémentation interne du composant à adapter, tandis que les techniques « boîtes noires » [79] exigent seulement la connaissance des interfaces du composant.
  La majorité des techniques existantes se prétendent « boîte noire ». Ceci est généralement dû à leur spécificité à un modèle de composants lui permettant d'acquérir des informations pour fixer une stratégie d'adaptation et de mettre à jour l'application en utilisant les propriétés d'intercession du modèle. Dans le cas où la spécificité à un modèle d'application adaptable ou adaptative n'est pas avérée, la technique « boîte noire » la plus répandue est le « *wrapping* ». Il existe également la *superimposition* [29] qui est une technique alternative. L'idée principale qui en ressort est que la fonctionnalité entière d'un composant (*i.e.* plutôt que celle d'une méthode simple) devrait être en surimpression vis-à-vis de certains comportements ;

- *la stratégie d'adaptation : transformation vs. habillage* (C_2)
  Les techniques d'adaptation peuvent être catégorisées en deux parties : d'une part, les techniques de transformation (telles que la modification de code, la restructuration, etc.) qui consiste modifier le code du composant à adapter de manière à ce qu'il réponde à de nouveaux besoins. Ce type de techniques peut permettre à la fois d'adapter le comportement et la structure d'un composant/d'une application ; d'autre part, les techniques d'habillage (telles que le *wrapping*, la superposition, etc.) qui consistent à intégrer à un composant ou à une application de nouveaux mécanismes. Ces mécanismes généralement « boîtes noires » sont utilisés pour ajouter de nouvelles fonctionnalités (non fonctionnelles) à un composant. Ainsi, la plupart des techniques d'habillage permettent de modifier uniquement le comportement d'un composant/d'une application et non sa structure ;

- *l'indépendance au modèle de composants utilisé* (C_3) et *à la plate-forme d'implémentation utilisée* (C_4)
  Les techniques d'adaptation peuvent être indépendantes de tout modèle de composants utilisé et de toute plate-forme d'implémentation. Cependant, la plupart des techniques « boîtes noires » requièrent des propriétés spécifiques au modèle ou à l'implémentation du modèle, telles que la réflexion, ou la possibilité de réaliser de la reconfiguration dynamique, etc. En fait, il existe très peu de techniques qui se veulent totalement génériques ;

- *la nécessité de traitements préalables* (C_5)
  Certaines techniques d'adaptation peuvent requérir des traitements spécifiques pour être mises en œuvre ; c'est notamment le cas concernant certains patrons de conception tel que le patron « *Strategy* » pour lequel le concepteur du composant doit prévoir plusieurs implémentations pour un même composant : chacune étant associée à un contexte spécifique ;

- *la cible de l'adaptation : comportement vs. structure* (C_6)
  Les techniques d'adaptations d'application à base de composants peuvent agir sur le comportement des composants ou bien sur leur structure. Comme nous pouvons le constater dans le tableau 1.7, la

grande majorité des techniques existantes permettent essentiellement d'adapter le comportement de l'application. Peu de techniques permettent d'adapter leur structure.

Technique	C_1	C_2	C_3	C_4	C_5	C_6
BCA	Boîte noire	Transformation	Non	Non	Non	Comportement
Paramétrisation	Boîte noire	Autre	Non	Non	Non	Comportement
Interfaces actives	Boîte noire	Habillage	Non	Oui	Oui	Comportement
Modification de code	Boîte blanche	Transformation	Oui	Oui	Non	Comportement/structure
Langages de script	Boîte blanche	Transformation	Oui	Oui	Non	Comportement/structure
Héritage	Boîte noire	Habillage	Non	Non	Non	Comportement/structure
Wrapping	Boîte noire	Habillage	Non	Non	Non	Comportement
Superposition	Boîte noire	Habillage	Non	Non	Oui	Comportement
Restructuration	Boîte blanche	Transformation	Oui	Oui	Non	Structure
Patrons de conceptions	Boîte blanche	Transformation/Habillage	Oui	Oui	Oui	Comportement/structure
Réflexion	Boîte noire	Habillage	Non	Non	Non	Comportement/structure
Tissage d'aspects	Boîte noire	Transformation	Oui	Non	Non	Comportement
Plugins	Boîte noire	Habillage	Oui	Oui	Non	Comportement
Cadres de conception	Boîte noire	Habillage	Oui	Oui	Non	Comportement
Interposition	Boîte noire	Habillage	Non	Non	Oui	Comportement
Délégation	Boîte noire	Habillage	Non	Non	Oui	Comportement
Reconfiguration	Boîte blanche	Transformation	Non	Non	Non	Comportement/structure

Table 1.7 – Évaluation des techniques d'adaptation existantes

1.4.3 Étude comparative des approches d'adaptation existantes

Comme nous avons pu le constater, beaucoup d'approches ont été proposées dans la littérature afin d'adapter des composants et des applications conçues à base de composants. Nous avons classifié les travaux que nous avons répertoriés suivant les critères que nous avons cités précédemment (voir Tableau 1.8). Une présentation de chaque travaux recensés est fournie dans l'annexe B.

1.4.3.1 Classification des approches existantes suivant les raisons de l'adaptation

Comme nous pouvons le constater, étant donné que la majorité des travaux actuels se positionnent dans des environnements ubiquitaires et mobiles, la plupart des approches visant à adapter une application conçue à base de composants a pour objectif de la faire évoluer en fonction de son environnement d'exécution (*i.e.* adaptation adaptative) ou à la rendre plus performante sous certaines conditions (par exemple, [44, 47, 81, 95]) ; et, de ce fait répondre en partie, aux attentes des utilisateurs dans ce type d'environnement.

Cependant, peu de travaux s'intéressent à rendre une application plus flexible (par exemple, [33, 50, 58, 73]) ou à améliorer la réutilisabilité des composants logiciels pour leur intégration dans des applications destinées à être exécutées dans des environnements totalement différents de ceux pour lesquels ils ont été conçus (par exemple, [29, 46]). Par exemple, un composant qui a été conçu pour être déployé sur un ordinateur standard (*i.e.* ordinateur de base disponible dans le commerce) peut ne pas être réutilisable pour la conception d'une application destinée à être exécutée dans un environnement à ressources limitées (PDA, téléphone portable, etc.).

1.4.3.2 Classification des approches existantes suivant la cible de l'adaptation

L'adaptation d'une application peut concerner son comportement ou sa structure. Cependant, nous avons constaté que la majorité des approches existantes a pour objectif de faire évoluer le comportement en fonction de son environnement d'exécution (*i.e.* adaptation adaptative) ou de le rendre plus performant (par exemple, [28, 44, 73, 114]). Peu de travaux se consacrent à adapter la structure de l'application afin de la rendre plus flexible (par exemple, [9, 47, 50]). Or, cette stratégie est grandement encouragée par le fort développement de l'informatique ubiquiste où la flexibilité est indispensable pour permettre le déploiement d'application dans certaines conditions (ressources limitées par exemple) et garantir la continuité et la qualité de service comme nous pourrons le constater dans la section 1.5.

Par ailleurs, les approches qui proposent d'adapter la structure d'une application à son environnement d'exécution se focalisent sur le placement des composants sur l'infrastructure de déploiement disponible (*i.e.* adaptation de l'architecture) (par exemple, [3]) et peu se proposent d'adapter l'application au niveau des composants logiciels (par exemple, [46, 47, 58]).

De plus, les approches qui permettent d'adapter la structure d'un composant logiciel présentent également d'autres limitations, notamment relatives au type de composants pris en considération. En effet, les approches destinées à adapter la structure d'un composant logiciel peuvent être catégorisées en deux parties : d'une part, les approches qui se proposent d'adapter des composants hiérarchiques (par exemple, [46, 47, 58]). Dans ce cas, l'adaptation prend généralement la même forme que l'adaptation d'une architecture (*i.e.* gestion des sous-composants, choix de configuration, etc.) ; d'autre part, l'adaptation de composants monolithiques. Dans ce cas, l'adaptation se traduit généralement soit par la reconfiguration de ses paramètres (*i.e.* attributs, etc.), soit par la mise en œuvre du patron de conception « *Strategy* » (*i.e.* définition de plusieurs implémentations possibles pour un même composant et sélection en fonction du contexte de l'implémentation à exécuter) qui permet de définir plusieurs comportements pour un même composant (par exemple, [28, 44, 95]). Ainsi, seul le comportement du composant peut être adapté.

A notre connaissance, il n'existe pas d'approche permettant d'adapter la structure d'un composant logiciel monolithique.

1.4.3.3 Classification des approches existantes suivant leur prise en considération de composants existants

Nous avons également constaté que la majorité des travaux que nous avons recensés sont étroitement liés à un modèle de composants particulier ou proposent eux-mêmes leur propre modèle de composants adaptatifs (par exemple, [28, 44, 47, 50]). De ce fait, l'adaptation de composants logiciels existants exige généralement que ces derniers soient développés à nouveau afin de les rendre conforme à un modèle spécifique lui garantissant son adaptation. Un problème se pose alors pour les composants existants tels que les COTS [67] qui ne peuvent être adaptés que s'ils sont développés à nouveau ; ce qui ne peut être envisagé pour des raisons de coût.

Pour que ces approches puissent être utilisées dans le cadre de l'adaptation de composants existants, les travaux en question doivent fournir des mécanismes (généralement un processus de transformation) permettant de transformer tout type de composants en un composant conforme au modèle qu'ils proposent. Cependant, dans la majorité des travaux étudiés, aucune méthode n'est proposée pour obtenir un tel composant. De plus, peu de travaux proposent des approches génériques (*i.e.* indépendantes de tout modèle de composants) pour adapter ou rendre adaptatif des composants logiciels existants (par exemple, [3, 33, 58]) bien que cette solution soit, avec la précédente (proposition d'un modèle et d'un processus permettant d'obtenir un composant existant conforme à ce modèle), la mieux appropriée pour favoriser la réutilisation de composants existants.

1.4.3.4 Classification des approches existantes suivant l'environnement d'exécution

A cause de la démocratisation des terminaux mobiles tels que les téléphones portables, PDA et autres, depuis quelques années, la majorité des travaux portant sur l'adaptation d'applications en général et plus récemment, d'applications conçues à base de composants logiciels se positionnent dans le cadre d'environnements ubiquitaires et mobiles (par exemple, [47, 81, 95]). En effet, l'informatique ubiquitaire et mobile dispose de caractéristiques telles que l'hétérogénéité des terminaux et des moyens de communication, qui rendent le domaine de recherche unique et fertile.

Dans le domaine des composants logiciels, les projets existants tentent de proposer des approches visant à introduire certaines propriétés liées à l'ubiquité telles que la prise en compte du contexte ou bien la gestion des connexions et des déconnexions des ressources d'un tel réseau, à des applications existantes.

1.4.3.5 Classification des approches existantes suivant le moment et la dynamicité de l'adaptation

La plupart des approches proposées dans la littérature ont pour objectif d'adapter les applications au moment de leur exécution et ce, sans l'arrêter (par exemple, [28, 44, 47, 76]). Ceci est dû, comme nous l'avons évoqué précédemment, au fort développement de l'informatique ubiquitaire qui nécessite la prise en compte permanente du contexte d'exécution par l'application de manière à lui garantir une continuité de service et une qualité de service.

D'autres approches se sont focalisées sur le déploiement d'applications conçues à base de composants (par exemple, [3, 46]). Cependant, comme nous l'avons évoqué, ces travaux se focalisent sur le placement des composants à déployer sur l'infrastructure disponible. A notre connaissance, aucun travail ne permet d'adapter le déploiement de composants logiciels.

Concernant les approches statiques (par exemple, [73]), elles sont destinées généralement à introduire des points de variabilité permettant à l'adaptation ou aux composants d'adapter leur comportement lors de l'exécution.

1.4.3.6 Classification des approches existantes suivant le niveau d'automatisation de l'adaptation

La plupart des travaux existants ont pour objectif de concevoir des applications auto-adaptatives (*i.e.* l'adaptation est réalisée par l'application elle-même) dès lors qu'ils se positionnent dans un contexte d'adaptation dynamique (par exemple, [46, 47, 58]). En effet, dans de nombreux environnements tels que les environnements ubiquitaires, le contexte d'exécution est en perpétuelle évolution. De ce fait, une adaptation manuelle, réalisée par l'administrateur de l'application ne peut être envisagée. Généralement, une couche non-fonctionnelle est intégrée aux composants afin de les rendre auto-adaptatifs. Cette couche peut prendre la forme d'un service non-fonctionnel intégré aux composants (par exemple, [47]), d'un composant non-fonctionnel ajouté au modèle (par exemple, [3, 9, 81]) ou bien d'un niveau méta (par exemple, [44, 58, 95]) permettant d'agir sur l'application pour l'adapter (*i.e.* déclenchement de l'adaptation, prise de décisions et réalisation de l'adaptation).

Dans la majorité des approches, l'adaptation doit être prévue dès la conception et le développement de l'application (par exemple, [28]). Par exemple, certaines stratégies d'adaptation utilisant le patron de conception « *Strategy* » doivent faire appel aux programmeurs qui sont chargés d'implémenter différentes versions d'un même composant de manière à ce qu'il exécute celle la mieux adaptée au contexte courant. Cependant, ce type d'adaptation est très limitatif car il ne permet pas de réagir à des évènements non prévus par les programmeurs. En effet, toutes les situations doivent être prévues dès la conception de l'application.

De plus, les politiques d'adaptation sont généralement définies par l'administrateur de l'application sous forme règles ECA (Évènement Condition Action) (par exemple, [47, 58]). Ces politiques peuvent dans certains cas être insérées pendant l'exécution de l'application. De ce fait, le rôle de l'administrateur reste prépondérant dans la majorité des approches proposées.

1.4.3.7 Classification des approches existantes suivant les techniques d'adaptation utilisées

Les techniques d'adaptation utilisées dans le cadre des approches existantes sont étroitement liées à la cible de l'adaptation et au moment de l'adaptation.

Concernant les techniques permettant d'adapter le comportement d'une application ou d'un composant, les plus utilisées dans la littérature sont la réflexion (par exemple, [44, 47, 50]) et la mise en œuvre de patron de conception (par exemple, [28, 44, 95]). En effet, la réflexion offerte, comme nous avons pu le constater précédemment, par la plupart des infrastructures logicielles permet d'adapter facilement le comportement d'une application ou des entités qui la compose de par l'utilisation des propriétés d'intercession qui permettent d'agir sur ces entités. Concernant les patrons de conception, les plus utilisés sont le patron d'observation (*Observer*) et le patron *Strategy*. Ces derniers permettent d'observer le contexte d'une application et d'adapter le comportement des composants en sélectionnant celui qui est le plus adapté à la situation.

Concernant les techniques permettant d'adapter la structure d'une application ou d'un composant, la plus utilisée est la reconfiguration (par exemple, [9, 47, 58, 81]). Cette technique permet de réorganiser la structure d'une application ou d'un composant de par la modification des connexions entre ses composants ou bien l'ajout, le remplacement ou la suppression de composants. Cependant, cette technique ne permet d'adapter que des assemblages de composants déjà existants (assemblages à plat ou bien assemblages hiérarchiques).

Très peu d'approches s'intéresse à adapter des composants sous forme monolithique. En fait, dans l'ingénierie des composants logiciels, seul le projet Coign [69] propose une approche permettant de distribuer automatiquement des composants COM sous format binaire. Coign est basé sur la construction d'un graphe représentant les interactions entre les différents composants d'une application. Des algorithmes de découpage sont alors appliqués sur ce graphe afin de minimiser le nombre de communications susceptibles d'être réalisées entre différentes parties du graphe ainsi que les délais d'attentes liés à l'infrastructure distribuée. Il dispose également d'algorithmes d'optimisation. Cependant, Coign ne traite pas tous les problèmes liés au partitionnement de code. Par exemple, le processus ne peut pas s'appliquer à des composants partageant des données au travers de pointeurs mémoires. De plus, il est spécifique à un modèle en l'occurrence le modèle COM ; ce qui exclut donc la grande majorité des composants existants.

Ainsi, nous pouvons constater qu'aucune approche proposée dans la littérature ne permet d'adapter la structure de tout type de composants logiciels existants (monolithiques et composites).

L'adaptation de la structure a cependant été étudiée dans le domaine de l'orienté objet où de nombreuses approches permettant de partitionner des applications existantes ont été proposées.

Jamwal et Iyer proposent dans [71] de construire des objets « cassables » appelés *Bobs* permettant de construire des architectures flexibles. Un Bob est une sorte d'objet qui possède une structure simplifiée de manière à pouvoir restructurer facilement ses fonctionnalités (*i.e.* son code). L'objectif est d'intégrer à ces objets des mécanismes permettant de mettre en œuvre leur fragmentation (découpage en sous-entités) et leur redéploiement. Dans leur implémentation réalisée en Java, les *Bobs* sont des classes Java Standard ayant des caractéristiques restreintes (pas d'héritage, pas d'objet actif, pas d'attribut public, etc.). Pour chaque *Bob*, le programmeur doit fournir les méthodes publiques de la classe et une spécification des

méthodes qui ne peuvent pas être séparées lors d'une fragmentation. Ces données sont alors compilées à l'aide d'un outil fourni par le projet, afin d'obtenir le code source Java correspondant. Une application construite dans le cadre de cette approche est donc constituée d'objets Java standard et de *Bobs*.

Pangaea [120] est un système qui permet de rendre distribué des programmes Java standard. Il est basé sur l'analyse statique de code source et permet de déterminer la stratégie de distribution la mieux adaptée au contexte et d'introduire des mécanismes de distribution, indépendamment de la plate-forme et de manière transparente et automatique. Ce système ne pose pas de restriction sur les objets Java utilisés ; cependant, il requiert l'utilisation d'un compilateur spécifique fourni dans le cadre du projet. Pangaea se focalise plus particulièrement sur les applications de type client/serveur. Par ailleurs, Pangaea comporte un outil graphique permettant de représenter la structure d'un programme comme un graphe d'objet.

Le projet Addistant [124] est également consacré au partitionnement de code binaire Java. En effet, le système Addistant permet de distribuer du code Java qui a été conçu pour être exécuté sur une seule machine virtuelle. Les utilisateurs du système doivent d'une part indiquer les sites sur lesquels les instances de classe doivent être allouées, d'autre part, définir comment sont implémentés les références sur des objets distants. Addistant transforme alors automatiquement le code binaire de l'application au moment de son chargement dans la machine virtuelle, en fonction des fichiers de configuration fournis par l'utilisateur. Addistant utilise une approche basée sur des *proxys* générés automatiquement pour établir des références distantes. Cependant, Addistant impose certaines restrictions au niveau des classes qui doivent être modifiables afin d'être traitées.

J-Orchestra [125] est également un système permettant de partitionner automatiquement des programmes Java. Il est relativement similaire au projet Addistant. En fait, il réside dans la transformation de code Java sous un format binaire en application distribuée pouvant être exécutée sur des machines virtuelles différentes. Ainsi, les appels locaux sont transformés en appels distants, les références directes à des objets locaux sont transformées en références vers des *proxys*, etc. Il utilise des fichiers de configurations similaires à ceux utilisés dans le cadre du projet Addistant. Cependant, ces derniers sont automatiquement générés De plus, J-Orchestra offre un support pour la migration d'objets et l'optimisation dynamique de la configuration de distribution. En effet, une fois le déploiement de l'application réalisé, J-Orchestra offre à l'utilisateur la possibilité de déplacer les classes d'un site à un autre. Des outils d'analyse et de supervision vérifient en permanence la validité du partitionnement.

D'autres projets tels que Orca [6] et JavaParty [105] ont pour objectifs d'automatiser les décisions permettant de paralléliser une application centralisée.

1.4.4 Bilan de l'étude comparative des approches d'adaptation existantes

Comme nous avons pu le constater, la réutilisation à grande échelle de composants logiciels existants est un challenge pour la conception de nouvelles applications. Dans la grande majorité des cas, pour être intégrés à une application, les composants disponibles ont besoin d'être adaptés. Pour répondre à ce besoin, beaucoup d'approches ont été proposées dans la littérature. Néanmoins, nous pouvons constater que, malgré cette grande diversité des approches proposées, la majorité se focalise sur l'adaptation du comportement des composants ou de leur assemblage (voir Tableau 1.8). Cependant, dans certains cas, et notamment pour des applications destinées à être exécutées dans des environnements ubiquitaires, adapter le comportement de composants n'est pas suffisant pour permettre leur réutilisation : il est indispensable d'adapter également leur structure.

Concernant l'adaptation de la structure d'un composant ou d'une application, les approches existantes permettent, pour la plupart, d'adapter des assemblages de composants que ce soit au niveau global (*i.e.* au niveau de l'architecture de l'application) ou au niveau des composants dans le cas de modèles hié-

Travaux	Type	Environnement	Raisons	Automatisation (acteurs)	Moment (dynamicité)	Facette cible (entités cibles)	Techniques
ACEEL [44]	Modèle spécifique *Framework*	Ubiquitaire Mobile	Adaptivité Évolution Performance	Semi-automatique (Couche méta, administrateur)	Pendant l'exécution (Dynamique)	Comportement (Composants)	Réflexion Patron « Strategy » AOP
Adaptive component [28]	Modèle spécifique	*n.c.*	Adaptivité Performance	Semi-automatique (Concepteur, compilateur, composants)	Pendant l'exécution (Dynamique)	Comportement (Composants)	Patron « Strategy »
Safran [47]	Modèle spécifique *Framework*	Ubiquitaire Mobile	Adaptivité Performance	Semi-automatique (Administrateur, couche méta)	Pendant l'exécution (Dynamique)	Comportement Structure (Composants + Configuration)	Réflexion AOP Transformation
K-component [50]	Modèle spécifique	Ubiquitaire Mobile	Adaptivité Flexibilité Performance	Semi-automatique (Composants, administrateur, couche méta)	Pendant l'exécution (Dynamique)	Structure (Configuration)	Réflexion Reconfiguration
Casa [95]	*Framework*	Ubiquitaire Mobile	Adaptivité Performance	Automatique (Composants spécifiques, couche méta)	Pendant l'exécution (Dynamique)	Comportement Structure (Connecteurs + configuration)	Réflexion Patron « Strategy » Paramétrage Reconfiguration
Plasma [81]	*Framework*	Ubiquitaire Mobile	Adaptivité Performance	Automatique (Composants spécifiques)	Pendant l'exécution (Dynamique)	Structure (Configuration)	Reconfiguration
Molène [114]	*Framework*	Mobile	Adaptivité Performance	Semi-automatique (Administrateur, composants spécifiques)	Pendant l'exécution (Dynamique)	Comportement (Composants)	Réflexion Paramétrage Reconfiguration
BCA [73]	Processus	*n.c.*	Correction Évolution Flexibilité Performance	Semi-automatique (Administrateur, compilateur)	Avant le déploiement (Statique)	Comportement (Composants)	Transformation
Dyva [74]	Processus	Distribué	Adaptivité Correction Évolution Performance	Semi-automatique (Concepteurs, composants)	Pendant l'exécution (Dynamique)	Comportement (Composants)	Interception
Brogi *et al* [33, 35]	Modèle abstrait Processus	*n.c.*	Adaptivité Évolution Flexibilité Interopérabilité	Automatique (Composants)	Pendant l'exécution (Dynamique)	Comportement (Composants)	Transformation
CADeComp [3]	Modèle abstrait Processus	Mobile	Adaptivité Flexibilité Performance	Semi-automatique (Concepteur, composants dédiés)	Pendant le déploiement (Dynamique)	Comportement Structure (Composants + configuration)	Patrons
Satin [97]	Modèle abstrait	*n.c.*	Correction	Automatique (Couche méta)	Pendant l'exécution (Dynamique)	Comportement Structure (Composants)	Réflexion
MaDcAr [58]	Modèle abstrait	Ubiquitaire Mobile	Adaptivité Flexibilité Performance	Semi-automatique (Composants spécifiques, administrateur, concepteur, couche méta)	Pendant l'exécution (Dynamique)	Comportement Structure (Configuration)	Réflexion Reconfiguration
Superimposition [29]	Processus	*n.c.*	Adaptivité Réutilisabilité	Semi-automatique (Concepteur, composants)	Pendant l'exécution (Dynamique)	Comportement (Connecteurs)	Transformation
Concerto [46]	*Framework*	Distribué Grids	Adaptivité Performance Réutilisabilité	Automatique (Composants)	Pendant le déploiement (Dynamique)	Comportement Structure (Composants)	
TranSAT [9]	*Framework*	*n.c.*	Évolution	Semi-automatique (Composants spécifiques, administrateur)	Pendant l'exécution (Dynamique)	Structure (Configuration)	Réflexion Reconfiguration AOP

Table 1.8 – Tableau comparatif des approches d'adaptation de composants logiciels

rarchiques (*i.e.* adaptation de l'assemblage des sous-composants). Peu d'approches se proposent d'adapter les applications au niveau des composants monolithiques car elles les considèrent comme des blocs de base qui ne peuvent être adaptés que par le paramétrage de leurs attributs ou par la mise en œuvre du patron de conception « *Strategy* » qui prévoit plusieurs implémentations possibles pour un même composant (chacune étant associée à un contexte spécifique). De ce fait, seul le comportement des composants monolithiques peut être adapté et non leur structure. Ainsi, dans la littérature, sont considérés comme structurellement adaptables, seuls les composants conçus initialement comme des assemblages de composants.

Le résultat est qu'aucune approche ne propose de techniques pour restructurer des composants monolithiques, bien que ce type d'adaptation puisse se révéler nécessaire dans beaucoup de situations. Pour illustrer ceci, considérons un composant monolithique pour lequel l'administrateur de l'application, souhaite distribuer les services qu'il fournit sur différents sites. Par exemple, en présence d'une infrastructure distribuée, les services directement liés aux bases de données pourraient être déployés sur un serveur sécurisé alors que les autres services seraient répartis sur plusieurs autres sites en fonction de leurs capacités (bande passante, vitesse de processeur, etc.). Cette répartition du composant pourrait être due à une contrainte technique ne permettant pas de le déployer dans son intégralité sur le site prévu à l'origine car les ressources disponibles sur ce site sont jugées insuffisantes. Il est évident que, dû à l'implémentation monolithique du composant, il est impossible d'obtenir la configuration de déploiement souhaitée. Il est donc indispensable de procéder à une adaptation de la structure du composant en question. Cette adaptation peut consister, par exemple, à partitionner le composant initial pour pouvoir le répartir sur l'infrastructure disponible.

1.5 L'adaptation logicielle dans les environnements ubiquitaires

Dans cette section, nous étudions un type particulier d'environnement : les environnements ubiquitaires et leurs spécificités par rapport aux besoins d'adaptation.

1.5.1 Spécificités de l'adaptation logicielle dans les environnements ubiquitaires

1.5.1.1 Présentation générale de l'informatique ubiquitaire

Le concept de l'informatique ubiquitaire a été introduit dans les années 90 comme la troisième vague d'informatique qui vient après l'ère des ordinateurs centraux et des ordinateurs personnels (voir Figure 1.8 [128]). À la différence des générations précédentes, l'informatique ubiquitaire s'exécute dans l'arrière plan de la vie quotidienne : les ordinateurs deviennent ainsi encastrés, adaptés, si naturellement, que nous les employons sans même penser à eux, ils sont invisibles et omniprésents [131]. Les promoteurs de cette idée espèrent que l'intégration du calcul dans l'environnement permettra aux gens de se déplacer et interagir avec d'autres ordinateurs plus naturellement qu'actuellement.

Cette nouvelle approche de l'informatique se différencie par le fait qu'elle intègre en premier lieu la mobilité de l'utilisateur. L'accès aux services se fait alors à n'importe quel moment et de n'importe où (*anytime anywhere*), ce qui signifie un accès sans fil et des terminaux mobiles. Ce type de réseau et ces terminaux impliquent une prise en compte de contraintes telles que la bande passante limitée, une latence plus importante, des déconnexions et des taux d'erreurs élevés, une capacité mémoire plus faible ou encore une gestion de l'énergie. La mobilité nécessite également de délimiter la zone de couverture et de router les données en fonction de la localisation du terminal.

En second lieu, l'informatique ubiquitaire repose sur la notion d'espaces intelligents. L'objectif est alors de pouvoir accéder à l'information tout le temps et partout (*all the time anywhere*). Cet objectif nécessite la prise en compte de l'environnement de l'utilisateur et la mise en place d'une communication entre les objets et l'utilisateur (ou son terminal) pour pouvoir accéder à l'information. Dans ce cadre, l'utilisation des réseaux *ad-hoc* permet de capter et diffuser de l'information partout et de gérer l'interaction avec l'environnement. Le contexte de l'utilisateur doit alors être représenté. On indique par exemple la taille du terminal de l'utilisateur, ou encore la liste des éléments nécessaires au bon fonctionnement de l'application. Cette approche permet de rendre invisible l'informatique à la fois de façon réelle par la miniaturisation des technologies, mais également de façon mentale par le fait que l'utilisateur est de plus en plus familier avec la technologie. Par exemple, avec ces espaces intelligents, on devrait lui permettre d'étendre l'interface de l'ordinateur de bureau vers tous les objets de son environnement (PDA, téléphone, écran, etc.) et prendre en compte ses différentes manifestations (voix, mouvement, activité, regards, etc.). Finalement, cette approche repose sur la géolocalisation qui permet de localiser l'utilisateur par rapport au réseau, puis le terminal le plus proche et non le terminal de l'utilisateur.

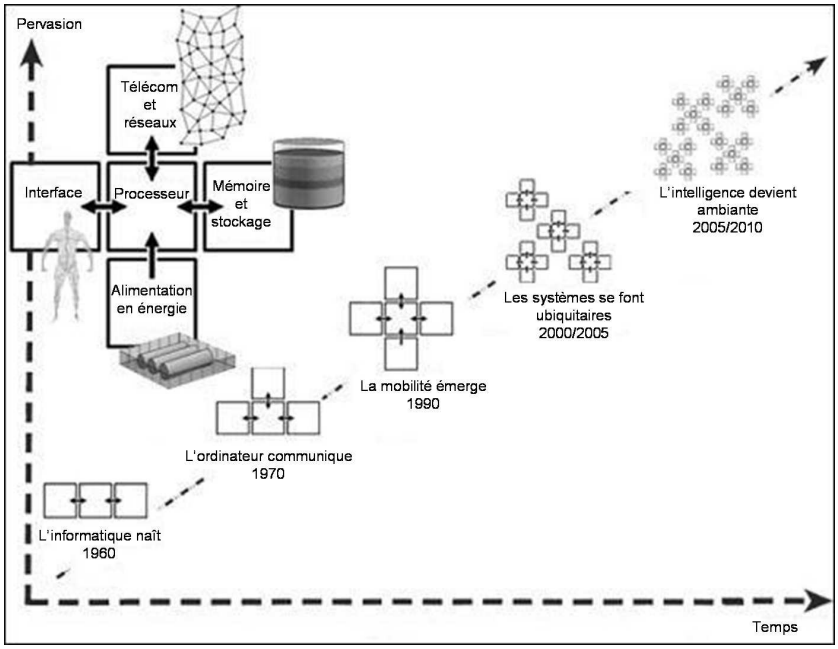

Figure 1.8 – Positionnement de l'informatique ubiquitaire dans l'évolution de l'informatique

1.5.1.2 Variabilité et adaptation par rapport aux spécificités des environnements ubiquitaires

Pour permettre la prise en compte des contraintes liées à ces environnements ubiquitaires, il est indispensable de proposer de nouvelles méthodes de conception d'applications ainsi que de nouvelles infrastructures logicielles.

Tout d'abord, la création d'application dites « ubiquitaires » amène à repenser les infrastructures logicielles pour les rendre adaptables dynamiquement, configurables et auto-administrables en fonction de l'environnement dans lequel elles se trouvent. Ces infrastructures prendront en compte la répartition, l'hétérogénéité, la mobilité mais également les ressources limitées des supports matériels. Elles doivent

alors offrir des garanties de sûreté de fonctionnement et de qualité de service dans leur comportement et dans les services offerts.

La conception d'applications fonctionnant dans ce type d'environnement pose également de nouveaux problèmes en plus de ceux dus à la réalisation d'applications réparties. Ces problèmes tels que la mobilité des utilisateurs, l'hétérogénéité des terminaux et des systèmes doivent généralement être pris en compte au moment de la conception de ce type d'applications. En fait, toute application destinée à être exécutée dans un environnement ubiquitaire doit présenter un ensemble de caractéristiques ou propriétés spécifiques. Ces caractéristiques peuvent être catégorisées en deux parties (voir Tableaux 1.9 et 1.10) : d'une part les caractéristiques fonctionnelles ; et d'autre part les caractéristiques non-fonctionnelles de l'application. Ces caractéristiques peuvent concerner le comportement ou la structure de l'application ou des entités logicielles qui la composent.

Caractéristiques fonctionnelles d'une application ubiquitaire Tout d'abord, une application ubiquitaire doit présenter un ensemble de caractéristiques fonctionnelles spécifiques afin de pouvoir être exécutée dans ce type d'environnement. Ces propriétés sont les suivantes :

- *la transparence des interfaces*
  Une des caractéristiques d'une application ubiquitaire réside dans sa capacité à masquer ses mécanismes fonctionnels. Cette capacité offre à l'utilisateur des modes d'interaction plus naturels avec son milieu. A terme, ces interactions devraient avoir lieu sans que l'utilisateur ait conscience d'utiliser les services fournis par des machines distribuées.
  Cette caractéristique doit être prise en compte au moment de la conception de l'application (et plus particulièrement de ses interfaces de communication avec l'utilisateur). Elle requiert la mise en place d'outils permettant d'acquérir le contexte d'exécution de l'application. Les informations ainsi récoltées doivent être utilisées afin d'anticiper les besoins de l'utilisateur et de guider les interactions entre les deux parties ;

- *l'adaptation aux moyens d'exécution*
  Les caractéristiques des machines sur lesquelles l'application est exécutée peuvent être très différentes (ordinateur de bureau, PDA, téléphone, etc.). L'application doit prendre en compte les moyens qui sont mis à sa disposition (réseaux disponibles, périphériques disponibles, etc.). L'application doit être capable de prendre en compte ces informations afin d'assurer un interfaçage de l'application qui soit la mieux adaptée à la situation. En effet, une interface *Homme-Machine* doit impérativement être exécutée sur la machine de déploiement de l'application.
  Leur prise en compte des possibilités de l'application à s'adapter à ce type de ressource doit être réalisée au moment de la conception de l'application. Des stratégies doivent être étudiées en fonction des ressources disponibles. Par exemple, si la machine d'installation ne dispose pas de sorties audio, l'application doit opter pour d'autres types de communication (par exemple, utilisation de l'écran de la machine ou d'un vibreur) ;

- *l'adaptation à la variation de l'environnement proche du terminal*
  Dans un environnement ubiquitaire et mobile, les conditions d'exécution d'une application varient en permanence. L'utilisateur et même parfois le terminal peuvent être en mouvement. L'application doit pouvoir être capable d'utiliser tous les moyens de communication avec l'utilisateur de manière adaptée à l'environnement. Par exemple, lorsque l'environnement d'exécution est très bruyant comme dans des lieux publics (gare, aéroports, etc.), l'application doit privilégier l'affi-

chage sur un écran plutôt que le son. Alors que lorsque l'utilisateur est en mouvement (marche, course, etc.), il est préférable qu'il utilise la voix pour communiquer ;

- *la mobilité*
  L'utilisateur doit pouvoir accéder à des données personnelles à partir de n'importe quelle station de travail. La sécurité est donc un critère essentiel que doit préserver l'adaptation. Un système d'identification de l'utilisateur doit donc être mis en œuvre.

L'ensemble des propriétés qui ont été énoncées ci-dessus sont des propriétés fonctionnelles. De plus, elles sont spécifiques à l'application en question et concernent plus particulièrement son comportement. De ce fait, leur introduction dans une application existante ne peut pas être réalisée de manière automatique. En fait, c'est au concepteur de l'application et des entités logicielles qui la composent de les introduire. L'intégration de ces propriétés n'entre donc pas dans les objectifs de cette thèse.

Caractéristiques non-fonctionnelles d'une application ubiquitaire Pour être exécutée dans un environnement ubiquitaire, une application logicielle doit également présenter un ensemble de propriétés non-fonctionnelles spécifiques. Ces propriétés liées à l'ubiquité sont les suivantes :

- *la distribution*
  Les services de l'application doivent être accessibles à partir de n'importe quel nœud de l'infrastructure disponible. Une application ubiquitaire doit donc intégrer des mécanismes de distribution. Or, l'intégration de tels mécanismes peut se révéler problématique dans de nombreux cas. Par exemple, un composant monolithique ne peut être distribué de manière *ad-hoc*. Pour répondre à cette caractéristique, il est donc indispensable de pouvoir modifier la structure du composant afin de lui introduire des mécanismes de distribution ;

- *l'adaptation aux moyens d'exécution*
  L'hétérogénéité des moyens d'exécution fait à la fois partie des caractéristiques non fonctionnelles liées aux applications ubiquitaires. En effet, les machines de déploiement d'une application peuvent disposer de capacités (batterie, taille de l'écran, puissance du processeur, capacité de stockage, etc.) très variables ; allant du simple téléphone portable doté de capacités minimales au *cluster* de milliers d'ordinateurs multi-processeurs. L'application doit donc tenir compte de la capacité de calcul de chaque unité afin de fonctionner correctement et ce dans les meilleures conditions d'utilisation et de performance.
  Contrairement aux interfaces *Homme-Machine* qui doivent impérativement être exécutées sur la machine de déploiement de l'application, les services dédiés aux calculs peuvent être distribués sur l'infrastructure disponible si besoin est. Ce besoin peut se traduire par l'impossibilité de déployer l'application dans son intégralité sur la machine concernée.
  Les ressources liées à la capacité de calcul des machines doivent être prises en compte au moment du déploiement de l'application. Par exemple, lorsque la taille mémoire disponible sur la machine de déploiement est insuffisante, l'application peut se partitionner automatiquement et se déployer sur plusieurs nœuds de l'infrastructure distribuée en fonction de la mémoire disponible sur chaque nœud. De la même manière, si la vitesse du processeur est insuffisante pour effectuer certains calculs, les services nécessitant des ressources CPU plus importantes que celles fournies par la machine de déploiement peuvent être déployés sur d'autres nœuds de l'infrastructure distribuée. Certaines ressources sont fortement liées. Par exemple, pour palier au problème de consommation

d'énergie (si l'application doit être déployée sur une machine à faible autonomie), il peut se révéler indispensable d'isoler les services consommant le plus d'énergies (forte utilisation du CPU, accès disques nombreux, utilisation de périphériques spécifiques, etc.) afin de les répartir sur d'autres nœuds de l'infrastructure et ainsi diminuer la consommation d'énergie.
L'adaptation de la structure des composants, au moment du déploiement, est donc là aussi, nécessaire à l'introduction de cette propriété ;

- *l'adaptation aux variations des performances des machines de déploiement*
  Les performances des machines de déploiement varient tout au long du cycle de vie de l'application. Ces variations peuvent être dues directement à l'exécution de l'application (un service qui crée des données donc diminution de la capacité de stockage de la machine) ou bien à d'autres applications exécutées sur cette même machine. Une solution à ce problème consiste à sonder en permanence les ressources (taux d'utilisation du processeur, stockage disponible, batterie restante) évoluant au cours du temps et à déclencher des phases d'adaptation de l'application si nécessaire. Par exemple, si la capacité de stockage disponible devient critique (en dessous d'un seuil préalablement défini par l'administrateur de l'application), une solution peut consister à réorganiser la structure de l'application de manière à isoler les services les plus consommateurs de cette ressource et à les redéployer sur d'autres nœuds de l'infrastructure disponible.
  Dans ce cas, les composants doivent être capables d'adapter leur structure à leur environnement de déploiement pendant leur exécution ;

- *l'adaptation à l'hétérogénéité des moyens de communication*
  La diversité des réseaux pouvant être accessible par une machine est également importante (réseaux physiques, *Wifi*, *Bluetooth*, infrarouge, etc.). L'application doit donc tenir compte de toutes ces informations afin de fonctionner correctement et ce dans les meilleures conditions d'utilisation et de performance. Cette caractéristique peut se traduire par la mise en place de mécanismes de distribution configurables qui vont permettre d'adapter la communication entre les différents composants de l'application en fonction des types de réseaux disponibles, de leur bande passante et de leur taux d'utilisation. De ce fait, l'adaptation de la structure du composant est nécessaire afin d'assurer la distribution et sa configuration ;

- *l'adaptation aux variations des performances du réseau*
  La bande passante d'un réseau peut varier en fonction du flux de données échangées ainsi que de la puissance du signal s'il s'agit d'un réseau sans fil. Le réseau peut même dans certains cas être inutilisable (trop faible réception du signal, positionnement hors zone de réception, etc.). Ainsi, l'hypothèse de stocker uniquement les composants sur des serveurs dédiés accessibles par l'intermédiaire d'une infrastructure distribuée ne peut être envisagée. Par exemple, considérons le cas de réseaux sans fil : si l'utilisateur se trouve dans une zone qui n'est pas couverte par le réseau, ou bien s'il se produit une panne matérielle du réseau ou de l'un des serveurs, aucun service ne sera disponible. Cependant, dans le cas où l'application ne peut être déployée dans son intégralité sur la machine concernée (ressources limitées), il est nécessaire de déterminer la meilleure stratégie de distribution possible de manière à prendre en compte les caractéristiques du réseau. Par ailleurs, il est préférable de réduire au minimum le nombre de connexions distantes entre les différents composants d'une application afin d'en assurer un fonctionnement optimal. L'adaptation dynamique de la structure des composants est donc nécessaire ;

- *l'adaptation à la disponibilité des services*
  Une application destinée à être exécutée dans un environnement ubiquitaire ou mobile doit être capable de supporter les éventuelles connexions et déconnexions des différentes machines qu'elle utilise. En effet, des services peuvent être inaccessibles (déconnections d'unités de l'infrastructure distribuée, pannes de réseau ou de machines, etc.) pendant des intervalles de temps inconnus. Cependant, ces services peuvent être utilisés par d'autres services de l'application ; ce qui peut engendrer des problèmes. Une application ubiquitaire doit pouvoir continuer à fonctionner malgré la non disponibilité de certains de ses services (*i.e.* l'indisponibilité d'un service utilisé ne doit pas entraîner le « crash » de l'application). Ainsi, elle doit assurer une continuité de service malgré les déconnexions. Par exemple, si l'utilisateur de l'application sort de la zone de couverture du réseau *Wifi* et que celle-ci utilise un service uniquement accessible au travers de ce réseau, l'application doit réagir en fonction de ces nouvelles conditions d'exécution. Elle peut, par exemple rechercher un service similaire au travers d'autres réseaux ou bien proposer à l'utilisateur un fonctionnement en mode dégradé (*i.e.* certains services ne seront pas accessibles à l'utilisateur). Ainsi, la structure des composants doit être suffisamment flexible pour supporter la non disponibilité de services qu'ils utilisent ;

- *l'adaptation à la disponibilité des ressources*
  De la même manière que pour les services, les ressources utilisées par une application ne sont pas figées. Une ressource qu'elle soit physique (imprimante, carte graphique, carte réseau, etc.) ou logicielle (variable, base de données, etc.) peut être indisponible pendant une certaine durée. En effet, les ressources évoluent en permanence tout au long du cycle de vie d'une application. Ces évolutions peuvent avoir de graves conséquences sur le fonctionnement de l'application. Par exemple, la déconnexion intempestive de l'un des périphériques de la machine (possibilité d'ajouter et de supprimer des périphériques à la volée) ou bien une chute de sa capacité de stockage disponible (lié à l'utilisation d'autres applications en parallèle) peut introduire des disfonctionnements dans l'application. Pour palier à ces problèmes et ainsi, assurer une continuité de services, l'application doit être capable de récupérer des informations sur son contexte d'exécution et de réagir si nécessaire à des évolutions ayant un impact notable sur son utilisation. Des stratégies de reprise après panne doivent également être établies. Ainsi, la structure des composants doit être suffisamment flexible pour supporter la non disponibilité de ressources

- *la fiabilité de l'acquisition des données sur le contexte due à l'utilisation de capteur*
  Un des problèmes liés à l'acquisition des données sur le contexte d'exécution de l'application réside dans la fiabilité des informations ainsi récoltées. Si certaines données fournies par le capteur paraissent erronées, l'application ne doit pas en tenir compte. Il est donc nécessaire de déterminer pour chaque élément du contexte des plages des valeurs permettant de déterminer si l'application doit réagir ou non à ce nouveau contexte.

Les propriétés citées ci-dessous sont des propriétés transverses aux applications. De ce fait, elles peuvent être introduites à une application existante de manière automatique. Par ailleurs, ces propriétés peuvent concerner le comportement de l'application ou bien sa structure (voir Tableaux 1.9 et 1.10). En fait, comme nous avons pu le constater, un grand nombre de ces propriétés nécessite l'adaptation de la structure de l'application et des entités logicielles qui la composent, pour être mise en œuvre. Cette stratégie passe par la prise en compte du contexte courant pour déterminer une structure pour l'application et ses constituants adaptée à la situation. De plus, étant donné que dans ce type d'environnement,

les conditions d'exécution d'une application évoluent en permanence, l'adaptation doit être réalisée de manière dynamique et entièrement automatique.

	Caractéristiques	(Propriété$_1$)	(Propriété$_2$)	(Propriété$_3$)	(Propriété$_4$)	(Propriété$_5$)
Type	Fonctionnelle				X	
	Non-fonctionnelle	X	X	X		X
Cible	Comportement				X	X
	Structure	X	X	X		

Propriété$_1$: adaptation aux variations des performances du réseau
Propriété$_2$: adaptation à la disponibilité des services
Propriété$_3$: adaptation à la disponibilité des ressources
Propriété$_4$: transparence des interfaces
Propriété$_5$: prise en compte de la fiabilité de l'acquisition des données sur le contexte

Table 1.9 – Caractéristiques d'une application ubiquitaire (a)

	Caractéristiques	(Propriété$_6$)	(Propriété$_7$)	(Propriété$_8$)	(Propriété$_9$)	(Propriété$_{10}$)	(Propriété$_{11}$)
Type	Fonctionnelle	X	X	X			
	Non-fonctionnelle	X			X	X	X
Cible	Comportement	X	X	X			
	Structure	X			X	X	X

Propriété$_6$: adaptation aux moyens d'exécution
Propriété$_7$: adaptation à l'environnement proche du terminal
Propriété$_8$: prise en compte de la mobilité
Propriété$_9$: adaptation à l'hétérogénéité des moyens de communication
Propriété$_{10}$: adaptation aux variations des performances des machines de déploiement
Propriété$_{11}$: distribution

Table 1.10 – Caractéristiques d'une application ubiquitaire (b)

1.5.2 Les applications sensibles au contexte comme solution à l'adaptation dans les environnements ubiquitaires

Comme nous avons pu le constater précédemment, la principale caractéristique d'une application ubiquitaire est son adaptabilité à différents éléments de son contexte d'exécution. Ainsi, une application ubiquitaire doit être capable de prendre en compte son contexte d'exécution afin de s'adapter à ses variations. Ce type d'application est dit « sensible au contexte »(en anglais *context-aware*).

1.5.2.1 Définition d'une application sensible au contexte

Par définition, une application sensible au contexte est une application qui capte et qui analyse le contexte provenant de différentes sources et qui prend les mesures adéquates en fonction des différents contextes [4]. En fait, un système est dit « sensible au contexte » s'il est capable d'extraire, d'interpréter et enfin d'utiliser les informations contenues dans le contexte dans le but d'adapter, de configurer ou bien de faire évoluer l'application. Il doit avoir la capacité d'analyser les données recueillies qui sont relatives au contexte afin de prendre des décisions pour effectuer l'adaptation. Pour cela, il est indispensable de déterminer les changements de contexte qui peuvent affecter le comportement de l'application. Il faut donc déterminer les éléments de contexte qui sont pertinents pour l'application que l'on doit adapter ou reconfigurer. Il est nécessaire de distinguer deux types de contextes qui peuvent affecter le comportement d'une application : celui qui est en permanence pertinent et celui qui est pertinent pour des valeurs

particulières. Les décisions doivent être prises en tenant compte de ces données. Dey et Abowd proposent dans [2] une définition différente qui fait apparaître les besoins de l'utilisateur de l'application :

> *« un système est context-aware s'il utilise le contexte pour fournir à l'utilisateur des informations et/ou des services appropriés dans le sens adapté à la tâche de l'utilisateur. »*

Cette définition est plus générale que les précédentes. Elle met en valeur l'objectif principal de ce type d'application à savoir améliorer les interactions *Homme-Machine*.

En fait, une application sensible au contexte a pour objectif principal d'adapter son comportement (*i.e.* adaptation ou configuration) selon l'environnement d'exécution. Sa principale qualité est la prise de conscience du contexte c'est-à-dire son aptitude à capturer le contexte, à l'analyser et à réagir selon celui-ci. Cette propriété est nécessaire à la réalisation d'applications destinées à être exécutées dans des environnements ubiquitaires ou mobiles car le contexte d'utilisation peut changer très rapidement et assez fréquemment. Cette prise en compte du contexte peut se faire à différent moment dans le cycle de vie de l'application : au déploiement (*i.e.* automatisation des phases de déploiement) ou pendant l'exécution (*i.e.* détection et analyse permanente du contexte).

1.5.2.2 La notion de contexte

Afin de construire des applications capables de s'adapter à leur environnement d'exécution, il est indispensable de définir le plus précisément possible la notion de contexte d'une application.

Le contexte tel qu'il est généralement défini dans les dictionnaires décrit l'ensemble des circonstances dans lesquelles se déroule un évènement. Cette définition, telle quelle est formulée, ne suffit pas pour être utilisée dans le domaine de l'informatique ubiquitaire. Ainsi, de nombreuses définitions ont été proposées dans la littérature.

La majorité des définitions du terme « contexte » que l'on retrouve dans les travaux du domaine se basent sur une énumération des éléments [7, 78] correspondant à son contenu (identité, informations spatiales, informations temporelles, informations sur l'environnement, situation sociale, ressources proches, disponibilité des ressources, mesures physiologiques, activités de l'utilisateur, etc.). Ces informations concernent généralement l'état et la localisation de l'utilisateur, l'équipement utilisé, infrastructure du réseau et l'environnement externe. Par exemple, voici la définition proposée par Brown, Bovey et Chen en 1997 dans [36] :

> *« Le contexte est la localisation, l'identité des personnes proches de l'utilisateur, l'heure, le jour, la saison, la température, etc. »*

Les catégories de contexte que l'on retrouve généralement dans les définitions proposées dans la littérature sont les suivantes :

- les caractéristiques techniques du support d'exécution (processeur, mémoire, périphériques, etc.), de l'environnement logiciel (système d'exploitation, etc.) et du réseau (bande passante, coût de communication, etc.),
- l'utilisateur (profil, activité, etc.),
- l'environnement (temps, lumière, bruit, etc.),
- la localisation (position, orientation, etc.),
- le moment (heure, date, saison, période, etc.).

Une définition plus générale de la notion de contexte pourrait être celle proposée par Dey *et al* dans [2] qui est la définition la plus communément acceptée dans le cadre de travaux sur la prise en compte et la modélisation du contexte d'une application :

> *« le contexte regroupe toutes les informations qui peuvent être utilisées pour caractériser la situation d'une entité. Une entité peut être une personne, un lieu, ou un objet qui est considéré comme ayant un lien avec l'interaction entre un utilisateur et une application incluant l'utilisateur et l'application. »*

Généralement, le contexte est limité aux informations qui peuvent être détectées. On parle alors de contexte d'utilisation. Il correspond à tout attribut détectable et pertinent de l'environnement d'exécution. Nous pouvons citer une autre définition du mot contexte donnée par Ayed *et al.* dans [5] :

> *« le contexte d'utilisation d'une application contient tout attribut détectable et pertinent de son environnement d'exécution. Ce contexte peut représenter les capacités du terminal utilisateur, sa connexion au réseau, son environnement externe, sa localisation, son profil, ses préférences, etc. »*

Ces attributs détectables et pertinents de l'environnement d'exécution constituent ainsi les éléments que l'application doit prendre en compte pour définir des points de variabilité adaptables suivants leurs valeurs.

1.5.2.3 Le traitement du contexte

Les applications sensibles au contexte sont des applications dotées de mécanismes leurs permettant de prendre en compte leur contexte d'exécution. Leur processus d'adaptation comporte quatre étapes :

1. *collecte d'informations relatives au contexte*
   L'acquisition des informations contextuelles se fait soit par l'utilisation de capteurs simples (thermomètre, anémomètre, etc.), soit par des gestionnaires de ressources (Niveau de la batterie, etc.) ou même des outils de supervision plus sophistiqués ;

2. *analyse de ces informations*
   L'analyse des données collectées au cours de l'étape précédente doit permettre d'extraire le contexte pertinent (*i.e.* contexte qui affecte le comportement de l'application pendant son déploiement ou bien au cours de son exécution) ;

3. *prise de décisions*
   En fonction des indications récoltées au cours de l'étape d'analyse, l'adaptateur doit établir une stratégie d'adaptation. Celle-ci peut consister par exemple à choisir l'implémentation des éléments qui la compose, la structure de l'application ou bien les machines d'implantation de ses composants les mieux adaptés à la situation ;

4. *action*
   La dernière étape du processus consiste à mettre en œuvre les décisions prises dans l'étape précédente. Les actions à réaliser peuvent être diverses : modification de paramètres (*i.e.* reconfiguration), adaptation de l'application (manipulation de code, transfert de code, remplacement de services, redirection de messages, etc.) évolution de l'application (ajout / suppression de fonctionnalités, etc.), présentation des données à l'utilisateur, etc.

1.5.2.4 Étude comparative des approches de création d'applications sensibles au contexte

Afin d'étudier les approches de création d'applications sensibles au contexte, proposées dans la littérature, nous avons défini un certain nombre de critères de comparaison nous permettant de classifier les travaux existants.

Critères de comparaison des approches existantes Nous avons défini six critères de comparaisons pour classifier les approches de création d'applications sensibles au contexte :

- *l'environnement d'exécution*
  De la même manière que nous l'avons vu pour les approches d'adaptation de composants et d'applications conçues à base de composants, la sensibilisation au contexte dépend de l'environnement d'exécution de l'application. Ce dernier peut être un environnement mono-machine, un environnement distribué, un environnement mobile ou bien ubiquitaire ;

- *le contenu du contexte*
  Comme nous l'avons dit précédemment, il n'existe pas de standard pour définir le contexte. De ce fait, les travaux menés dans le cadre d'adaptation à un contexte comme c'est le cas pour les applications ubiquitaires, définissent eux même leur contexte et ce qu'il contient. Ainsi, le contexte peut contenir différents éléments tels que :

  1. *les caractéristiques techniques* : elles regroupent l'ensemble des informations concernant le support d'exécution de l'application (par exemple, la puissance du processeur, capacité de stockage, bande passante du réseau, etc.) ;

  2. *l'environnement proche de l'interaction entre l'application et l'utilisateur* : il regroupe toutes les informations concernant l'environnement d'exécution de l'application (par exemple, le bruit, température, luminosité, qualité de l'air, météo, humidité, etc.) ;

  3. *la localisation* : elle regroupe toutes les informations concernant la position géographique de l'utilisateur suivant un référentiel préalablement défini (par exemple, GPS, RF, etc.) ;

  4. *le temps* : il contient les informations relatives le moment d'utilisation de l'application (date, heure, etc.) ;

  5. *le profil de l'utilisateur* : il regroupe toutes les informations relatives à l'utilisateur de l'application telles que sa situation sociale, ses mesures physiologiques, ses activités, etc. ;

  6. *autres* : d'autres éléments du contexte peuvent également être pris en compte tels que la vitesse de l'utilisateur au moment de son interaction avec l'application, son accélération, son orientation, les évènements d'actualité, etc. Ces éléments sont généralement spécifiques à une application. Ils ne sont généralement pas pris en compte de manière automatique.

- *les propriétés de l'environnement prises en compte*
  Comme nous avons pu le constater précédemment, tout environnement d'exécution dispose de

caractéristiques spécifiques. Par exemple, concernant les environnements ubiquitaires, les applications doivent être capable de fonctionner malgré les variations des ressources disponibles, les connexions/déconnections des machines de déploiement, etc. De ce fait, un critère de classification des approches de création d'applications sensible au contexte est relatif à la prise en compte des propriétés de l'environnement dans lequel ils se positionnent ;

- *la modélisation du contexte*
  Le contexte d'une application peut être modélisé de différentes manières. Les techniques de modélisation les plus utilisées dans la littérature sont les suivantes :

  1. *données brutes acquises à la volée*
     Dans ce cas, les informations contextuelles sont directement accessibles à l'application (*i.e.* interrogation directe des capteurs). Aucun historique sur les données n'est conservé. Cette stratégie peut être utilisée pour la prise en compte de peu d'éléments contextuels et dont la prise en compte ne se fait que peu fréquemment. Ces données n'ont généralement pas besoin ni d'être interprétées, ni d'être agrégées ;

  2. *association (Description/Valeur)*
     L'une des premières manières de modéliser le contexte fut proposée par Schilit en 1993 [111]. Elle consiste à créer des associations contenant d'une part le nom de l'élément de contexte et d'autre part, sa valeur. Cette stratégie simpliste permet de stocker le contexte. Cependant, elle peut se révéler rapidement inutilisable dans le cas où le contexte contient un grand nombre d'éléments évoluant en permanence ;

  3. *arborescence*
     Les données contextuelles peuvent être représentées sous forme d'arbre. Cette stratégie permet d'obtenir facilement des vues hiérarchiques sur le contexte. Les éléments sont regroupés en fonction de leur catégorie. Les feuilles représentent les valeurs des éléments de base acquis par l'intermédiaire de capteurs et les nœuds correspondent à des agrégats de données. Pour accéder à un élément du contexte, il suffit alors de fournir son chemin d'accès (*i.e.* branche de l'arbre de modélisation du contexte). Une des techniques couramment utilisée pour mettre en œuvre cette stratégie est la création de données XML interrogeables en utilisant des langages de type *XPath* [47] ;

  4. *architecture en couches*
     Le contexte peut également être modélisé sous la forme d'une architecture en couches [112]. La couche la plus basse est la couche matérielle. Elle contient les capteurs qui vont permettre l'acquisition du contexte. Les autres couches sont chargées d'obtenir des vues plus abstraites sur le contexte de par la mise en œuvre d'outils d'interprétation ou d'agrégation de données qui vont traiter les signaux provenant des couches les plus basses ;

  5. *orientée objet*
     Dans le cadre d'une modélisation orientée objet [65], chaque élément du contexte est représenté par un objet. Les attributs de cet objet correspondent aux propriétés de l'élément concerné. De plus, les différents objets peuvent être reliés entre eux grâce aux relations d'as-

sociation, d'héritage, etc. Ainsi, le contexte pourra être structuré en fonction des dépendances entre ses éléments ;

6. *ontologie*
   Les ontologies peuvent également être utilisées pour modéliser le contexte d'exécution d'une application. Une ontologie est un ensemble de termes hiérarchiquement structurés pour décrire un domaine et pouvant être utilisés comme squelette pour une base de connaissance [122]. Ainsi, une ontologie est un formalisme de représentation permettant de modéliser des entités ainsi que leur relation sémantique. Cette stratégie permet de faciliter la recherche d'information sur le contexte de par l'utilisation de techniques définies dans la communauté web sémantique [66].

La modélisation du contexte est primordiale dans la conception d'applications sensibles au contexte. Cependant, étant donnée la grande diversité des éléments mis en jeu, il est impossible de fournir une modélisation générique du contexte. En fait, le choix de la modélisation du contexte est adapté au domaine d'utilisation de ces données ;

- *le type de sensibilisation au contexte*
  La sensibilisation d'une application à son contexte peut être réalisée de différentes manières. Ainsi, trois types de sensibilisations peuvent être référencés [11] :

  1. le *context-aware* actif : l'application change automatiquement de comportement en fonction des informations captées. Ce type de sensibilisation est notamment utilisé dans le cadre d'approche d'auto-adaptation ;

  2. le *context-aware* passif : l'application présente à l'utilisateur une mise à jour du contexte ou bien des informations sur l'environnement et laisse à l'utilisateur le choix de décider ce que l'application doit faire ;

  3. la personnalisation : l'utilisateur spécifie lui-même ses propres paramètres pour que l'application sache ce que faire si elle est dans telle ou telle situation.

- *le type d'applications cibles*
  Enfin, le dernier critère de comparaison que nous avons choisi est relatif au type d'application destinée à acquérir et à traiter des informations sur le contexte. Ces dernières peuvent être conçues en utilisant différentes technologies telles que l'orienté objet, les composants logiciels, etc.

Étude comparative des approches existantes Nous avons pu constater que la majorité des travaux existants relatifs à la conception d'applications sensibles au contexte (voir Tableau 1.11) laisse entièrement à la charge de l'administrateur de l'application la définition du contexte dit pertinent (*i.e.* ayant un impact sur l'application) bien que le type de sensibilisation reste du « context-aware » actif. Cette stratégie est notamment due aux approches d'adaptation qui permettent pour la plupart uniquement d'adapter le comportement comme nous l'avons vue dans la section précédente. En effet, le comportement d'une application étant spécifique à celle-ci, il est impossible de prévoir de manière générique les évènements

qui vont l'influencer. Cependant, concernant la structure d'une application, il est possible de déterminer de manière générique certains éléments du contexte qui peuvent l'influencer (voir Chapitre 4).

Travaux	Environnement	Éléments du contexte	Propriétés prises en compte	Modélisation du contexte	Applications cibles
ACEEL [44]	Ubiquitaire Mobile	Caractéristiques techniques	Variation des ressources disponibles	Modélisation objet	Composants logiciels
Cyberguide [1]	Mobile	Localisation	Variation de la localisation de l'utilisateur	Données brutes	n.s.
Context-Fabric [68]	n.s.	n.s.	-	Modèle spécifique	n.s.
Context-Toolkit [110]	n.s.	n.s.	-	Modèle en couches	n.s.
Wildcat [47]	Ubiquitaire	n.s.	-	Arborescence XML	Composants logiciels
Casa [95]	Ubiquitaire Mobile	Caractéristiques techniques	Variation des caractéristiques du terminal	Modèle en couches	Composants logiciels
Plasma [81]	Ubiquitaire Mobile	Caractéristiques techniques	Variation des caractéristiques du terminal Variation de sa connexion réseau	Données brutes	Composants logiciels
Molène [114]	Mobile	Caractéristiques techniques	Variation des caractéristiques du réseau	Modèle en couches	Composants logiciels
CADeComp [3]	Mobile	Utilisateur Caractéristiques techniques Autres	Variation de la localisation de l'utilisateur Variation de sa connexion réseau Variation des caractéristiques du terminal Variation de l'environnement	Arborescence XML	Composants logiciels
Concerto [46]	Systèmes distribués Grids	Caractéristiques techniques	Variation des ressources disponibles	Modélisation objet	Composants logiciels
Rossi *et al* [107]	Mobile	Utilisateur Environnement Localisation Temps	Variation de la position de l'utilisateur Variation de l'environnement	Données brutes	Objets

Table 1.11 – Tableau comparatif des approches de création d'applications sensibles au contexte

1.5.3 Bilan de l'étude de l'adaptation dans les environnements ubiquitaires

L'étude des caractéristiques d'une application ubiquitaire nous a permis de les classifier en deux catégories (voir Tableaux 1.9 et 1.10) : d'une part, celles visant à adapter les services des composants comme la transparence des interfaces ou bien la prise en compte de l'environnement et d'autre part, celles liées au déploiement du composant comme la distribution ou l'adaptation aux ressources disponibles. La première catégorie (*i.e.* adaptation des services) a été largement étudiée dans la littérature. Concernant la deuxième catégorie relative au déploiement, nous avons constaté que ce dernier dépendait fortement de la structure des composants concernés. En effet, un composant composite peut être déployé de manière distribuée (*i.e.* les sous-composants peuvent être répartis sur différents nœuds de l'infrastructure distribuée disponible) alors qu'un composant monolithique ne peut être déployé sur une machine que si les ressources disponibles sur cette machine sont suffisantes.

Aussi, dans un tel d'environnement, il est nécessaire de pouvoir adapter la structure de tout type de composants logiciels (monolithiques et composites) afin de leur garantir une continuité de service et de leur maintenir une qualité de service.

1.6 Conclusion

Le positionnement de la problématique d'adaptation logicielle par rapport à celle liée à la réutilisation nous a permis de montrer le besoin d'introduire de la variabilité au niveau des entités logicielles et des applications logicielles afin de pouvoir les adapter et ainsi afin d'augmenter leur capacité d'être réutilisées.

Nous avons pu alors constater que beaucoup d'approches ont été proposées dans la littérature pour répondre à ce besoin. Cependant, nous avons montré que la majorité des approches existantes se focalisent sur l'adaptation du comportement des composants et que celles qui sont consacrées à l'adaptation de leur structure considère comme unité de base (ne pouvant être adaptés que par configuration de leurs attributs) les composants monolithiques bien que leur adaptation structurelle puisse se révéler nécessaire

dans beaucoup de cas et notamment pour des applications destinées à être exécutées dans des environnements ubiquitaires. De plus, la grande majorité des travaux existants ne prend pas en considération les composants existants. En fait, ils ne proposent que des modèles ou des méta-modèles permettant de construire des composants adaptables ; ce qui se révèle très limitatif étant donné que la réutilisation à grande échelle est l'objectif principal de l'ingénierie des composants logiciels. Aussi, nous avons pu constater qu'il n'existe pas d'approche permettant d'adapter la structure de tout type de composants existants.

En se basant sur ces considérations, notre objectif dans la suite de ce manuscrit est de proposer une approche permettant la restructuration de composants logiciels existants y compris les composants monolithiques. Cette propriété aura pour objectif de répondre à des besoins en termes de déploiement flexible des composants logiciels (*i.e.* adaptation à l'infrastructure de déploiement) ou de chargement flexible de services en vue d'un fonctionnement en mode dégradé (*i.e.* adaptation aux ressources disponibles).

CHAPITRE 2

Adaptation structurelle de composants logiciels : contexte et problématique

2.1 Introduction

Comme nous avons pu le voir précédemment, la réutilisation d'entités logicielles existantes est de plus en plus présente dans la conception de nouvelles applications. La technologie à base de composants logiciels constitue un énorme pas pour favoriser la réutilisation. Cependant, à cause de la grande multiplicité des environnements de déploiement, il est nécessaire, dans la majorité des cas, d'adapter les composants réutilisés afin d'exploiter au mieux leurs capacités.

Néanmoins, dans le chapitre précédent dédié à l'état de l'art, nous avons pu constater que les approches d'adaptation existantes dans le domaine de l'ingénierie des composants logiciels présentent des limitations. En effet, les approches permettant d'adapter la structure de composants ne permettent pas de tirer pleinement partie de ce type d'adaptation. Notamment, le cas de composants existants reste problématique.

Par ailleurs, nous avons pu constater que, dans certains types d'environnements, tels que les environnements ubiquitaires, le besoin en adaptation structurelle est très fort. En effet, à cause des ressources limitées, les composants doivent être capables d'adapter leur structure afin de réaliser de l'adéquation entre l'architecture logicielle et l'architecture matérielle. Cette adaptation va ainsi permettre de garantir une continuité et une qualité de services à l'application qui les contient.

De ce fait, nous proposons de combler les insuffisances constatées dans les approches existantes en proposant une approche d'adaptation structurelle de composants logiciels appelée SCORPIO (*Software COmponent stRuctural adaPtatIOn*).

Ce chapitre présente une vue générale de l'approche d'adaptation structurelle. Dans un premier temps, nous introduisons notre cadre d'étude. Puis, nous présentons notre proposition et les différents concepts qui lui sont associés. Ensuite, nous détaillons les motivations qui nous ont poussées à développer notre approche à travers la présentation de quelques applications possibles de l'adaptation structurelle. Nous présentons alors les fondements de la solution retenue. Enfin, nous introduisons le cas d'étude que nous avons choisi afin d'illustrer notre approche.

2.2 Cadre de l'étude

Notre étude se positionne dans le cadre des composants logiciels, plus particulièrement ceux existants tels que les COTS [67]. De plus, nous nous sommes intéressés à appliquer notre approche pour l'adaptation d'applications dans des environnements ubiquitaires. Les travaux menés dans ce cadre font partis du projet Mosaïques auquel nous avons participé.

2.2.1 Cadre technologique : les composants logiciels

Notre approche d'adaptation porte sur les composants logiciels. En effet, comme nous l'avons évoqué précédemment, le fort développement du paradigme de « composants logiciels » ainsi que leur capacité intrinsèque à mettre en œuvre la réutilisation, nous ont poussé à développer notre approche sur ce type particulier d'entités logicielles afin d'accroître leur réutilisabilité. En effet, le principal avantage de l'ingénierie des composants logiciels réside dans le fait qu'il soit possible de concevoir des applications à partir d'entités logicielles de haut niveau. De ce fait, pour maximiser leur capacité de réutilisation, ces entités doivent être dotées de caractéristiques leur permettant de s'adapter ou d'être adaptées facilement à de nouveaux contextes d'utilisation. Aussi, l'adaptation doit à la fois concerner leur comportement et leur structure.

Par ailleurs, nous avons pu constater dans le chapitre précédent que très peu d'approches proposées dans la littérature traitent de l'adaptation de la structure de composants existants tels que les COTS [67]. En effet, les approches qui proposent d'adapter la structure de composants existants ne sont applicables que sur des composants sous forme composite : les composants monolithiques ne sont pas considérés comme structurellement adaptables. De ce fait, notre objectif va être d'intégrer à des composants existants (monolithiques et composites) des mécanismes leurs permettant d'être adaptés ou de s'adapter à des conditions nouvelles d'utilisation ; la difficulté étant que les personnes réalisant l'adaptation d'un composant sont différentes des concepteurs de celui-ci.

2.2.2 Cadre applicatif : les environnements ubiquitaires

Avec l'arrivée de la miniaturisation, de la mobilité et de l'accès sans fil, l'informatique ubiquitaire (en anglais *Ubiquitous Computing*) est l'un des enjeux de demain. Ce nouveau domaine de recherche a pour objectif d'intégrer les technologies informatiques au quotidien de l'homme le plus simplement possible. Plus précisément, elle devrait rendre familier et instinctif l'outil informatique et en faciliter l'utilisation dans de nombreux domaines tels que l'information, la formation, le transport ou encore la médecine.

Comme nous l'avons vu précédemment, une application dite « ubiquitaire » doit présenter un ensemble de caractéristiques précises qui lui permet d'être capable d'être déployée dans ce type d'environnement tout en garantissant une continuité de service et une qualité de service. De nouvelles méthodes de conception d'applications ainsi que de nouvelles infrastructures logicielles sont alors nécessaires pour permettre la prise en compte des contraintes liées à ces environnements ubiquitaires. Pour cela, de nombreux projets de recherche ont vu le jour.

Le projet MOSAIQUES (MOdèles et InfraStructures pour Applications ubIQUitairES) [52] auquel nous avons participé en fait partie. Son objectif est de définir un cadre de développement (méthodologie et outillage) pour la définition d'applications fonctionnant dans un environnement ubiquitaire. Ce projet étudie plus particulièrement la prise en compte de la notion d'adaptabilité à l'exécution dans les applications et les infrastructures logicielles, qui fait partie des caractéristiques principales d'une application ubiquitaire.

Ce travail passe tout d'abord par la définition d'un support méthodologique pour la construction d'applications ubiquitaires. Une telle méthodologie doit permettre d'intégrer à une application les caractéristiques de l'ubiquité que nous avons recensées dans le chapitre 1. Or, comme nous avons pu le constater, certaines de ces caractéristiques sont étroitement liées à la structure de l'application. Par exemple, la structure de l'application doit être suffisamment flexible pour s'adapter automatiquement à l'infrastructure matérielle disponible. De ce fait, nous avons proposé d'appliquer l'adaptation structurelle à des composants destinés à être exécutés dans des environnements ubiquitaires. Cette stratégie n'est qu'une première étape pour la construction d'applications ubiquitaires car d'autres caractéristiques, notamment liées au comportement de l'application, sont à prendre en considération. Les autres étapes sont fournies par le projet.

2.3 Proposition : l'adaptation structurelle de composants logiciels

Notre principal objectif est d'adapter la structure d'un composant logiciel existant en vue de le réutiliser dans des conditions nouvelles ou évolutives. Cette stratégie passe par l'intégration d'une propriété au composant lui permettant de modifier sa structure tout en préservant ses services et son comportement : l'adaptabilité structurelle.

2.3.1 Les concepts

L'adaptation structurelle fait référence à une mise à jour de la structure d'un composant tout en préservant ses services et son comportement.

La structure d'un composant est un graphe dont les nœuds représentent les entités structurelles contenues dans le composant et les arêtes représentent les relations structurelles entre ces entités.

Une entité structurelle d'un composant fait référence à un élément identifiable de manière unique faisant partie de la structure du composant (*i.e.* faisant partie de son espace de noms). Les entités structurelles peuvent être externes ou internes. En fait, les entités structurelles externes sont les ports, les interfaces, les classes ou les méthodes correspondant aux services fournis par ces interfaces. Les entités structurelles internes sont les méthodes ou les classes internes. Il existe un type particulier d'entités structurelles appelé ressource logicielle.

Une ressource logicielle désigne n'importe quelle entité structurelle faisant partie de la définition du composant à adapter, qui est identifiable de manière unique, manipulable et possédant un état qui peut être mis à jour et dont la persistance est supérieure à celle de la méthode dans laquelle elle est utilisée. Par exemple, pour des composants logiciels implémentés en utilisant l'approche objet et où chaque composant est un objet, un attribut défini dans cette implémentation est une ressource.

Les entités structurelles peuvent être composées d'autres entités structurelles (par exemple, les ports sont composés d'interfaces et les interfaces sont composées de services) ou bien hériter des propriétés d'autres entités structurelles de même type.

Ainsi, les relations structurelles entre les entités structurelles peuvent être la composition (par exemple, un port est composé d'interfaces, une interface est composée de méthodes, etc.) ou l'héritage (par exemple, un composant peut hériter d'un autre composant, une interface peut hériter d'une autre interface, etc.).

2.3.2 Les différentes facettes de l'adaptation structurelles

L'adaptation structurelle peut concerner la structure externe d'un composant (ses ports, ses interfaces, etc.) et/ou sa structure interne (ses sous-composants, ses classes d'implémentation, etc.). Ainsi, nous définissons deux types d'adaptation structurelle : l'adaptation structurelle externe et l'adaptation structurelle interne.

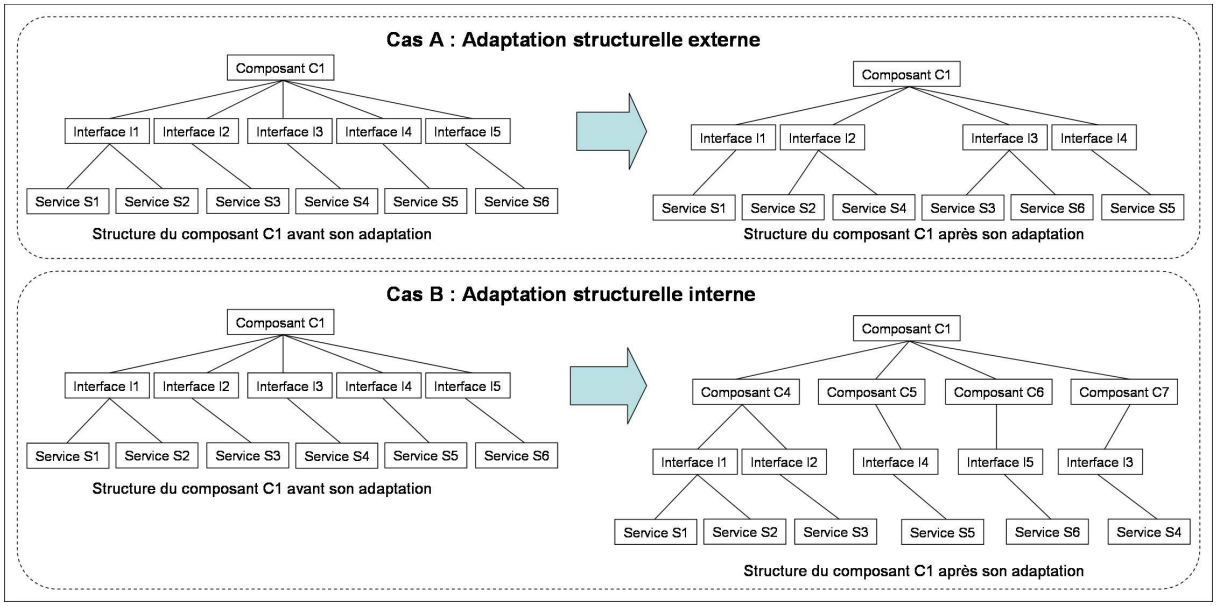

Figure 2.1 – Adaptation structurelle de composants logiciels

2.3.2.1 Adaptation structurelle externe

L'adaptation structurelle d'un composant peut consister à mettre à jour l'ensemble de ses interfaces en modifiant le contenu de chacune d'entre elles ou bien en restructurant le contenu des ports du composant.

Dans la figure 2.1, le cas A montre un exemple d'adaptation structurelle externe d'un composant appelé C_1. Sa structure externe est modifiée comme suit : l'interface S_1 fournit uniquement le service S_1 ; les services S_2 et S_4 sont regroupés au sein de l'interface I_2 ; les services S_3 et S_6 sont fournis par l'interface I_3 ; enfin, l'interface I_4 fournit le service S_5. L'interface I_5 du composant initial ne fournissant plus aucun service du composant C_1 est alors supprimée.

Ainsi, l'adaptation structurelle externe peut être utilisée pour faciliter l'assemblage de composants logiciels ayant des structures externes hétérogènes. En fait, l'assemblage de deux composants ou l'intégration d'un composant dans une application existante peut nécessiter la restructuration des interfaces fournies et requises de l'un des deux composants. Cette opération est nécessaire dès lors que l'un des composants fournit tous les services requis par l'autre, bien qu'ils ne soient pas structurés de manière à pouvoir s'assembler.

2.3.2.2 Adaptation structurelle interne

L'adaptation structurelle peut également concerner la structure interne d'un composant logiciel. Par exemple, elle pourrait résider dans la fragmentation du composant à adapter en plusieurs sous-composants fournissant chacun un ensemble de services parmi ceux définis par le composant initial.

Dans la figure 2.1, le cas B montre un exemple d'adaptation structurelle interne d'un composant appelé C_1. Sa structure interne est modifiée comme suit : quatre nouveaux sous-composants (C_4, C_5, C_6 et C_7) issus de la fragmentation du composant C_1 ont été générés ; C_4 fournit les interfaces I_1 et I_2 ; C_5 fournit l'interface I_4 ; C_6 fournit l'interface I_5 ; et enfin, C_7 fournit l'interface I_3.

Ainsi, l'adaptation de la structure interne d'un composant peut être utilisée pour mettre en œuvre des stratégies de déploiement flexible. En effet, ce type d'adaptation peut être utilisé afin d'organiser les services du composant à adapter dans des sous-ensembles distincts ; chacun de ces sous-ensembles étant défini dans un nouveau composant généré par la fragmentation du composant initial. Ensuite, les composants ainsi générés peuvent être déployés sur une ou plusieurs machines (*i.e.* infrastructure distribuée). Cette stratégie peut être très utile dans le cadre d'une politique de répartition de charges.

2.4 Motivations de notre proposition : les applications de l'adaptation structurelle

De manière générale, l'adaptation d'un composant logiciel permet d'augmenter sa réutilisabilité en le rendant opérationnel dans des contextes différents. Ainsi, adapter la structure d'un composant logiciel sans altérer son comportement, permet de répondre à cet objectif général. Notre approche se distingue par la prise en compte d'autres besoins tels l'adaptation pour une meilleure adéquation entre les architectures logicielles et matérielles ou la restructuration des composants et ainsi de leurs applications pour être déployés suivant un schéma précis.

2.4.1 Adaptation pour une meilleure adéquation entre architectures logicielles et matérielles

Dans le cadre d'environnements à ressources contraintes, il peut être impossible de déployer certains types de composants tels que des composants monolithiques, sur des unités car les ressources fournies sont insuffisantes. Ce constat est du à la structure de ce type de composants qui ne permet pas de le déployer de manière flexible. En effet, il est impossible de déployer séparément une partie d'un composant, définissant un ensemble de services, si cette partie n'est pas structurellement identifiable et ainsi séparable. Or, l'adaptation structurelle interne peut être utilisée pour réorganiser la structure d'un composant logiciel de manière à isoler des morceaux du composant identifiables et séparables.

Ainsi, l'adaptation structurelle interne peut être utilisée afin de réaliser l'adéquation entre la structure d'un composant et l'architecture matérielle disponible. Pour cela, deux approches peuvent être envisagées : d'une part, le déploiement flexible du composant/de l'application en fonction des ressources disponibles ; d'autre part, le chargement flexible du composant en fonction des ressources disponibles.

- *L'adéquation par la distribution du composant/de l'application en fonction des ressources disponibles*
  La première stratégie possible pour réaliser l'adéquation entre l'architecture logicielle et l'architecture matérielle disponible réside dans le déploiement flexible des composants logiciels qui constituent l'application. Cette stratégie consiste à distribuer les services fournis par un composant en

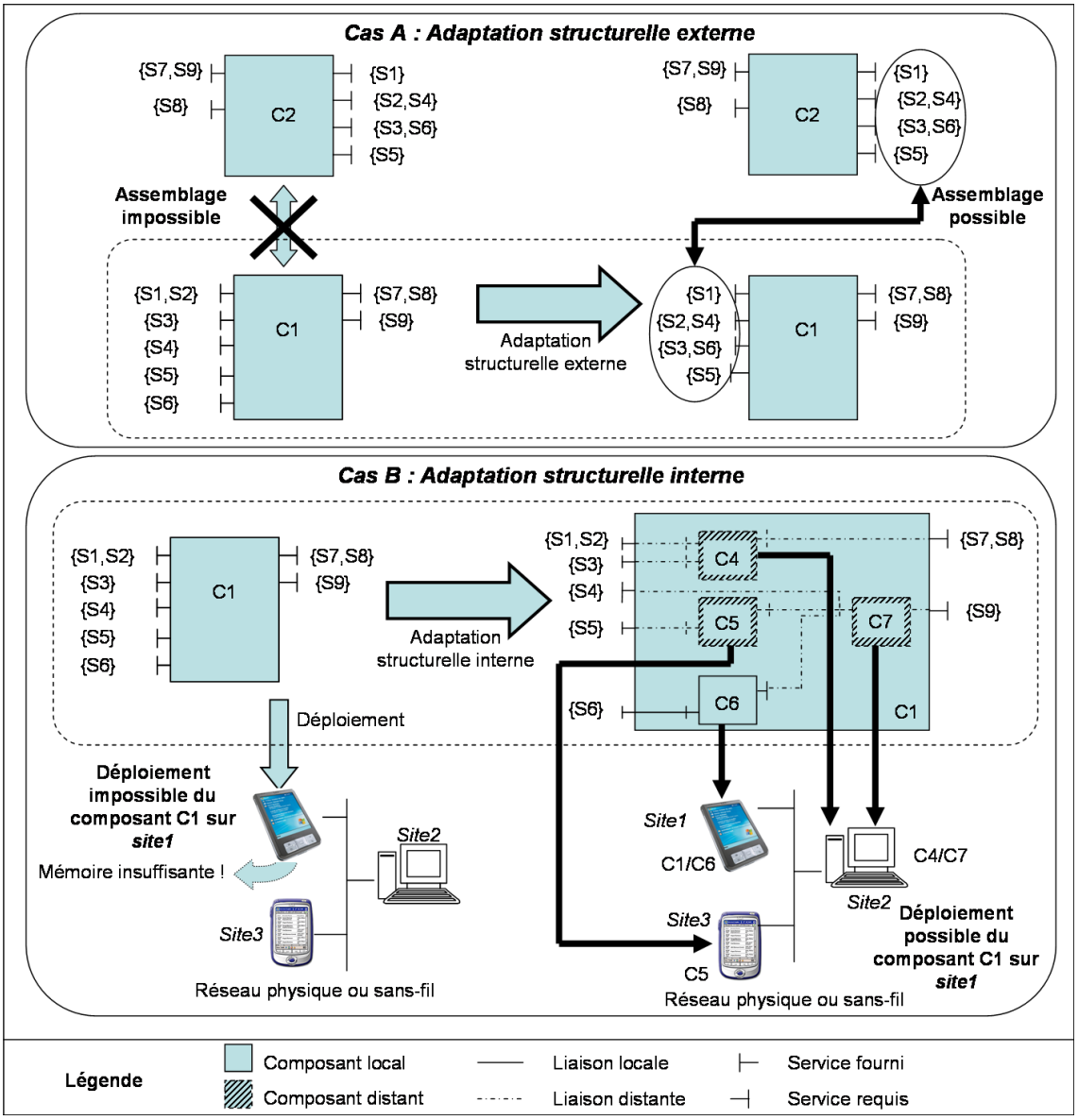

Figure 2.2 – Les motivations de l'adaptation structurelle de composants logiciels

fonction de son contexte d'exécution (voir Figure 2.2, cas B). Le contexte désigne aussi bien les caractéristiques techniques de l'infrastructure de déploiement (*i.e.* architecture matérielle) que les besoins liés à son utilisation ou à son comportement. Diverses stratégies peuvent être adoptées en fonction de la situation. Par exemple, dans le cadre d'environnements ubiquitaires, où les déconnections de la machine de l'utilisateur peuvent être fréquentes, il serait judicieux de déployer un maximum de services sur cette machine puis privilégier les nœuds les plus accessibles (en termes de meilleure bande passante et d'occupation du réseau) ou bien les serveurs fixes pour déployer les autres services en fonction des ressources disponibles.
Cette adaptation sera rendue possible en utilisant notre approche d'adaptation structurelle interne qui va nous permettre de redéfinir et de recréer, suivant les besoins du déploiement, les différentes entités structurelles à manipuler de manière séparée pour pouvoir leur appliquer des procédures de déploiement et de re-déploiement différentes.
Cette stratégie permet principalement de garantir la continuité de service du composant concerné (*i.e.* adaptation pour la flexibilité). Cependant, elle peut également être utilisée pour améliorer ses performances (*i.e.* adaptation perfective). En effet, le choix des machines de déploiement pour les différents services fournis par le composant peut permettre de diminuer le temps d'exécution de certains services étant donné que les services peuvent être déployés sur des machines dotées de caractéristiques techniques nettement supérieures à celles de la machine de déploiement initiale (par exemple, les nouvelles machines de déploiement peuvent être dotées d'une meilleure vitesse de processeur). De ce fait, si le gain obtenu n'est pas annihilé par les pertes liées à la distribution des services, les performances du composant sont grandies.

- *L'adéquation par le chargement flexible du composant/de l'application en fonction des ressources disponibles*
  Dans le cas précédent, nous avons évoqué la possibilité de réaliser un déploiement flexible de composants logiciels dans un environnement à ressources limitées. Cependant, dans le cas d'un environnement mono-machine, les services qui ne peuvent pas être déployés sur la machine de l'utilisateur ne pourront être disponibles. De ce fait, il est nécessaire d'intégrer au composant des mécanismes lui permettant de charger et de décharger des services en fonction des besoins liés à son utilisation et des ressources qu'il dispose (*i.e.* adaptation adaptative).

Par exemple, dans la figure 2.2, le cas B montre un composant logiciel (C_1) qui ne peut pas être déployé sur le site 1 car les ressources matérielles telles que la capacité mémoire, la vitesse du processeur, etc. de la machine correspondant à ce site sont insuffisantes. Ainsi, l'adaptation de ce composant par la fragmentation de sa structure interne permet sa transformation en un composant composite dont les sous-composants pourront être répartis sur différents sites disponibles au travers de l'infrastructure distribuée. Ainsi, les services fournis par le composant C_1 seront disponibles.

Cependant, cette flexibilité de la structure des composants ne doit pas être introduite à n'importe quel prix. Les performances de l'application doivent être maintenues ou faiblement dégradées (à cause de la distribution). De ce fait, il est indispensable de proposer des mécanismes de distributions adaptés et de prendre en compte les éléments influençant les performances d'un composant tel qu'une sélection judicieuse des machines de déploiement de chaque service. Ainsi, le choix de la nouvelle structure du composant doit être effectué en prenant en compte de nombreux paramètres tels que les caractéristiques techniques de chaque site de déploiement concerné (tels que la capacité de stockage, la vitesse du processeur, etc.), les caractéristiques des moyens de communication entre les différents sites de déploiement

(tels que le type de réseau, la bande passante, etc.) ou bien la disponibilité des sites de déploiement (*i.e.* prise en compte des risques de déconnexions possibles des sites de déploiement d'un composant). En fait, ce choix va conditionner les performances de l'application car les pertes liées à la distribution du composant peuvent être en partie compensées par les performances des nouveaux sites de déploiement choisis.

2.4.2 Adaptation pour une meilleure adéquation des applications à base de composants à des schémas d'utilisation

L'adaptation de la structure d'un composant peut également être utilisée pour réaliser l'adéquation des applications à base de composants à des schémas d'utilisation. Cette adéquation peut prendre deux formes :

- *l'adéquation entre l'architecture logicielle et la configuration des utilisateurs*
  L'adaptation de la structure d'un composant peut également être utilisée pour réaliser l'adéquation entre l'architecture logicielle de l'application et la configuration de ses utilisateurs. Par exemple, dans le cadre d'une application muti-utilisateurs pour laquelle ses différents utilisateurs potentiels peuvent se trouver géographiquement répartis, il est nécessaire d'adapter la structure de l'application afin de déployer les services en fonction des nœuds de l'infrastructure servant de points d'interaction entre l'application et les utilisateurs. Une telle stratégie permet notamment de limiter les problèmes liés aux déconnexions des machines des utilisateurs et de réduire les coûts liés aux communications distantes (*i.e.* amélioration des performances) ;

- *l'application de schémas de déploiement précis*
  Dans certaines situations, il peut être nécessaire de prévoir des schémas de déploiements précis. Par exemple, les parties d'une application ou d'un composant destinées à manipuler des données confidentielles doivent être déployées sur des unités spécifiques sécurisées. Cependant, pour des raisons de coût en termes de maintenance et de performance, toutes les parties de l'application ne peuvent être déployées sur une telle machine. De ce fait, il est nécessaire d'instaurer un schéma de déploiement précis relatif aux différentes parties d'une application et à ses sites de déploiement.
  L'adaptation structurelle pourrait alors répondre à de telles attentes. En effet, elle peut être utilisée pour isoler certaines parties d'un composant afin de les déployer sur des nœuds spécifiques de l'infrastructure de déploiement. Ainsi, si nous reconsidérons notre exemple, les parties relatives à la gestion de données pourront être isolées de manière à pouvoir les déployer sur des serveurs sécurisés.

2.4.3 Adaptation perfective et corrective pour l'introduction de nouvelles propriétés aux composants

L'adaptation de la structure d'un composant peut également être utilisée pour intégrer de nouvelles propriétés aux composants. A titre d'exemple, nous pouvons citer l'intégration des propriétés suivantes :

- *l'intégration de la propriété d'interopérabilité*
  L'assemblage d'un composant logiciel avec d'autres composants nécessite la vérification de l'adéquation des services fournis et requis mutuellement par ces composants. Cependant, dans certains cas, malgré que les services requis par l'un puissent être fournis par l'autre, l'assemblage de deux

composants ne peut être réalisé. En fait, cette impossibilité d'assemblage est due au fait que les composants logiciels, structurant leurs services en interfaces ou en port, peuvent proposer d'organiser ces services de manières différentes (*i.e.* interfaces différentes ou ports différents), de sorte que les interfaces requises et fournies ne peuvent pas être mises en correspondance.
Dans ce cas, l'adaptation structurelle externe permet de restructurer les services des composants assemblés en redéfinissant leurs interfaces/ports requis et fournis. Cette mise à jour de la structure des interfaces/des ports permet, ainsi, d'offrir une correspondance exacte entre les interfaces des composants (voir Figure 2.2, cas A). Cette adaptation réalisée sans changement de la sémantique des services des composants leur permet d'être correctement assemblés (*i.e.* adaptation pour l'interopérabilité).
Par exemple, dans la figure 2.2, le cas A montre deux composants C_1 et C_2 qui ne peuvent pas être assemblés car les interfaces requises de C_1 ne sont pas compatibles avec celles fournies par C_2, bien que C_2 fournit tous les services requis par C_1. En utilisant notre approche pour adapter la structure externe du composant C_1 et ainsi la rendre compatible avec celle de C_2, l'assemblage entre C_1 et C_2 devient alors possible. L'adaptation structurelle externe permet donc d'éviter l'utilisation d'un nouveau composant (*i.e.* composant d'adaptation) destiné à assurer les connexions entre les deux composants concernés (C_1 et C_2) ;

- *l'adaptation pour la réutilisation partielle de composants logiciels*
  La réutilisation d'un composant logiciel peut être intégrale mais aussi partielle. Le cas d'une réutilisation intégrale est lié au fait que l'ensemble des services proposés par le composant concerné reste utile au niveau de l'application (*i.e.* les besoins de l'utilisateur et de l'application correspondent aux services fournis).
  Le cas de la réutilisation partielle est différent. Il se traduit par le fait que certains services, intégrés au composant lors de sa conception, ne présentent plus d'intérêt ou ne doivent pas être invoqués lors de sa nouvelle utilisation. Ce cas de figure peut se produire, par exemple, lorsque les services qui ne doivent pas être réutilisés ont été remplacés par d'autres fournis par de nouveaux composants, ou bien ils ne font plus partie des besoins de l'utilisateur, ou bien encore, pour des raisons de choix du concepteur de l'application (sécurité, etc.) ces services ne doivent pas pouvoir être invoqués par l'utilisateur. De ce fait, l'adaptation structurelle pourrait permettre d'isoler les parties du composant fournissant des services « indésirables » dans la nouvelle situation, afin de les supprimer ou de les remplacer par de nouveaux services. Cette adaptation va permettre d'améliorer les performances car les ressources matérielles requises par le composant pour être déployer (espace mémoire, etc.) seront réduites. De plus, elle permet de répondre à certains besoins des concepteurs d'applications comme nous avons pu le constater ;

- *l'adaptation pour la correction de composants logiciels*
  L'adaptation structurelle peut également être utilisée pour la correction de composants logiciels (*i.e.* adaptation corrective). En effet, comme nous l'avons évoqué précédemment dans le cadre de la réutilisation, l'adaptation structurelle permet d'isoler certaines parties d'un composant, et notamment des parties défectueuses. Il devient alors plus facile de les corriger ou même de les remplacer en fonction des situations. Une telle approche est fortement encouragée pour la correction de composants monolithiques car leur code source est généralement centralisé dans un même fichier (par exemple, dans le cadre de composant Julia [38], les composants monolithiques sont implémentés par une seule classe Java). De ce fait, l'adaptation structurelle permettrait d'isoler le code source défaillant afin de le traiter plus facilement.

2.5 Démarche et approche de problématique

2.5.1 Les différents problèmes traités dans le cadre de l'adaptation structurelle

Notre objectif dans cette thèse est de traiter différents problèmes liés à l'adaptation structurelle (voir Figure 2.3) de manière à proposer une approche complète permettant d'adapter la structure d'un composant logiciel existant. Les différents problèmes traités sont les suivants :

- *l'adaptation structurelle par la ré-ingénierie de composants existants et son application pour l'introduction de la propriété de distribution*
  Dans un premier temps, notre objectif va être de définir une approche d'adaptation structurelle permettant à partir du code source d'un composant existant d'obtenir un composant dont la structure est conforme à de nouveaux besoins liés à son utilisation. Pour cela, nous proposons un modèle de composants structurellement adaptés définissant les interfaces permettant à un composant adapté de garantir son intégrité et sa cohérence et un processus de ré-ingénierie dont l'objectif est de transformer tout composant existant en un composant conforme à notre modèle de composants structurellement adaptés. L'approche est semi-automatique : elle nécessite une spécification manuelle de la structure du composant adaptée au nouveau contexte. Le processus de transformation est quant à lui entièrement automatique.
  Cette approche d'adaptation structurelle peut être utilisée pour intégrer à un composant des mécanismes de distribution. Pour cela, nous devons proposer un modèle de composants structurellement adaptés distribués et un processus permettant d'obtenir un composant conforme à ce modèle à partir d'un composant issu de notre processus d'adaptation structurelle statique. Ces propositions doivent être présentées comme des extensions du modèle de composants structurellement adaptés et du processus de transformation structurelle ;

- *l'adaptation structurelle dynamique de composants logiciels*
  L'adaptation structurelle par la transformation statique ne permet pas de répondre à tous les besoins relatifs à l'adaptation de la structure. En effet, il doit être possible d'adapter la structure d'un composant logiciel sans arrêter l'application qui le contient ; et ce afin de répondre à des besoins intervenant lors de l'exécution du composant. Ainsi, nous devons proposer une approche permettant de réaliser l'adaptation structurelle de manière dynamique. Cette proposition passe par la définition d'un modèle de composants structurellement et dynamiquement adaptables et d'un processus de ré-ingénierie permettant de rendre un composant existant conforme à ce modèle ;

- *l'adaptation structurelle automatique de composants logiciels*
  Par ailleurs, dans certains environnements tels que dans les environnements ubiquitaires, le contexte d'exécution d'une application change en permanence. De plus, le besoin en adaptation structurelle est très présent comme nous avons pu le constater dans le chapitre précédent. De ce fait, il ne peut être envisageable de fournir une approche d'adaptation semi-automatique. Ainsi, notre objectif va être de proposer une approche réalisant de manière automatique tout un processus d'adaptation ; y compris le déclenchement de l'adaptation et la spécification d'une structure pour le composant, qui soit adaptée à une nouvelle situation. Cette proposition passe par la définition d'un modèle de composants structurellement auto-adaptatifs et d'un processus de ré-ingénierie permettant de rendre un composant existant conforme à ce modèle ;

- *l'adaptation structurelle d'architectures logicielles*
  L'adaptation structurelle d'un composant logiciel peut avoir des impacts sur les autres composants d'une architecture logicielle. De ce fait, nous devons proposer une approche permettant de réaliser l'adaptation structurelle au niveau d'architectures logicielles afin de tenir compte des adaptations de chaque composant qu'elle contient. Pour cela, il est nécessaire de proposer une stratégie de coordination de l'adaptation structurelle au niveau macro (*i.e.* au niveau de l'architecture logicielle).

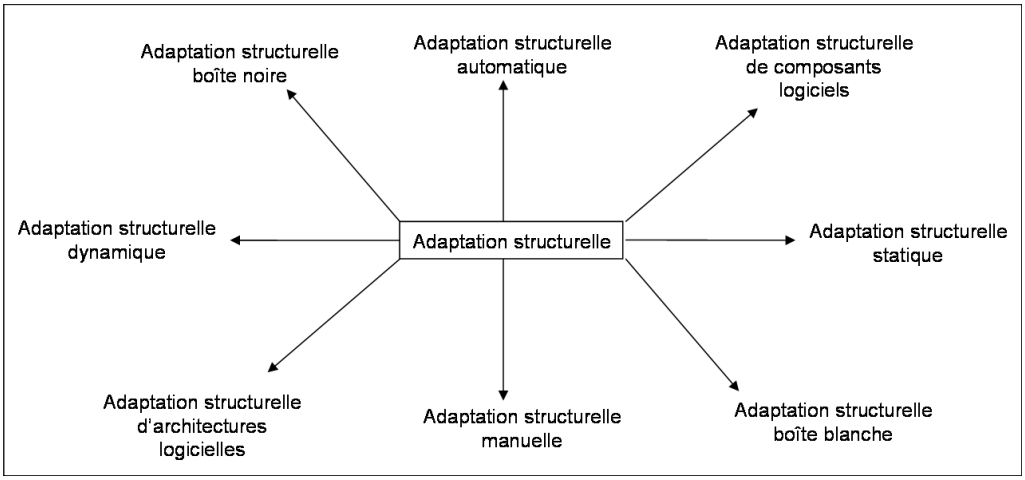

Figure 2.3 – Présentation des différents problèmes à traiter dans le cadre de l'adaptation structurelle

2.5.2 Contraintes et limitations du cadre de l'étude

L'étude de notre approche d'adaptation structurelle nous a imposé certains choix ou hypothèses de travail que nous exposons dans cette section.

Choix réalisés pour la mise en œuvre de notre approche Avant de développer notre approche d'adaptation structurelle, nous avons du faire un certain nombre de choix nous permettant de mieux cerner notre apport. Ces choix sont les suivants :

- *adaptation de la structure interne de composants logiciels*
  Dans nos travaux, nous avons choisi de nous focaliser sur l'adaptation de la structure interne d'un composant logiciel pour répondre à des besoins tels que le déploiement ou le chargement flexible de services. Une telle stratégie peut nécessiter la fragmentation du composant (ou des entités structurelles qui le composent) à adapter et sa reconstitution de manière à garantir le résultat de l'adaptation. Or, comme nous l'avons vu précédemment, un composant est constitué de partie architecturale et d'une partie implémentatoire. De ce fait, l'adaptation structurelle va nécessiter l'étude de problèmes relatifs au partitionnement et à la parallélisation de code tels que la gestion de ressources partagées, etc. ;

- *adaptation au niveau des interfaces de composant*
  Nous considérons, dans le cadre de nos travaux, les interfaces d'un composant comme des briques

de base qui ne peuvent pas être fragmentées. Ainsi, la restructuration d'un composant est réalisée par le regroupement des interfaces fournies par le composant adapté au sein de nouveaux composants considérés comme des sous-composants du composant adapté.
Nous avons choisi d'opérer à ce niveau de granularité car il nous assure une flexibilité suffisante dans la majorité des cas où la restructuration d'un composant est nécessaire. Cependant, il peut être envisagé des extensions à notre approche permettant de manipuler des entités structurelles de plus bas niveaux telles que les points d'entrées d'une interface, les méthodes, etc. comme des briques de base à la restructuration d'un composant.

Hypothèses de travail Les choix que nous avons pris nous imposent de fixer des hypothèses sur le composant à adapter et sur le modèle de composants utilisés afin de mettre en œuvre notre approche d'adaptation structurelle. Ces hypothèses sont les suivantes :

- *indépendance du comportement et de la structure*
  Nous supposons également que l'implémentation du composant que l'on souhaite adapter est indépendante vis-à-vis de sa structure. Ainsi, les services métiers du composant à adapter ne doivent pas faire appel à des données relatives à sa structure. Par exemple, les informations relatives aux nombres d'interfaces requises par d'autres composants de l'application liés aux interfaces fournies par le composant à adapter ne doivent pas être utilisées par les services métiers de ce dernier. Il en va de même en ce qui concerne le nombre de sous-composants contenus dans le composant ou le nombre de composants partagés ;

- *disponibilité du code source des composants pour nos processus de ré-ingénierie*
  La mise en œuvre de nos processus de ré-ingénierie permettant de réaliser l'adaptation structurelle de composants existants ou bien d'introduire des mécanismes d'auto-adaptation dynamique dans des composants existants nécessite la disponibilité du code source du composant à adapter. Cette hypothèse n'est pas en contradiction avec le principe des COTS du fait du fort développement des logiciels « *open-source* ». De plus, différentes stratégies peuvent être envisagées pour permettre la réalisation d'un processus d'adaptation structurelle sans que l'utilisateur puisse accéder au code source de l'application ;

- *implémentation orientée objet, sans objet actif et sans accès à des entités logicielles ou matérielles externes à l'application*
  Dans le cadre de notre étude, nous avons supposé que les composants sur lesquels nous agissons sont implémentés dans un langage orienté objet et qu'ils ne contiennent pas d'objets actifs tels que les *threads*. Cette dernière hypothèse, couramment émise dans les travaux de transformation de code (tels que le partitionnement, etc.), nous permet d'éviter certains problèmes liés à la parallélisation de code qui ne font pas l'objet de cette thèse. Pour plus de détails sur le traitement de *threads* dans le cadre de partitionnement de code, nous invitons le lecteur à consulter [116].
  Par ailleurs, nous supposons que les services du composant à adapter n'utilisent pas des entités logicielles (fichiers, base de données, etc.) ou matérielles externes (périphérique spécifique, etc.) à l'application. En effet, la prise en compte de ces entités ne peut être effective que si ces entités sont dotées de caractéristiques spécifiques. Notamment, elles doivent pouvoir être accédées à distance ; ce qui peut parfois se révéler impossible ou très complexe à mettre en œuvre. De plus, elle peut nécessiter un traitement spécifique qui, dans la majorité des cas, ne peut pas être automatisé. Étant

donné que nous avons pour objectif de proposer une approche entièrement automatique, nous ne pouvons pas prendre en considération l'accès à ces entités.

2.5.3 Notre modèle de composants restructurables

Pour mettre en œuvre notre approche, nous avons défini un modèle générique de référence auxquels les composants que nous proposons d'adapter devront être conformes. Puis, nous avons choisi un modèle de composants existant dans la littérature qui soit conforme à notre modèle générique afin d'illustrer notre approche.

2.5.3.1 Modèle générique de référence

Le modèle de composants à adapter est un modèle hiérarchique (*i.e.* modèle permettant de concevoir des composants de type composite). Cette propriété nous permet notamment de garantir la transparence de l'adaptation. En effet, les composants issus de la fragmentation du composant à adapter pourront être encapsulés dans un composant composite dont la structure externe sera identique à celle du composant initial. Dans le cas où le modèle de composants utilisé n'est pas hiérarchique, notre approche peut être utilisée pour fragmenter un composant existant en plusieurs composants. L'assemblage obtenu pourra remplacer le composant existant. Cependant, dans ce cas, nous ne pouvons plus parler de processus d'adaptation car certaines propriétés du composant ne sont plus préservées (transparence de l'adaptation, encapsulation des sous-composants générés, etc.).

Pour illustrer notre approche, nous considérons des composants logiciels conformes à la spécification générique décrite dans la figure 2.4. Concernant le niveau architectural, les composants peuvent être de type composite ou primitif. Chaque composant définit un ensemble d'interfaces qui peuvent être fournies ou requises. Une interface peut contenir un ou plusieurs services. Pour modéliser le niveau implémentatoire, nous avons utilisé le modèle objet. Comme nous pouvons le remarquer les deux niveaux sont étroitement liés : un composant primitif est implémenté par une hiérarchie de classes, une interface de composant est associé à une interface objet et un service correspond à une méthode de la classe d'implémentation du composant. Ainsi, la restructuration d'un composant doit être réalisée sur les deux niveaux (architectural et implémentatoire). En fait, toute mise à jour de la structure sur l'un des deux niveaux doit impérativement être répercutée sur l'autre niveau.

2.5.3.2 Modèle applicatif

Le modèle de composants, proposé dans la littérature, le plus proche de la spécification présentée dans la figure 2.4 est le modèle de composants Fractal [39] et son implémentation Java appelé Julia [38]. La seule différence réside dans l'association d'un composant primitif à une seule classe d'implémentation. Le niveau architectural d'un composant Fractal est, quant à lui, décrit au travers d'un langage de description d'architecture appelé FractalADL [38] et dont la DTD est fournie dans la figure 5.3. De plus, Fractal présente les caractéristiques idéales pour mettre en œuvre une adaptation dynamique à savoir qu'il intègre dans son modèle les propriétés d'introspection et d'intercession comme nous avons pu le constater précédemment. De ce fait, pour illustrer notre approche, nous avons utilisé ce modèle de composants. Pour plus de détails sur le choix du modèle de composants Fractal et de son implémentation Julia, consulter le chapitre 5.

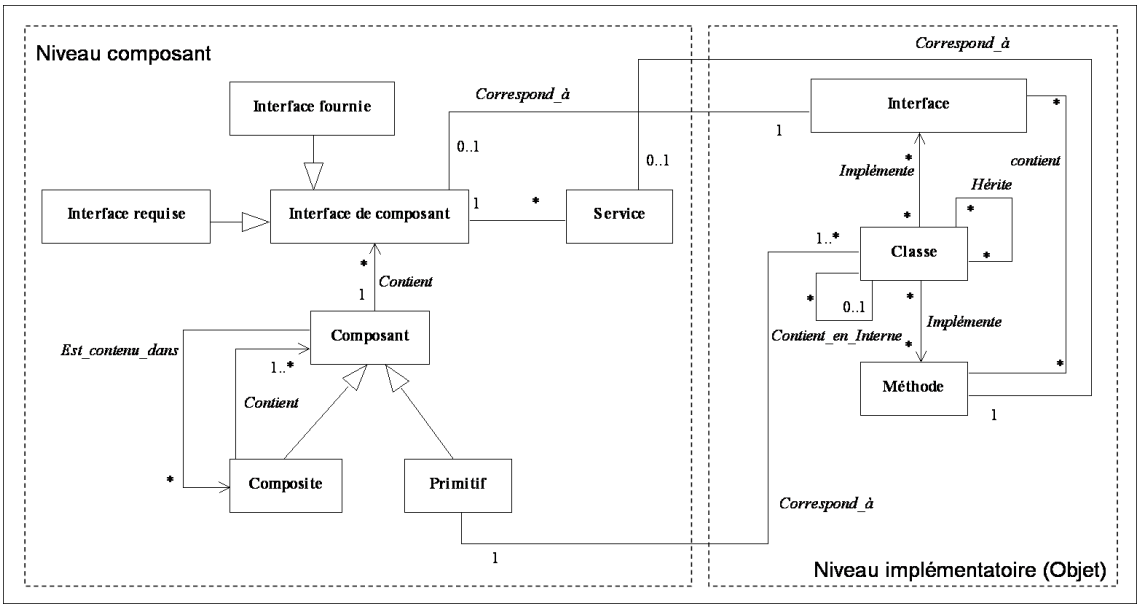

Figure 2.4 – Modèle de composants logiciels de référence

2.6 Cas d'étude : un support de communication dans un projet collaboratif

Dans l'optique d'illustrer notre approche, nous utilisons, tout au long de ce manuscrit, l'exemple d'une application destinée à fournir un support de communication pour un projet collaboratif, appelé *Com-In-Project*. Cette application est déployée sur une infrastructure distribuée dont les nœuds sont dotés de ressources matérielles très variables (*i.e.* environnement à ressources limitées).

2.6.1 Présentation de l'application *Com-In-Project* : rôle des services proposés

L'application *Com-In-Project* a pour rôle de fournir un ensemble de services permettant aux différents acteurs d'un projet de communiquer par l'intermédiaire de supports audio, vidéo ou textuel. Les supports audio/vidéo se traduisent par la possibilité donnée aux acteurs d'organiser des conférences de ce type. Quant au support textuel, il est disponible sous la forme d'un tableau blanc pouvant être utilisé par plusieurs acteurs simultanément et d'un système de messagerie instantanée. L'application *Com-In-Project* fournit également des services permettant aux différents acteurs du projet, d'organiser leurs tâches à accomplir : un système d'agenda partagé leur permet de consulter les disponibilités de chaque acteur, organiser des réunions ou bien planifier des évènements.

2.6.2 Architecture de l'application *Com-In-Project*

L'application *Com-In-Project* est conçue comme un assemblage de composants logiciels existants. Elle offre différents services dont le but est de permettre la communication audio/vidéo et le partage de supports d'affichages et d'outils au sein d'un groupe de travail. Les composants logiciels mis en jeu dans le cadre de cette application sont les suivants :

1. un composant de gestion d'audio-conférences permettant la communication audio entre les machines de déploiement équipées d'entrées/sorties audio ;

2. un composant de gestion de vidéo-conférences permettant la communication vidéo entre les machines de déploiement équipées d'entrées/sorties vidéo ;

3. un composant de service mail permettant d'envoyer des courriels à un individu ou un groupe d'individus ;

4. un composant de gestion de messagerie instantanée permettant aux utilisateurs d'échanger des informations sous forme de textes ;

5. un composant tableau blanc permettant d'écrire des messages (dessins, etc.) destinés à être lus par tous les membres d'un groupe ;

6. un composant de gestion d'agendas partagés permettant de stocker ou de consulter les agendas personnels d'un groupe d'individus et de coordonner les évènements dépendants entre ces agendas.

Ces six composants sont des composants logiciels disponibles sur étagères dont le code source est disponible (COTS « *open-source* »). Ils ont été développés dans le cadre d'autres applications et ont été réutilisés pour créer notre application de travail collaboratif. L'architecture de l'application est fournie dans la figure 2.5 : les composants de gestion de conférences audio et vidéo sont encapsulés au sein d'un même composite dédié à la gestion des conférences (composant *Conférence*) ; les cinq composants ainsi obtenus sont assemblés horizontalement puis encapsulés (*i.e.* assemblage vertical) au sein d'un même composant fournissant l'ensemble des services du support de communication.

2.6.3 Besoin de l'adaptation structurelle dans l'application *Com-In-Project*

Dans certains environnements où les ressources disponibles sont en perpétuelle évolution, tels que les environnements ubiquitaires, il est indispensable d'adapter l'application *Com-In-Project* pour d'une part permettre son déploiement et d'autre part garantir sa continuité de service.

2.6.3.1 Description de la situation nécessitant l'adaptation structurelle

Imaginons que lors du déploiement de cette application sur la machine de l'utilisateur, les ressources nécessaires ne soient pas suffisantes. Par exemple, la taille mémoire disponible sur la machine de déploiement est trop faible. De ce fait, l'application de support de communication doit être adaptée pour être déployée. Nous avons exclu la possibilité de déployer les composants dans leur intégralité sur des machines distantes par mesure de sécurité mais également afin que l'utilisateur de l'application puisse garder une certaine autonomie d'utilisation. En effet, l'utilisateur peut, dans certains cas, imposer que certains services ou données ne soient déployés que sur sa propre machine ou sur des serveurs sécurisés. De plus, du fait de la volatilité des connexions de la machine de l'utilisateur à l'infrastructure disponible, certains services peuvent devenir indisponibles pendant de longues durées. De ce fait, il parait

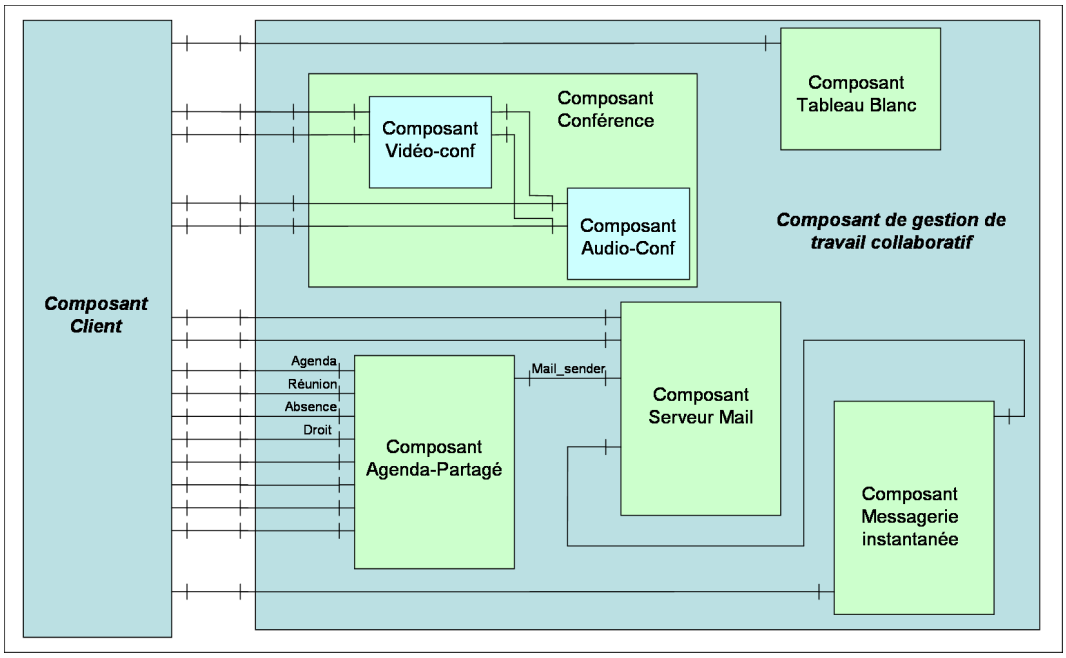

Figure 2.5 – Architecture de l'application : *Com-In-Project*

indispensable de déployer un maximum de services (fonction des ressources disponibles) sur la machine de l'utilisateur afin qu'il puisse conserver une certaine autonomie vis-à-vis des risques de déconnexion pouvant être non négligeables dans certains cas.

2.6.3.2 Besoins d'adaptation structurelle pour la distribution

Étant donné que nous disposons d'une infrastructure distribuée (c'est-à-dire, d'autres machines sont disponibles à travers le réseau), une solution consisterait à déployer certains services sur d'autres nœuds de l'infrastructure (*i.e.* répartition des tâches sur plusieurs sites) en fonction du contexte ; ce dernier faisant référence en premier lieu aux caractéristiques techniques de l'infrastructure de déploiement (*i.e.* adéquation à l'architecture matérielle) mais aussi aux propriétés des utilisateurs et aux conditions d'utilisations (par exemple, les services susceptibles d'être les plus utilisés doivent être déployés en priorité sur la machine de l'utilisateur).

Cependant, les composants logiciels constituant l'application de support de communication ne permettent pas de distribuer les services qu'ils proposent tel que le souhaite l'administrateur. En effet, la majorité des composants a été conçue comme des entités monolithiques. De ce fait, l'ensemble des services qu'ils proposent ne peuvent pas subir de déploiement distribué (*i.e.* un composant monolithique doit être déployé sur un seul nœud de l'infrastructure). De ce fait, deux solutions peuvent être envisagées pour obtenir la configuration souhaitée par l'administrateur : soit, développer à nouveau les composants incapables de distribuer certains des services qu'ils fournissent (tels que les composants monolithiques), en fonction des nouveaux besoins. Cette solution ne peut pas être envisagée pour des raisons de coût ; soit adapter la structure de ses composants de manière à la rendre conforme à la configuration souhaitée.

Par exemple, considérons le composant *Agenda-partagé*. Ce composant a été implémenté comme un composant monolithique fournissant les services suivants :

- gérer un agenda personnel (authentification, consultation d'évènements, recherche de contacts ou d'évènements, etc.).
  Ce service est offert au travers de l'interface *Agenda* ;

- organiser des réunions (confirmer la possibilité d'organiser une réunion dont la date et la liste des personnes concernées sont données comme paramètres, proposer des dates possibles pour la programmation de réunions entre un ensemble d'individus, etc.).
  Ce service est offert au travers de l'interface *Réunion* ;

- gérer les jours d'absences (consulter les dates d'absences pour un individu dont le nom est donné en paramètre, etc.).
  Ce service est offert au travers de l'interface *Absence* ;

- gérer les droits (modifier les droits de consultation, modification et suppression d'un agenda, paramétrer un agenda en attribuant des droits d'absences à un individu, etc.).
  Ce service est offert au travers de l'interface *Droit* ;

- mettre à jour un agenda personnel, les dates de réunions, les dates d'absences et les droits d'absences d'une personne.
  Ces services sont offerts, respectivement, au travers des interfaces *MiseAjourAgenda*, *MiseAjourRéunion*, *MiseAjourAbsence* et *MiseAjourDroit*.

Ce composant requiert également un service permettant d'envoyer des courriels aux personnes concernées par l'organisation d'une réunion. Ce service est fourni par l'interface *Mail_sender* du composant *Serveur Mail*.

Le composant *Agenda-partagé* a été implémenté en utilisant la plate-forme Julia [38] qui est l'implémentation Java du modèle de composants Fractal [39] (voir Chapitre 5). Ainsi, le composant *Agenda-partagé* est défini sur deux niveaux :

1. le niveau architectural qui contient la description de la structure du composant réalisée en utilisant le langage FractalADL (voir Section 5.2) :

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/standard.dtd">
4
5  <component name="AgendaPartagé">
6  <interface name="It_reunion" role="server" signature="Reunion"/>
7  <interface name="It_miseAjourRéunion" role="server" signature="MiseAjourRéunion"/>
8  <interface name="It_absence" role="server" signature="Absence"/>
9  <interface name="It_miseAjourAbsence" role="server" signature="MiseAjourAbsence"/>
10 <interface name="It_droit" role="server" signature="Droit"/>
11 <interface name="It_miseAjourDroit" role="server" signature="MiseAjourDroit"/>
12 <interface name="It_agenda" role="server" signature="Agenda"/>
13 <interface name="It_Mail_sender" role="client" signature="Mail_sender"/>
14 <content class="AgendaPartagéImpl"/>
15 </component>
```

2. le niveau implémentatoire qui est constitué uniquement de la classe d'implémentation du composant *Agenda-partagé* (*AgendaPartagéImpl.java*) car ce dernier est de type primitif :

```
1  public interface Reunion {
2  void creer_reunion(Date jour,Vector personne);
3          void consulter_reunion(String n);
4          boolean confirmer_reunion(Date jour,Vector personne);
5          boolean est_en_reunion(String n,Date jour);
6    ... }
7
8  public interface Absence {
9          void ajouter_absence(String n,Date d);
10         void consulter_absence(String n);
11         boolean est_absent(String n, Date d);
12         int consulter_nb_absence(String n);
13         ... }
14 ...
15
16 public class AgendaPartagéImpl implements Reunion, Absence, ... , AgendaPartagéAttributes
17 {
18 ...
19 }
```

Comme nous pouvons le constater, le composant *Agenda-partagé* est doté d'une structure monolithique et celle-ci ne correspond pas aux besoins liés à son utilisation dans le cadre de l'application *Com-In-Project*. En effet, certains services fournis par ce composant doivent être déployés sur d'autres nœuds de l'infrastructure : tout d'abord, la machine de l'utilisateur ne dispose pas des ressources nécessaires pour déployer le composant *Agenda-partagé* en intégralité. De plus, il ne peut être envisagé que le composant soit déployé dans son intégralité sur une machine distante car certains services comme la gestion des agendas personnels doivent rester en permanence accessibles à l'utilisateur malgré les déconnexions éventuelles ; contrairement aux services de gestion de réunions qui nécessitent une connexion pour fonctionner, donc ces services peuvent être distribués dans l'infrastructure disponible. Donc, le composant *Agenda-partagé* doit être déployé de manière distribuée. Cette opération passe obligatoirement par l'adaptation de sa structure. Ainsi, grâce à l'adaptation structurelle par la ré-ingénierie de composants existants (voir Chapitre 3), de nouveaux sous-composants fournissant des sous-ensembles de services fournis par le composant initial (issus de la fragmentation du composant initial) doivent être créés. Chaque sous-composant pourra alors être distribué sur les différents nœuds de l'infrastructure concernés. Par exemple, les services qui concernent respectivement la gestion des agendas personnels, la gestion des réunions et la gestion des absences peuvent être spécifiés de manière à être installés sur différents sites. Ainsi, nous définissons trois nouveaux sous-composants pour le composant *Agenda-partagé* :

1. *GestionnaireDAgenda*, fournissant les interfaces *Agenda* et *MiseAjourAgenda* ;

2. *GestionnaireDeReunion*, fournissant les interfaces *Réunion* et *MiseAjourReunion*. Ce composant devra être redéployé sur un nouveau site dont l'adresse IP est la suivante : 10.1.10.157 ;

3. *GestionnaireDAbsence*, fournissant les interfaces *Absence*, *MiseAjourAbsence*, *Droit* et *MiseAjourDroit*. Ce composant devra être redéployé sur un nouveau site dont l'adresse IP est la suivante : 10.1.10.160.

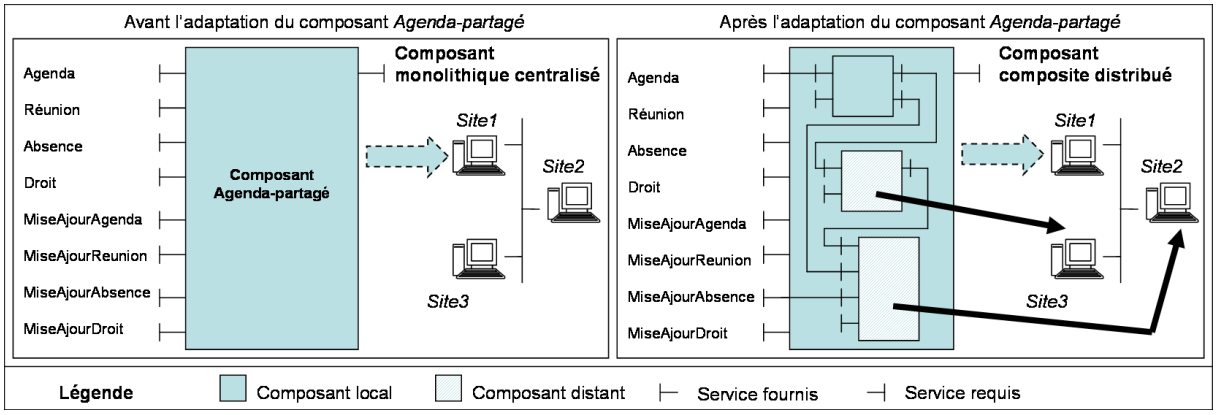

Figure 2.6 – Adaptation structurelle du composant *Agenda-partagé*

Ainsi, grâce à l'adaptation structurelle par la ré-ingénierie, le composant *Agenda-partagé* peut être déployé conformément aux besoins de l'administrateur de l'application *Com-In-Project*.

2.6.3.3 Besoin d'adaptation structurelle dynamique et automatique

Cependant, dans ce type d'environnement où les ressources sont limitées, le contexte d'exécution d'une application change en permanence. De ce fait, la continuité de service de l'application *Com-In-Project* peut être remise en question à tout moment car les ressources disponibles sur un site de déploiement peuvent devenir insuffisantes pour les services qui y sont déployés. Aussi, les composants de l'application doivent être capables d'adapter dynamiquement et automatiquement leur structure. Pour cela, nous devons, avant de déployer l'application, intégrer à ses composants des mécanismes leur permettant de réaliser cette adaptation (voir Chapitre 4). Ainsi, la continuité de service de l'application *Com-In-Project* pourra être garantie tout au long de son exécution.

2.7 Conclusion

Dans ce chapitre, nous avons présenté, de manière générale, notre approche permettant d'adapter la structure de tout type de composants logiciels existants. Cette approche, appelée adaptation structurelle (SCORPIO), est basée sur la considération d'une nouvelle facette de l'adaptation à savoir la structure du composant.

Comme nous avons pu le constater, ce type d'adaptation peut répondre à de multiples besoins tels que le déploiement flexible de composants logiciels pour réaliser l'adéquation aux ressources disponibles de l'architecture matérielle ou bien l'assemblage flexible de composants existants.

Dans les chapitres suivants, nous allons détailler la mise en œuvre de notre approche d'adaptation structurelle, tout d'abord, de manière statique, de par la proposition d'un processus de ré-ingénierie permettant d'adapter la structure de composants logiciels existants (voir Chapitre 3) puis de manière dynamique (*i.e.* sans arrêter l'application qui contient le composant à adapter) et automatique (*i.e.* prise de décisions réalisée automatiquement en fonction du contexte courant) dans le contexte des environnements ubiquitaires (voir Chapitre 4). Enfin, dans le chapitre 5, nous présenterons nos expérimentations à

savoir notre prototype appelé Scorpio-Tool ainsi qu'un scénario de mise en œuvre de l'adaptation structurelle.

CHAPITRE 3

Adaptation structurelle par la ré-ingénierie de composants existants

3.1 Introduction

Comme nous avons pu le constater dans le chapitre 1, la réutilisation de composants logiciels existants tels que les COTS [67] pour la conception d'applications est à la base de l'ingénierie des composants. Cependant, nous avons vu que dans la grande majorité des cas, il est indispensable de procéder à une adaptation des composants existants avant de pouvoir les réutiliser car les points de variabilité dont ils sont dotés, ne sont pas toujours suffisants pour répondre aux contraintes imposées dans le cadre d'une nouvelle utilisation des composants en question. Or, pour adapter un composant existant dans le cas où celui-ci n'a pas été conçu pour répondre aux nouvelles attentes liées à sa réutilisation, il est nécessaire de proposer un processus de ré-ingénierie qui va permettre de rendre le composant apte à être exécuté dans de nouvelles conditions de par la mise à jour de son implémentation.

Par ailleurs, nous avons également pu constater dans le chapitre 1 qu'aucune approche proposée dans la littérature ne permettait d'adapter la structure de composants logiciels existants. De ce fait, nous avons pour objectif de proposer dans ce chapitre un processus de ré-ingénierie qui va permettre à partir d'un composant existant d'obtenir, par un ensemble de transformations, un composant dont la structure répond aux attentes liées à sa réutilisation dans un nouveau contexte.

Parmi les principales motivations de l'adaptation de la structure d'un composant logiciel, nous avons pu noter la possibilité de réaliser sa distribution. Or, les composants issus de décomposition du composant initial sont centralisés. De ce fait, nous proposons également dans ce chapitre d'intégrer au composant résultat de l'adaptation structurelle, des mécanismes de distribution.

Cette approche de ré-ingénierie va donc permettre d'augmenter la réutilisabilité des composants logiciels existants. En effet, grâce au processus que nous proposons de fournir, la structure d'un composant pourra être adaptée en fonction des nouveaux besoins de son utilisation. Pour mettre en œuvre ce processus, nous avons défini un modèle de composants logiciels auxquels tout composant structurellement adapté en utilisant notre approche doit être conforme ; ceci afin de garantir le comportement du résultat de notre processus de transformation. Nous proposons également, dans ce chapitre, une étude des performances de notre modèle de composants support de l'adaptation structurelle.

3.2 Processus de transformation structurelle de composants existants

Dans cette section, nous présentons notre processus de transformation permettant d'obtenir à partir d'un composant existant, un composant dont la structure correspond aux nouveaux besoins de son utilisation.

3.2.1 Présentation du processus de transformation structurelle de composants existants

3.2.1.1 Objectif du processus de transformation structurelle de composants existants

Notre processus de transformation structurelle de composants existants a pour objectif de réaliser l'adaptation de la structure interne d'un composant existant telle que nous l'avons décrite dans le chapitre précédent. Nous nous focalisons dans ce chapitre sur l'adaptation d'un composant par sa décomposition en sous-composants. Ainsi, l'adaptation structurelle permet d'organiser les services fournis par le composant en divers sous-ensembles ; chacun étant un fragment du composant initial. De cette manière, les différents fragments (sous-composants) obtenus peuvent être traités individuellement. Par exemple, les sous-composants pourraient être distribués sur l'infrastructure de déploiement ou bien mis à jour de manière individuelle.

Cette possibilité peut être utilisée, entre autre, pour définir une stratégie flexible de déploiement de composants. Par exemple, la figure 3.1 montre un composant monolithique C_1 qui ne peut être déployé sur la machine de déploiement (*Site1*) du fait des ressources limitées qu'il dispose (*i.e.* capacité mémoire faible, etc.). De ce fait, l'administrateur de l'application peut répartir le composant C_1 sur l'infrastructure disponible afin de le déployer. Pour obtenir ce résultat, il peut utiliser notre processus de ré-ingénierie afin de transformer le composant qui ne correspond pas à ses attentes en un composant pouvant être réparti. Ainsi, la structure interne du composant à adapter va être modifiée de par la création de nouveaux sous-composants (C_2, C_3, C_4, C_5) issus de la fragmentation du composant C_1. Les sous-composants ainsi obtenus pourront alors être déployés, en fonction du contexte, sur différents nœuds de l'infrastructure distribuée disponible : C_2 sera déployé sur le site 1, C_4 et C_5 sur le site 2 et enfin C_3 sur le site 3. Quant au composant C_1, il reste disponible sur le site 1 bien que ses sous-composants soient répartis, de manière transparente.

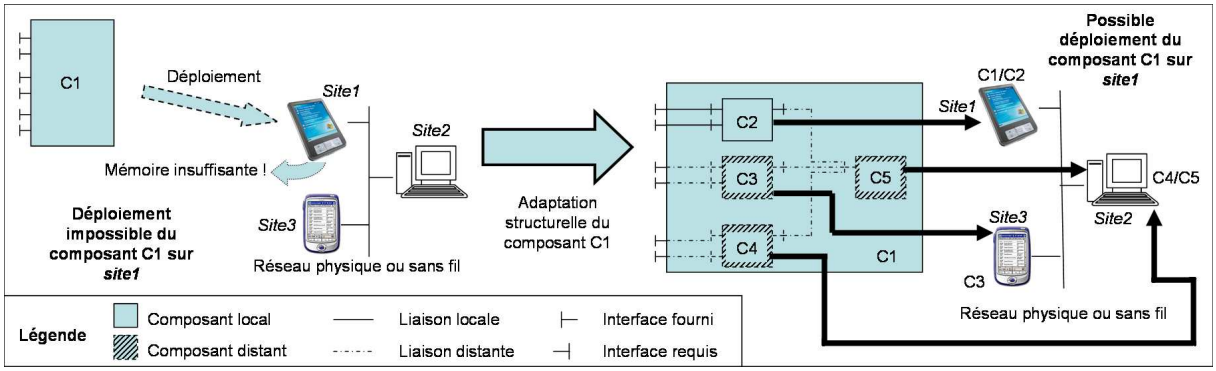

Figure 3.1 – Exemple d'adaptation structurelle pour le déploiement flexible

3.2.1.2 Contraintes du processus de transformation structurelle

Pour être valide, notre processus de transformation doit préserver un ensemble de propriétés que nous avons extraites à partir de l'étude des invariants de l'adaptation dans le chapitre état de l'art. La satisfaction de ces propriétés permet de garantir que le comportement du composant après sa transformation reste identique à celui du composant avant sa transformation. Ces propriétés sont les suivantes :

- *l'intégrité des sous-composants générés par la transformation*
  Tout d'abord, afin de garantir le comportement du résultat de l'adaptation, il est nécessaire d'assurer l'intégrité des composants générés par la fragmentation du composant initial. En fait, la validité de leur implémentation doit être vérifiée : elle doit être correcte syntaxiquement et sémantiquement, complète et cohérente. En effet, le code doit être conforme aux règles élémentaires du langage (grammaire et syntaxe), chaque partie du code doit pouvoir accéder aux éléments (fonctions, attributs, etc.) dont elle a besoin pour fonctionner correctement et enfin le comportement correspondant à un composant généré doit être conforme à un comportement partiel du composant initial ;

- *la cohérence du résultat de la transformation*
  Le comportement de l'assemblage de composants issus de la fragmentation du composant initial (*i.e.* nouvelle structure interne du composant adapté) doit être identique au comportement du composant initial. De ce fait, la cohérence de l'assemblage obtenue doit être préservée.
  Le maintien de cette propriété impose de vérifier que les comportements locaux des composants générés soient cohérents entre eux. Par exemple, si plusieurs composants partagent une même ressource, l'état de cette ressource doit être cohérente ; ce qui impose l'insertion de mécanismes de synchronisation et de notification de ces ressources au sein même des composants concernés ;

- *l'interopérabilité du résultat de la transformation*
  Afin de préserver son interopérabilité, la structure externe du composant après sa transformation (services fournis, services requis) doit être identique à celle initiale. Les deux composants : celui avant sa transformation et celui après sa transformation, doivent donc être substituables. En fait, si l'on considère le composant adapté comme une boîte noire, les deux composants doivent être totalement identiques ;

- *la transparence du résultat de la transformation*
  La transformation d'un composant ne doit pas engendrer la création de nouveaux services fournis. Contrairement à la propriété d'interopérabilité qui ne s'applique que sur la structure externe du composant, la transparence doit être vérifiée également pour la structure interne. Ainsi, si de nouveaux services sont créés au cours de la transformation, ces derniers doivent être rendus inaccessibles pour le reste des composants de l'application. Ces services font essentiellement référence à ceux susceptibles d'être introduits au niveau des sous-composants pour assurer la cohérence de leur assemblage. Par exemple, il doit être strictement interdit à d'autres composants (autres que ceux générés au cours de la transformation) d'accéder aux services permettant de modifier l'état d'une ressource partagée ;

- *l'autonomie des composants générés par la transformation*
  Les nouveaux composants issus de la fragmentation du composant initial doivent être manipu-

lables, dès leur création, comme des entités indépendantes. Par exemple, il doit être possible de spécifier une configuration de déploiement en désignant directement les composants générés.

3.2.1.3 Stratégie de réalisation du processus de transformation structurelle

Le processus de transformation structurelle que nous avons défini requiert le code source du composant à adapter. En fait, l'analyse, la transformation et l'instrumentation du code source du composant sont utilisées pour produire le code des nouveaux composants générés par la décomposition du composant initial. Cette condition est en conflit avec une des propriétés généralement admise dans le cadre de composants COTS qui est la non disponibilité du code source du composant au moment de son utilisation. De ce fait, nous proposons deux stratégies pour implémenter notre processus de transformation d'un composant. La première respecte cette propriété alors que la seconde ne la respecte pas.

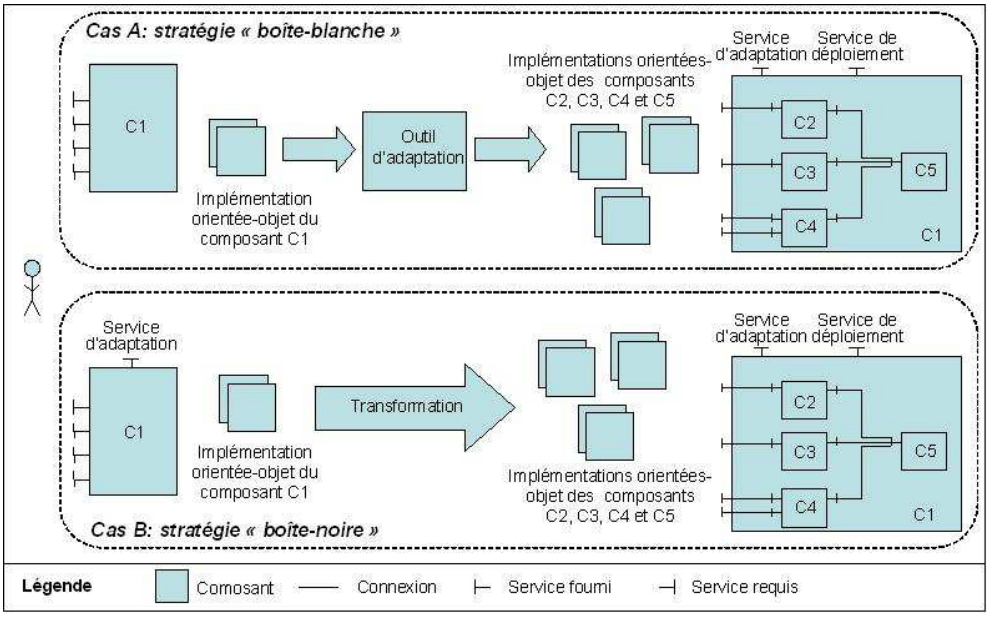

Figure 3.2 – Choix de réalisation de l'approche d'adaptation structurelle

- *Mise en œuvre du processus au travers d'un service non fonctionnel*
  La première stratégie consiste à concevoir notre processus de transformation structurelle comme un service non fonctionnel supplémentaire, appelé service d'adaptation structurelle, fourni par le composant (voir Figure 3.2, cas B). Le code source du composant peut être encapsulé dans le composant sous un format crypté dont seul le service d'adaptation structurelle en connaît la clef. Le code binaire du composant adapté pourra alors être généré automatiquement sans pour autant que les instigateurs de l'adaptation puissent consulter le code source du composant. Cependant, ce service doit impérativement être intégré au composant par son éditeur qui devra fournir la clef pour décrypter le code source du composant.

- *Mise en œuvre du processus par une entité logicielle externe*
  La deuxième stratégie consiste à concevoir une entité logicielle externe à l'application (voir Figure 3.2, cas A) qui requiert le code source du composant à adapter et génère automatiquement, en fonction d'une spécification fournie, le code source correspondant au composant résultat de l'adaptation. Dans ce cas, le code source du composant doit être accessible pour l'adapter. Cette stratégie est favorable pour les composants dits « open sources ».

3.2.1.4 Réalisation du processus de transformation structurelle

Tout d'abord, le composant initial est transformé afin que sa structure interne devienne conforme à celle souhaitée par un acteur externe de l'adaptation. Le composant obtenu est alors dit « structurellement adapté ». Un tel composant est en fait un composant composite doté du même comportement (*i.e.* même services fournis) que le composant initial mais dont la structure interne a été modifiée. En fait, de nouveaux sous-composants issus de la fragmentation du composant initial ont été générés ; chacun d'eux fournissant un sous-ensemble des services fournis par le composant initial.

La transformation structurelle est réalisée par l'intermédiaire d'un processus comportant deux étapes (voir Figure 3.3) :

1. *la décomposition du composant à adapter*
   Avant toute opération de transformation, la nouvelle structure du composant correspondant aux nouveaux besoins doit être spécifiée (*i.e.* spécification des sous-composants à générer réalisée par la désignation de leurs interfaces fournies et de leur site de déploiement). Cette opération est réalisée manuellement par un acteur externe de l'adaptation. Ensuite, pour obtenir une structure interne correspondant à la spécification donnée, il est nécessaire de décomposer le composant initial en sous-composants. Cette opération doit être réalisée par la fragmentation automatique du composant initial. Ainsi, la description de l'architecture de l'application contenant le composant adapté doit être mise à jour et l'implémentation du composant doit être partitionnée tout en garantissant l'intégrité structurelle et la cohérence du code généré ;

2. *la recomposition du composant à adapter*
   La décomposition du composant permet de générer les nouveaux sous-composants spécifiés. Cependant, ces composants ne sont généralement pas indépendants les un des autres. De ce fait, ils doivent être assemblés pour garantir leur comportement. L'assemblage est réalisé en deux étapes : la première étape consiste à matérialiser les dépendances entre les composants générés. Cette opération passe par l'introduction de nouvelles interfaces pour chaque composant généré. La mise en œuvre de ces interfaces nécessite la génération de code au niveau de chaque classe d'implémentation correspondant aux composants générés. La deuxième étape réside dans l'encapsulation de l'assemblage précédemment obtenu dans un composant composite. L'encapsulation permet de masquer les services qui ont été créés lors de l'assemblage horizontal du composant. De ce fait, le composant composite obtenu présente la même structure externe (*i.e.* même services fournis et requis) que celle du composant adapté : les deux composants sont alors substituables.

Il est à noter que toute modification réalisée sur la structure du composant à adapter doit être répercutée sur son code source (*i.e.* niveau implémentatoire) et réciproquement (voir Figure 3.4).

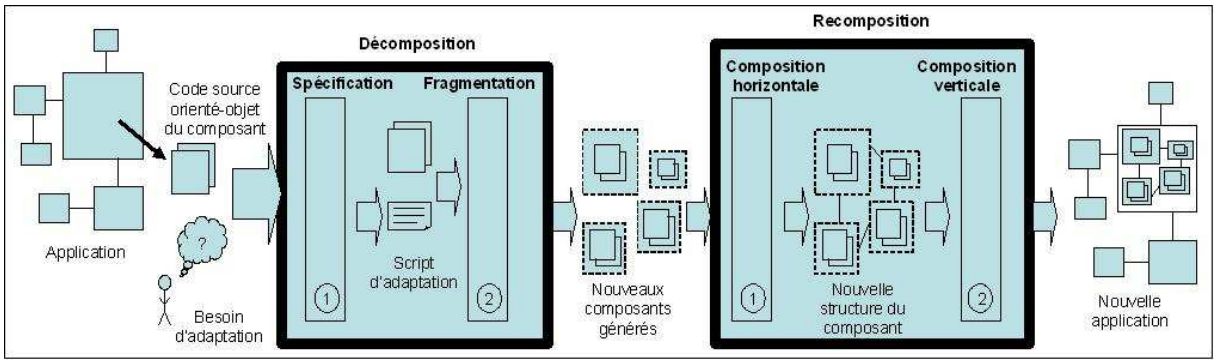

Figure 3.3 – Processus d'adaptation structurelle par ré-ingénierie de composants existants

Une troisième étape peut être ajoutée à notre processus de transformation structurelle. Cette étape, optionnelle, permet d'offrir la possibilité à un acteur de l'adaptation de configurer certaines propriétés liées à la relation de composition ou bien d'intégrer au composite de nouvelles propriétés :

- *la configuration de propriétés liées à la relation de composition*
  Afin d'intégrer au composant composite issu de l'adaptation structurelle de nouveaux points de variabilité, en termes de flexibilité et en vue de fournir des facilités pour une éventuelle future adaptation fonctionnelle (*i.e.* adaptation comportementale), nous proposons un modèle de composants composites donnant la possibilité à un acteur externe de l'adaptation de configurer certaines propriétés du composite, liées à la relation de composition (encapsulation, cycle de vie, etc.) ;

- *l'intégration de mécanismes de distribution au composant composite*
  Par ailleurs, nous avons pu constater dans les applications de l'adaptation structurelle que, souvent, la réutilisation de composant logiciel pouvait requérir l'introduction de mécanismes de distribution au sein du composant adapté (application pour le déploiement flexible, etc.). Or, le composant issu de notre processus de transformation structurelle est centralisé. De ce fait, nous proposons d'étendre notre processus de transformation structurelle afin de lui permettre d'introduire, si besoin est, des mécanismes de distributions dans le composant obtenu. Ces mécanismes vont être chargés d'assurer la communication entre les sous-composants générés lors de la décomposition du composant initial, qui seront déployés sur des machines différentes. L'introduction de ces mécanismes passe par la définition d'un modèle de composants composites distribués (voir Section 3.4) et l'intégration de nouvelles étapes à notre processus permettant d'obtenir un composant conforme à ce modèle (voir Section 3.2.4.2).

Dans ce chapitre, nous nous focalisons plus particulièrement sur les composants monolithiques car leur transformation nécessite la mise en œuvre de mécanismes de fragmentation de leur code source en plus de la mise à jour de la description de l'architecture de l'application qui le contient (qui peut être suffisante lors de la transformation de composants sous forme composite). Cependant, ce processus peut être facilement étendu à des composants logiciels déjà sous forme composite. Dans ce cas, l'étape de décomposition est appliquée à tous les composants monolithiques encapsulés par le composite à adapter. Ensuite, l'étape de recomposition est identique à celle présentée ci-dessous.

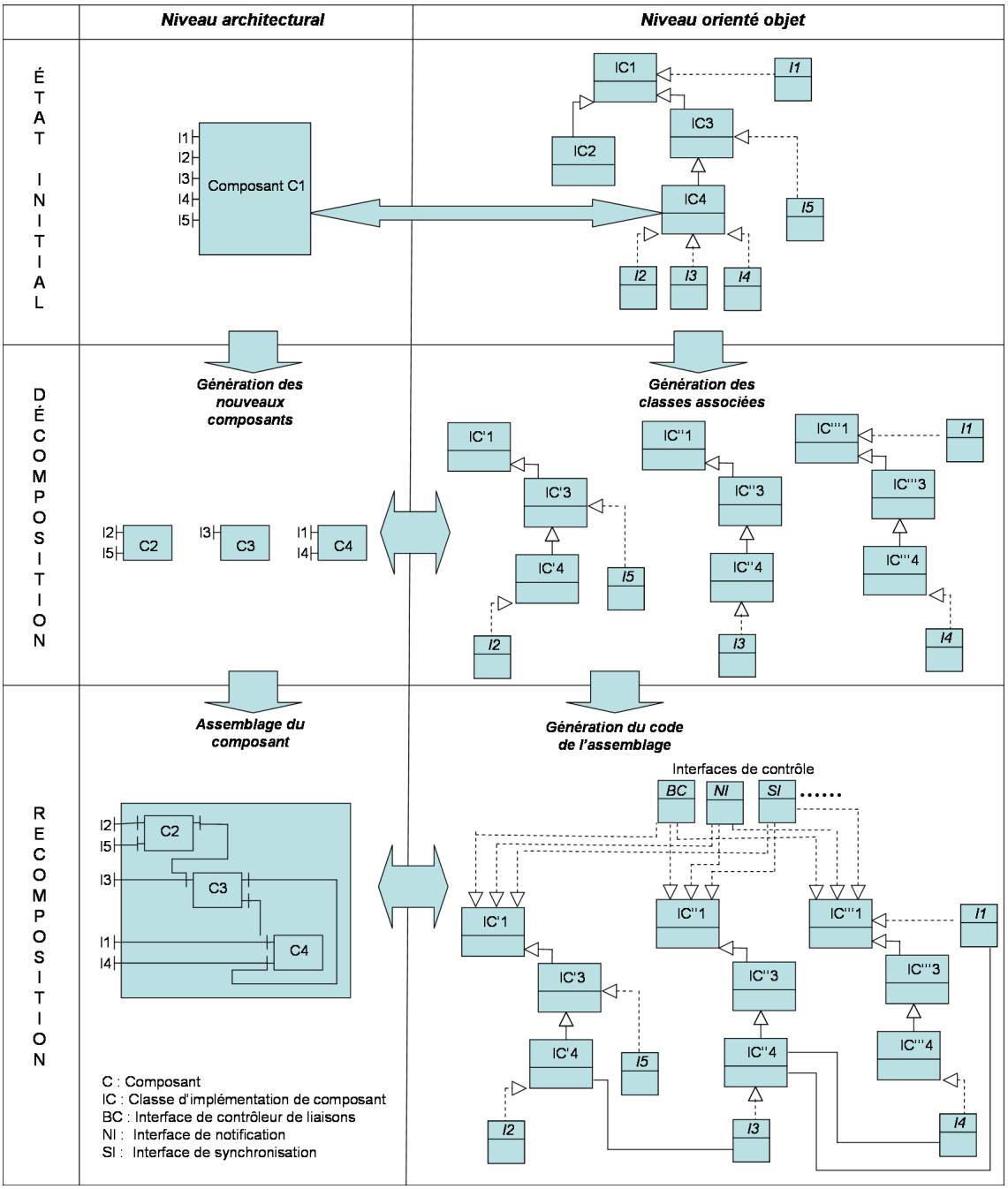

Figure 3.4 – Les deux niveaux de transformation d'un composant par adaptation structurelle

3.2.2 Transformation du composant monolithique vers un composant fragmenté

La première étape de notre processus d'adaptation structurelle réside dans la décomposition du composant à adapter (voir Figure 3.5). Elle comporte deux sous-étapes qui sont :

- *la spécification du résultat de l'adaptation*
  Tout d'abord, un acteur externe de l'adaptation doit spécifier les résultats attendus en précisant la nouvelle structure interne du composant qu'il souhaite obtenir. Cette opération doit être réalisée tout en exerçant des contrôles sur cette spécification de manière à en assurer sa validité ;

- *la fragmentation du composant initial*
  Ensuite, l'implémentation du composant initial est automatiquement fragmentée en fonction de la spécification donnée. En fait, cette description permet de fragmenter la structure du composant initial en différentes entités qui vont servir de base à la construction du nouveau composant, résultat de l'adaptation. La fragmentation doit être réalisée au niveau de la description de l'architecture de l'application contenant le composant ainsi qu'au niveau de l'implémentation orienté objet du composant. La réalisation de ces deux tâches nécessite le contrôle de l'intégrité de chacun de ces nouveaux composants ainsi que le maintien de leur cohérence.

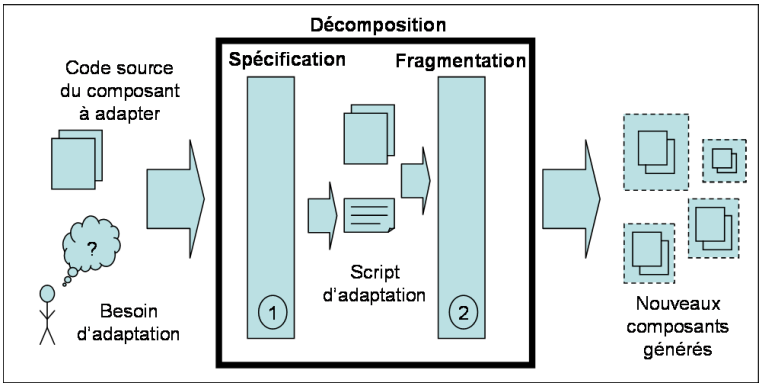

Figure 3.5 – Phase de décomposition du composant logiciel

3.2.2.1 Spécification des besoins de l'adaptation

Cette première étape du processus de transformation structurelle est réalisée en utilisant un ADL (*Architecture Description Language*) de type XML et dont la DTD est fournie dans la figure 3.6. La spécification produite permet de définir les composants à générer et pour chaque composant, ses interfaces et son site de déploiement. Un contrôle destiné à vérifier la validité de cette spécification doit être appliqué : les nouveaux composants doivent définir un ensemble d'interfaces comme étant chacun un sous-ensemble des interfaces du composant initial. De plus, l'union de ces sous-ensembles doit être égale à l'ensemble des interfaces fournies par le composant initial. Également, aucun élément spécifié dans cette description d'architecture ne doit être en contradiction avec celle du composant initial. Par exemple, les signatures des interfaces doivent être identiques.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- An XML DTD for Scorpio : adaptationScript.dtd -->
<!-- Component adaptation script -->

<!ELEMENT a-component (g-component*)>
<!ELEMENT g-component (service+)>
<!ELEMENT service>

<!ATTLIST a-component
 name CDATA #REQUIRED
 class CDATA #REQUIRED
>
<!ATTLIST g-component
 name CDATA #REQUIRED
 host CDATA #IMPLIED
>
<!ATTLIST service
 signature CDATA #REQUIRED
>
```

Figure 3.6 – Spécification du résultat attendu du processus de transformation structurelle

Exemple de l'agenda-partagé
Pour illustrer cette étape, considérons l'exemple du composant *Agenda-partagé*. L'objectif de son adaptation est de réorganiser sa structure en créant trois nouveaux sous-composants : *GestionnaireDAgenda*, *GestionnaireDeReunion* et *GestionnaireDAbsence*. Ainsi, la spécification de l'adaptation permettant d'obtenir cette nouvelle structure du composant *Agenda-partagé* est donnée dans la figure 3.7.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://org/objectweb/fractal/adl/xml/adaptationScript.dtd">

 <a-component name="Agenda-partage">
        <g-component name="GestionnaireDAgenda">
                <service signature="Agenda" />
                <service signature="MiseAjourAgenda" />
        </g-component>
        <g-component name="GestionnaireDeReunion" host="10.1.10.157">
                <service signature="Reunion" />
                <service signature="MiseAjourReunion" />
        </g-component>
        <g-component name="GestionnaireDAbsence" host="10.1.10.160">
                <service signature="Absence" />
                <service signature="MiseAjourAbsence" />
                <service signature="Droit" />
                <service signature="MiseAjourDroit" />
        </g-component>
 </a-component>
```

Figure 3.7 – Spécification d'adaptation du composant *Agenda-partagé*

3.2.2.2 Construction du graphe structurel et comportemental du composant

La fragmentation est réalisée en analysant le code source du composant monolithique, puis en déterminant pour chaque nouveau composant le code d'implémentation lui correspondant tout en tenant

compte des dépendances pouvant exister entre les différents éléments ainsi reconstruits. Cette étape est principalement basée sur la construction de graphes, appelés SBDG (*Structural and Behavioral Dependency Graph*) ; un SBDG est ainsi créé pour chaque nouveau composant à générer.

Définition Un SBDG est un graphe où les nœuds sont des entités structurelles et les arcs représentent les dépendances structurelles et comportementales existantes entre ces entités. Un SBDG représente donc une image de la structure et du comportement d'un composant.

Les relations de dépendances représentées dans un SBDG peuvent être structurelles ou comportementales. Les dépendances structurelles font référence aux relations structurelles définies dans le chapitre 2. Il s'agit par exemple de la composition (par exemple, un port est composé d'interfaces, une interface est composée de méthodes, etc.) ou de l'héritage (par exemple, un composant peut hériter d'un autre composant, une interface peut hériter d'une autre interface, etc.). Ainsi, un composant est structurellement dépendant de ses ports, un port est structurellement dépendant de ses interfaces, une interface est structurellement dépendante de ses services, etc. Les dépendances comportementales représentent des appels de méthodes ou de services d'un composant. Les dépendances structurelles et comportementales sont déterminées en analysant le code source du composant.

Exemple de l'agenda-partagé
Le code source du composant *Agenda-partagé* présenté ci-dessous, nous permet d'extraire, par analyse, différentes informations : la méthode *confirmer_reunion* (lignes 3-10) de l'interface *Réunion* fait appel à la méthode *est_en_reunion* de la même interface ainsi qu'à la méthode *est_absent* de l'interface *Absence* (ligne 6). De ce fait, ces dépendances comportementales devront être créées entre ces méthodes.

```
1  ...
2  public class AgendaPartagéImpl implements Reunion, Absence, ... , AgendaPartagéAttributes{
3  public boolean confirmer_reunion(Date jour,Vector personne)
4          {...
5  if ( est_en_reunion((String)personne.elementAt(i),jour) ||
6                          est_absent((String)personne.elementAt(i),jour) )
7          ...
8          }
9  ...
10 }
```

Un exemple de SBDG est fourni dans la figure 3.8. Il représente une partie du SBDG du composant *Agenda-partagé*.

Il convient donc de noter que la propriété de polymorphisme liée à un code orienté objet ne permet pas d'identifier, par une analyse statique et de manière déterministe, tous les liens comportementaux existant entre les méthodes. Ainsi, nous avons choisi, par construction, qu'un SGBD inclut tous les liens comportementaux possibles existant entre les entités structurelles correspondant aux méthodes.

Une fois que le SBDG correspondant à chaque nouveau composant à générer est construit, une nouvelle description de la structure du composant est générée et le code source correspondant aux nouveaux composants générés est produit.

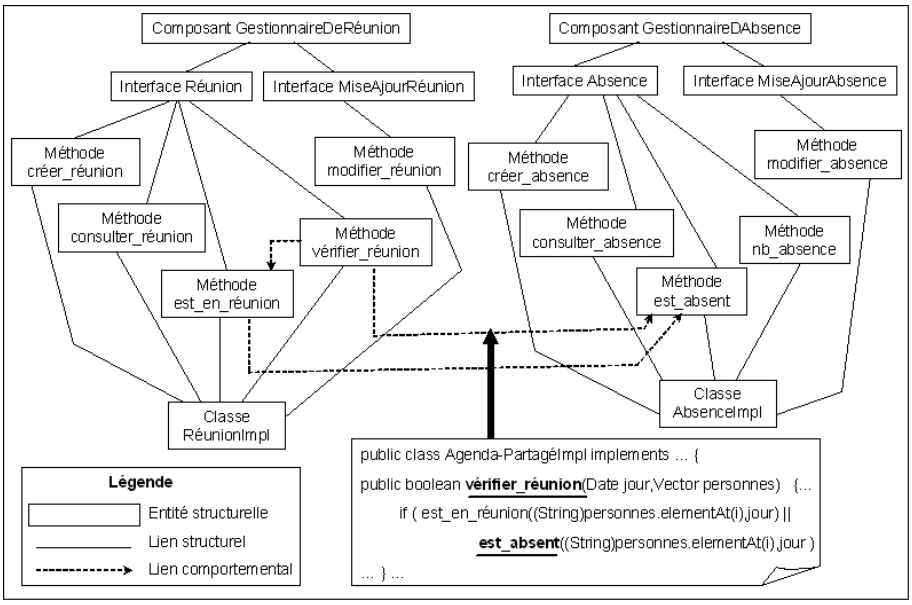

Figure 3.8 – Partie du graphe structurel et comportemental associé au composant *Agenda-partagé*

3.2.2.3 Génération de la structure externe des composants créés (niveau architectural)

Les nouveaux composants spécifiés dans la description de la structure résultat du processus de transformation sont créés en fragmentant le composant initial. Les interfaces fournies de chaque nouveau composant sont déterminées par l'analyse de la spécification du résultat de l'adaptation. En fait, la description de la structure résultat du processus est mise en correspondance avec la description de l'architecture de l'application afin d'extraire les propriétés de chaque interface (par exemple la signature, le type d'interface : standard ou collection, etc.). Concernant les interfaces requises, elles sont déterminées pendant l'étape d'assemblage (voir Section 3.2.3.1). Puis, une nouvelle description d'architecture correspondant à la structure du composant composite résultat de l'adaptation est produite. Celui-ci sera compilé et interprété par la plate-forme à composants utilisée afin de créer les instances des nouveaux composants générés.

Exemple de l'agenda-partagé

La description d'architecture générée à partir de la spécification décrite précédemment est la suivante :

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/standard.dtd">
4
5    <component name="GestionnaireDAgenda">
6      <interface name="It_agenda" role="server" signature="Agenda"/>
7      <interface name="It_miseajouragenda" role="server" signature="MiseAjourAgenda"/>
8    </component>
9
10   <component name="GestionnaireDeReunion">
11     <interface name="It_reunion" role="server" signature="Reunion"/>
12     <interface name="It_miseajourreunion" role="server" signature="MiseAjourReunion"/>
13   </component>
14
15   <component name="GestionnaireDAbsence">
```

```
16      <interface name="It_absence" role="server" signature="Absence"/>
17      <interface name="It_miseajourAbsence" role="server" signature="MiseAjourAbsence"/>
18      <interface name="It_droit" role="server" signature="Droit"/>
19      <interface name="It_miseajourdroit" role="server" signature="MiseAjourDroit"/>
20    </component>
```

3.2.2.4 Génération de l'implémentation orientée objet des nouveaux composants générés (niveau implémentatoire)

Tout d'abord, les classes d'implémentation de chaque nouveau composant sont générées. Puis, pour chaque classe d'implémentation, le code des méthodes correspondant aux services fournis par le composant est transféré de la classe d'implémentation du composant initial vers la nouvelle classe correspondante tout en garantissant l'intégrité du code au travers de pré-conditions et de post-conditions (voir Figure 3.9).

Interface_Transfer(c,i,c') :
Precondition :
is_Class(c) $\wedge$ is_Class(c') $\wedge$ is_Interface_of(c,i) $\wedge\neg$is_Interface_of(c',i) $\wedge$
($\forall$ m $\in$ Method(c) / is_Defined_in(i,m), $\neg$ is_Method_of(c',m))

Postcondition :
($\forall$ m $\in$ Method(c)$\cup$Method(c') / is_Defined_in(i,m), $\neg$ is_method_of(c,m) $\wedge$ is_reachable(c,m)) $\wedge$
($\forall$ m $\in$ Method(c)$\cup$Method(c'), is_accessible(m,get_Struct_Entities(m)) $\wedge$
($\forall$ r $\in$ Resource(c) / is_used(m,r), is_coherent(m,r))

Where :
Boolean is_Class(Class c) : return true iff c is a class.
Boolean is_Interface_of(Class c,Interface i) : return true iff the interface i is implemented by the class c.
Boolean is_Defined_in(Interface i, Method m) : return true iff the method m is defined in the interface i.
Boolean is_Method_of(Class c, Method m) : return true iff the method m is implemented in the class c.
Object[] Method(Class c) : return the list of method implemented in class c.
Object[] get_Struct_Entities(Method m) : return the list of structural-entities used in method m.
Boolean is_reachable(Class c, Method m) : return true iff the method m the reachable by the class c.
Boolean is_accessible(Method m, Object[] o) : return true iff all structural-entities contained in o are accessible by the method m.
Object[] Resource(Class m) : return the list of resources defined in the class m.
Boolean is_used(Method m,Object r) : return true iff the method m uses the resource r.
Boolean is_coherent(Method m,Object r) : return true iff the resource r state will be coherent when the method m will be invoke.

Figure 3.9 – Génération de l'implémentation du composant

Pour garantir l'intégrité du code généré, chaque service transféré doit pouvoir accéder à toutes les entités structurelles qu'il utilise. Ces entités sont de trois types :

1. *les méthodes correspondant à des services de composants*
   Concernant les méthodes correspondant à des services de composants, celles-ci sont des méthodes dont la signature est contenue dans une interface de composant. Deux cas peuvent être rencontrés lors de l'analyse du code source des méthodes de chaque composant généré : soit le service utilisé est également fourni par le même composant : dans ce cas l'intégrité structurelle est préservée ; soit le service utilisé est fourni par un autre composant : dans ce cas, il est nécessaire de mettre en place une liaison entre les deux composants mis en jeu. Cette opération est réalisée au moment de l'assemblage des composants générés (voir Section 3.2.3.1) ;

2. *les méthodes internes*
   Les méthodes internes à un composant sont des méthodes dont la signature est contenue dans au-

cune interface de composant. Ce sont des méthodes non accessibles de l'extérieur. Pour assurer l'intégrité structurelle et fonctionnelle des composants générés, les composants dont les services utilisent ces méthodes doivent pouvoir y accéder. Pour cela, deux solutions peuvent être envisagées :

- *la centralisée des méthodes internes*
  La première solution consiste à produire du code partagé. En fait, ce code sera présent dans une seule implémentation de composant mais devra être accessible pour les autres implémentations de composants le partageant.
  La mise en place d'une telle solution pour la gestion des méthodes internes implique la création d'un service associé à chacune de ces méthodes de manière à ce qu'elles soient accessibles à tous les composants qui leur font appel. Ces services pourront être fournis soit par un nouveau composant dédié, soit par un composant existant. De plus, ils doivent être considérés comme des services issus de l'adaptation et donc inaccessibles aux autres composants de l'application.
  Le choix du composant destiné à fournir un service correspondant à une méthode interne peut être déterminé en fonction des dépendances fonctionnelles. Par exemple, le composant dont les services ou les ressources ont le plus de dépendances fonctionnelles avec la méthode interne va fournir le nouveau service correspondant à celle-ci. Les autres sous-composants pourront alors accéder à cette méthode par l'intermédiaire du service ainsi créé.
  Cette solution permet de réduire au minimum la capacité mémoire occupée mais n'est pas optimale vis-à-vis des temps de réponse relatifs aux services des composants car le code partagé peut se trouver dans l'implémentation d'un composant déployé sur un autre nœud de l'infrastructure distribuée ;

- *la duplication des méthodes internes*
  Contrairement à la technique centralisée, l'approche avec duplication n'engendre pas la création de nouveaux services. En fait, le code de chaque méthode interne est dupliqué dans chaque composant pour lequel au moins un de ses services ou méthodes internes fait appel à cette méthode. Ainsi, tous les services disposeront des méthodes internes dont ils ont besoin pour fonctionner.
  Cette solution réduit au minimum la durée des communications entre les composants, étant donné que chaque composant dispose de tous les éléments nécessaires à son fonctionnement. Cependant, cette stratégie n'est pas optimale en ce qui concerne la capacité mémoire utilisée du fait de la duplication de code.

Le choix de la meilleure stratégie pour gérer l'éventuel partage des méthodes internes dépend des objectifs fixés pour l'application à déployer : réduire le temps CPU au minimum ou bien diminuer la mémoire utilisée. Nous avons opté, dans notre approche, pour une stratégie de gestion flexible. En fait, la politique de gestion du partage des entités structurelles est paramétrable à travers une interface d'administration. Ainsi, suivant le besoin, les acteurs de l'adaptation peuvent choisir l'une ou l'autre des stratégies ;

3. *les ressources logicielles*
   Comme nous l'avons évoqué précédemment, il existe un type particulier d'entités logicielles qui possèdent un état dont la persistance est supérieure à celle du service dans lequel elle est utilisée : les ressources logicielles. De la même manière que pour les méthodes internes, pour assurer le

maintien de l'intégrité structurelle relative à l'utilisant de ressources logicielles dans le corps de méthode, deux stratégies peuvent être envisagées :

- *une gestion centralisée des ressources logicielles*
  Cette stratégie peut se décliner sous deux formes : soit la mise en place d'un composant spécifique exclusivement dédié à la gestion des ressources partagées *i.e.* toutes les ressources logicielles et les méthodes pour y accéder sont exclusivement définies dans la classe d'implémentation associée à ce composant spécifique et tous les nouveaux composants générés doivent s'adresser à lui pour accéder à une ressource partagée ; soit la centralisation des ressources au niveau d'un seul composant généré *i.e.* la ressource est définie dans la classe d'implémentation d'un seul composant généré et pour y accéder, les autres composants doivent connaître le nom de ce composant ;

- *une gestion des ressources logicielles avec duplication*
  Une autre stratégie consiste à dupliquer les ressources partagées dans chaque classe d'implémentation des composants dont les méthodes les utilisent.

Dans tous les cas, le maintien de la cohérence de l'état des ressources partagées est nécessaire. Dans le cadre de notre approche, nous avons privilégié la stratégie de duplication des ressources partagées car elle assure une plus grande autonomie des composants logiciels générés. En effet, par exemple, si l'on opte pour une stratégie centralisée et qu'un composant définissant une ressource partagée devient inaccessible, le comportement des autres composants utilisant cette ressource n'est plus garanti. Alors que s'il contient une copie de la ressource partagée, il pourra continuer à fournir ses services en se basant sur l'état courant de la ressource.

Exemple de l'agenda-partagé
Si l'on se place dans l'exemple de l'agenda partagé, au départ le composant a pour ressource le nombre de jours libres d'une personne.

```
1  public class AgendaPartagéImpl implements Reunion, Absence, ... , AgendaPartagéAttributes{
2  ...
3  private int nb_jours_libres=0;
4  ...
5  }
```

Étant donné que cette ressource est utilisée par des méthodes des interfaces *Réunion* et *Absence*, elle va être dupliquée dans ces composants *GestionaireDeRéunion* et *GestionnaireDAbsences*.

```
1  public class GestionnaireDeAgendaImpl implements Agenda, AgendaAttributes, ... {
2  ...
3  private int nb_jours_libres=0;
4  ...
5  }
6
7  public class GestionaireDeAbsence implements Absence, AbsenceAttributes, ... {
8  ...
9  private int nb_jours_libres=0;
10 ...
11 }
```

La duplication des ressources va entraîner l'obligation de gérer la cohérence des copies. Pour cela, il faut mettre en place un système de gestion de cohérence et de synchronisation (voir Section 3.3).

Le code ainsi généré lors de cette étape de fragmentation représente la première version des codes sources des composants à générer. La prochaine étape consiste à transformer ce code de manière à prendre en compte les liens comportementaux existants entre les méthodes définies respectivement par deux SBDG différents. Cette transformation est réalisée pendant l'étape de recomposition.

3.2.3 Transformation du composant fragmenté en un composant composite

Une fois que le composant à adapter est fragmenté, le composant initial doit être recomposé (voir Figure 3.10) afin de préserver les invariants que nous avons fixés pour garantir le résultat de l'adaptation. Cette étape peut être décomposée en deux sous-étapes qui sont :

- *l'assemblage des nouvelles entités*
  L'étape de décomposition produit des composants déconnectés les uns des autres et fournissant chacun un sous-ensemble des services du composant initial. Cependant, ces services ne sont pas indépendants. En fait, ils peuvent être liés par des dépendances comportementales ou bien par des dépendances dues au partage de ressources. Les dépendances comportementales font références aux appels de méthodes. En fait, certaines méthodes peuvent faire appel à d'autres méthodes qui initialement étaient définies dans l'espace de nom du composant mais suite à la fragmentation ne le sont plus : elles sont maintenant définies dans d'autres composants. Ainsi, pour garantir la cohérence de l'assemblage, il est nécessaire de garantir que toutes les méthodes définies dans l'implémentation d'un composant puissent accéder aux méthodes qu'elles utilisent.
  Par ailleurs, nous avons vu que les ressources logicielles peuvent être dupliquées dans plusieurs implémentations associées à des composants générés : ce sont des ressources partagées. Ainsi, pour assurer la cohérence de l'assemblage des composants générés, il est indispensable de garantir la cohérence des états des ressources partagées. Pour cela, il est nécessaire d'introduire dans les composants partageant des ressources, des mécanismes de notification et de synchronisation ;

- *l'intégration du résultat de l'adaptation structurelle*
  Le résultat de la sous-étape précédente fournit un assemblage de composants dont le comportement est identique à celui du composant initial. Cependant, les propriétés d'interopérabilité et de transparence ne sont pas respectées. Ainsi, l'objectif de cette dernière étape est d'intégrer ces propriétés à l'assemblage. Le résultat de l'adaptation est encapsulé dans un composant dont la structure externe est identique à celle du composant initial.

3.2.3.1 Assemblage des composants générés

Comme nous l'avons mentionné précédemment, l'assemblage des nouvelles entités générées au cours de la fragmentation du composant initial a pour objectif de garantir l'intégrité et la cohérence du résultat de la transformation. La réalisation de cette tâche passe par :

- *le maintien de l'intégrité fonctionnelle des composants créés*
  Certains composants créés lors de l'étape de décomposition de notre processus de restructuration peuvent ne pas définir, dans leurs espaces de noms, certaines entités structurelles (par exemple, les méthodes) dont ils ont besoin pour l'exécution de leurs services. Ces entités sont en fait définies dans l'implémentation d'autres sous-composants générés lors de la fragmentation. Donc, pour garantir l'intégrité structurelle et fonctionnelle des composants créés, il est nécessaire de leurs

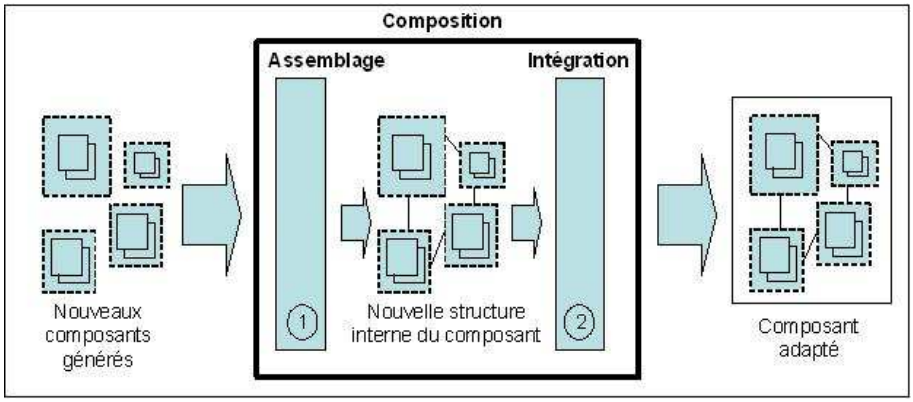

Figure 3.10 – Phase de recomposition du composant adapté

introduire des mécanismes leurs permettant d'accéder aux entités structurelles dont ils ont besoins pour fonctionner.
En fait, avant la fragmentation, une méthode pouvait invoquer une autre méthode de manière classique (*i.e.* appel de méthodes standard dans l'orienté objet par envoi de message). Cependant, après transformation, la méthode invoquée peut être devenue un service fourni par un autre composant généré. De ce fait, le maintien de l'intégrité passe par la transformation d'appels de méthodes en appels de services fournis par d'autres composants qui ont été générés. De plus, il est nécessaire d'introduire dans les composants concernés, de nouvelles interfaces leurs permettant d'accéder à ces services qui se trouvent fournis par d'autres composants. Nous appelons ces interfaces, des interfaces de communication. Leur mise en œuvre est décrite lors de la présentation de notre modèle de composants support de l'adaptation structurelle (voir Section 3.3) ;

- *le maintien de la cohérence entre les différents composants créés*
  Les nouveaux composants issus de l'étape de décomposition de notre processus de restructuration peuvent partager des ressources logicielles. Comme nous l'avons défini précédemment, une ressource possède un état qui peut être mis à jour et dont la persistance est supérieure à celle de la méthode dans laquelle elle est utilisée. De ce fait, l'état de chacune des ressources logicielles partagées doit rester cohérent pour maintenir la cohérence de l'assemblage de composants générés. Étant donné que nous avons choisi une stratégie de duplication des ressources partagées, leur gestion nécessite l'introduction de mécanismes permettant d'assurer la cohérence des états des différentes copies locales de ces ressources. Ces mécanismes de gestion de partage se chargent de deux tâches :

  1. *la communication des états de ressources entre les composants*
     La première tâche consiste à gérer les échanges entre les différents composants définissant au moins une ressource partagée afin de pouvoir préserver la cohérence de leur état. Cette tâche est accomplie au travers des interfaces de notification dont nous détaillerons la mise en œuvre lors de la présentation de notre modèle de composants support de l'adaptation structurelle (voir Section 3.3) ;

2. *la synchronisation des accès aux ressources par les composants.*
   La deuxième tâche concerne l'interdiction de l'accès multiple et simultané à une ressource. Cette tâche est réalisée au travers des interfaces de synchronisation dont nous détaillerons la mise en œuvre lors de la présentation de notre modèle de composants support de l'adaptation structurelle (voir Section 3.3).

3.2.3.2 Intégration du composant composite

Afin que le résultat du processus de transformation respecte les propriétés de transparence et d'encapsulation, l'assemblage obtenu précédemment est encapsulé dans un composant composite dont la structure externe et le comportement sont identiques à celui du composant avant sa transformation. De plus, nous introduisons dans le composite, des mécanismes permettant de configurer certaines propriétés liées à la composition.

Définition Un composant composite est un composant logiciel ayant des liens de composition avec d'autres composants logiciels, dénommés ses sous-composants (ou composants internes), qui décrivent chacun une de ses parties. Ces composants peuvent être à leur tour des composants-composites (*i.e.* un sous-composant peut être composé d'autres composants) ou des composants simples. Ainsi, un composite est avant tout un composant. Il est lié à ses sous-composants par la relation de composition.

En fait, les composants composites permettent de représenter des composants complexes par la composition de parties représentées par des composants ayant des liens structurels (*i.e.* services composés d'autres services), fonctionnels (*i.e.* services nécessitant l'utilisation d'autres services) ou plus généralement sémantiques (*i.e.* regrouper un ensemble de services suivant un même thème) entre eux. Les sous-composants peuvent être assemblés afin de remplir les services du composite. Le composant composite dispose de ses propres ports et interfaces. Les services fournis par un composant composite peuvent être délégués à un de ses sous-composants ou bien ils peuvent être propres au composite.

L'encapsulation de l'assemblage obtenu précédemment passe par la génération d'une membrane qui va permettre de masquer la structure interne du composant résultat de l'adaptation et la mise en œuvre de mécanismes permettant d'assurer les communications entre cette membrane et les sous-composants concernés[1]. Ces deux tâches sont étroitement liées au modèle de composants utilisé. Dans le cadre du modèle de composants Fractal, l'encapsulation passe uniquement par une modification de la description de l'architecture de l'application contenant le composant adapté : définition du composite (désignation, interfaces fournies et requises définies, etc.) et par la mise en place de liaisons d'exportation entre le composite et ses sous-composants.

Exemple de l'agenda-partagé

Le composant *Agenda-partagé* encapsule les composants générés (*GestionnaireDeReunion*, *GestionnaireDAbsence*, etc.) (lignes 5-25). Il définit les mêmes interfaces fournies et requises que le composant initial (lignes 6-8) qui sont liées aux interfaces fournies par les sous-composants correspondants (lignes 22-24). Ainsi, la nouvelle description de l'architecture du composant composite obtenu après adaptation du composant *Agenda-partagé* est la suivante :

[1] La mise en œuvre de ces mécanismes est réalisée au travers des interfaces de communication entre le composite et ses sous-composants, que nous détaillerons lors de la présentation de notre modèle de composants structurellement adaptés (voir Section 3.3)

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/standard.dtd">
4
5  <definition name="AgendaPartagé">
6    <interface name="It_reunion" role="server" signature="Reunion"/>
7    <interface name="It_absence" role="server" signature="Absence"/>
8    ...
9    <interface name="It_Mail_sender" role="client" signature="Mail_sender"/>
10   <component name="GestionnaireDeReunion">
11     <interface name="It_reunion" role="server" signature="Reunion"/>
12     <interface name="It_absence" role="client" signature="Absence"/>
13     <interface name="It_Mail_sender" role="client" signature="Mail_sender"/>
14     ...
15   </component>
16
17   <component name="GestionnaireDAbsence">
18     <interface name="It_absence" role="server" signature="Absence"/>
19     ...
20   </component>
21
22   <binding client="GestionnaireDeReunion.It_absence" server="GestionnaireDAbsence.It_absence"/>
23   ...
24   <binding client="this.It_reunion" server="GestionnaireDeReunion.It_reunion"/>
25   <binding client="this.It_absence" server="GestionnaireDAbsence.It_absence"/>
26   ...
27   <binding client="GestionnaireDeReunion.It_Mail_sender" server="this.It_Mail_sender"/>
28   ...
29 </definition>
```

Nous pouvons noter que dans le cadre du modèle de composants Fractal et de son implémentation Julia, aucune modification au niveau implémentatoire n'est nécessaire pour la mise en œuvre des interfaces de communication entre un composite et ses sous-composants.

L'encapsulation permet ainsi de masquer les services issus de l'adaptation (*i.e.* filtrage de l'accès aux services). En fait, le composant fournit les mêmes services fonctionnels que ceux disponibles avant son adaptation. Par ailleurs, il peut intégrer de nouveaux services non fonctionnels destinés à manipuler ses sous-composants. Ces services permettent, par exemple le paramétrage d'un déploiement distribué de ce nouveau composant composite ou bien la possibilité de réaliser une nouvelle adaptation. La figure 3.11 présente le composant composite associé au composant *Agenda-partagé*.

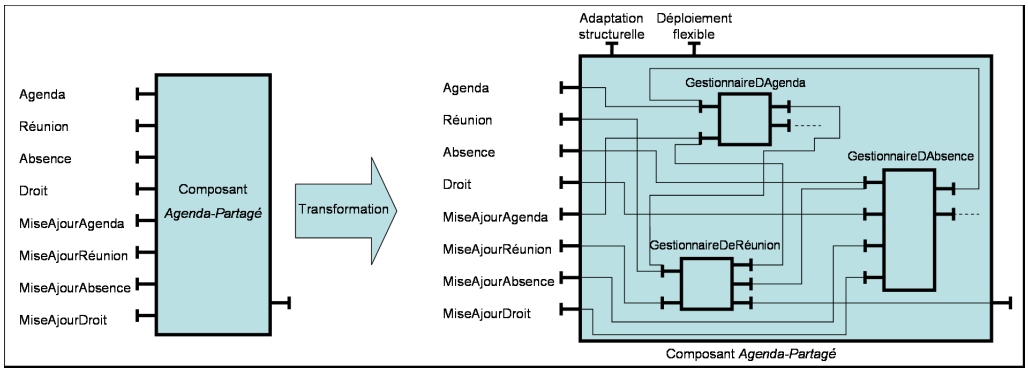

Figure 3.11 – Résultat de la transformation du composant *Agenda-partagé* en composant composite

3.2.4 Configuration et enrichissement du composant résultat des transformations

Une fois que le composant composite dont la structure interne correspond aux nouvelles attentes, a été généré, nous proposons une étape optionnelle qui consiste à configurer certaines propriétés du composite liées à la relation de composition et à enrichir le composite de mécanismes de distribution.

3.2.4.1 Configuration du composant composite issu de la transformation structurelle

Comme nous l'avons évoqué précédemment, afin d'introduire de nouveaux points de variabilité, nous proposons de fournir à un acteur de l'adaptation des mécanismes lui permettant de configurer certaines propriétés du composite résultat de l'adaptation afin qu'il réponde à plus de besoins liés à son utilisation. Les propriétés que nous proposons de rendre configurables font références à la relation de composition entre le composite et ses sous-composants [23, 127]. Elles peuvent être configurées de manière à introduire des facilités d'adaptation ou à gérer la politique de sécurité mise en place par un acteur externe de l'adaptation. Les propriétés configurables sont les suivantes :

- *la cardinalité* et *la cardinalité inverse*
  la cardinalité représente le nombre de sous-composants pouvant appartenir à un objet composite via un lien de composition. Si ce nombre est égal à 0, cela signifie que la composition de composants est impossible. Si ce nombre est supérieur à 1, cela implique que la composition de composants est possible.
  La cardinalité inverse correspond au nombre de composites qui pourront se partager un même sous-composant via un lien de composition. Si ce nombre est strictement supérieur à 1, cela implique que le partage des sous-composants est possible. C'est-à-dire qu'un composant référencé par un lien de composition partagé pourra être référencé par un nombre quelconque de liens de composition partagé et donc pourra faire partie de plusieurs composites simultanément. Si ce nombre est égal à 1, cela signifie l'exclusivité de chaque sous-composant à son composite. Dans ce cas, un composant référencé par un lien de composition exclusif ne peut être référencé que par ce seul lien de composition et ne peut donc faire partie que d'un seul composite. Enfin, si ce nombre est égal à 0, cela signifie que la composition de composants est impossible ;

- *le niveau de composition*
  cette propriété permet de spécifier le niveau de composition possible. Si ce nombre est égal à 0, cela signifie que la composition de composants est impossible.
  La limitation du nombre de composition possible peut permettre de diminuer les risques de conflits liés à la propagation des valeurs des attributs et permet d'augmenter la compréhension de la structure du composant ;

- *le degré d'encapsulation des sous-composants*
  cette propriété fait référence à la visibilité et à l'accessibilité des sous-composants vis-à-vis des autres composants de l'application (*i.e.* autres que ceux générés au cours de la fragmentation du composant initial). Il existe quatre configurations possibles pour cette propriété :

  1. un composant « boîtes blanches »
     Tout d'abord, le composite peut être considéré comme une boîte blanche (voir Figure 3.12, cas A). Dans ce cas tous ses sous-composants sont visibles et accessibles directement par les

autres composants de l'application. Cette stratégie de composition implique une très faible encapsulation des sous-composants. Elle est généralement utilisée pour regrouper un ensemble de services fonctionnellement indépendants mais sémantiquement proches dans un même composant ;

2. un composant « boîte noire »
   Dans le cas d'un composite boîte noire (voir Figure 3.12, cas B), ses sous-composants ne sont ni visibles ni accessibles par les autres composants de l'application. Tous les appels des services des sous-composants provenant d'un composant externe doivent passer par le composite pour accéder aux services fournis par les sous-composants. Les sous-composants sont donc totalement encapsulés dans le composant composite. Tout composant interagissant avec le composite ignore la manière dont il est constitué. Étant donné qu'il est obligatoire de passer par le composite, il est donc possible de mettre en place très facilement des outils de traitement des messages entrant et sortant provenant de l'extérieur du composant (*i.e.* autres composants de l'application) et ainsi faciliter la maintenance et l'adaptation de l'application. De plus, la sécurité est un atout majeur des boîtes noires car les composants internes sont inaccessibles directement et donc protégés ;

3. un composant « boîte grise »
   Lorsqu'un composite est spécifié comme étant une boîte grise (voir Figure 3.12, cas C), sa structure interne est visible par les autres composants de l'application mais ses sous-composants ne sont pas accessibles directement. Seuls les services fournis par le composite sont utilisables. En termes d'accessibilité, un composant « boîte grise » offre les mêmes caractéristiques qu'un composant « boîte noire » ;

4. un composant « boîte mixte »
   Si le composite est considéré comme une boîte mixte (voir Figure 3.12, cas D), certains sous-composants sont accessibles directement par les autres composants de l'application mais pas tous.

Cette propriété d'encapsulation peut être utilisée pour masquer certains sous-composants ou services en fonction de la politique de sécurité établie. Par exemple, concernant le composant *Agenda-partagé*, le sous-composant *GestionnaireDAgenda* contenant les agendas personnels, peut être configuré, en utilisant cette propriété (*i.e.* avec la valeur « boîte mixte »), comme étant invisible et inaccessible pour les autres composants de l'application.

- *l'accessibilité interne*
  la propriété d'accessibilité interne permet de spécifier comment un sous-composant accède à un autre sous-composant du composite. Cette propriété peut être configurée de la manière suivante : l'accès peut être effectué via le composant composite ou bien via une référence directe vers le sous-composant en question. Nous distinguons trois stratégies possibles :

  1. un accès direct
     La première stratégie consiste à autoriser un accès direct aux services d'un autre sous-composant du composite (voir Figure 3.13, cas A) ;

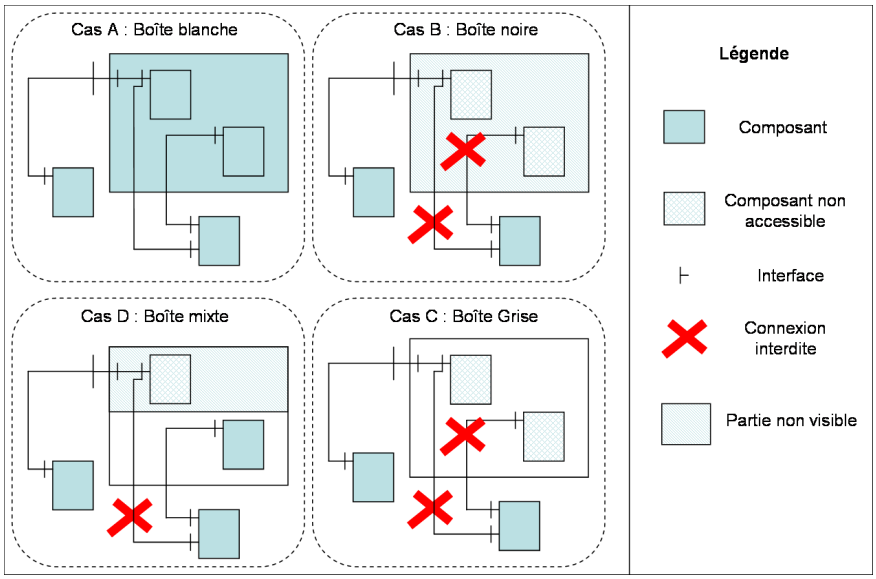

Figure 3.12 – Configuration du degré d'encapsulation des sous-composants du composite résultat de l'adaptation structurelle

2. un accès par les services du composite
   La deuxième stratégie consiste à passer par le composite pour accéder à un service d'un autre sous-composant (voir Figure 3.13, cas B). Ainsi, les messages échangés entre les sous-composants doivent traverser la membrane du composite puis sont redirigés au travers d'un lien d'exportation vers le service invoqué (*i.e.* délégation d'un service du composite vers un service d'un sous-composant). Dans ce cas, l'encapsulation des sous-composants est faible car ces derniers sont capables de traverser la membrane pour aller se connecter à un port (ou une interface) fourni par le composite. Cette approche est donc peu compatible avec la stratégie boîte noire où l'encapsulation est forte (*i.e.* les messages échangés à l'intérieur du composite ne doivent pas transiter par l'extérieur de ce dernier) ;

3. un accès par l'implémentation non-fonctionnelle du composite (*i.e.* membrane du composite)
   La dernière solution réside dans l'utilisation de la membrane du composite comme connecteur entre les sous-composants (voir Figure 3.13, cas C). Cette solution permet de traiter facilement les messages échangés entre les sous-composants (*i.e.* possibilité d'adaptation en redirigeant les messages échangés) où même faciliter certaines tâches comme la gestion de la concurrence et de la synchronisation relatives aux ressources partagées par plusieurs sous-composants. En fait, elle permet de centraliser les contrôleurs du composite au niveau de sa membrane.

Cette propriété peut être utilisée pour faciliter l'insertion de code d'adaptation fonctionnel. En effet, si l'envoi de message entre les sous-composants transite par le composite, il devient plus facile de contrôler ces messages et de les modifier, si besoin est. Cependant, une telle stratégie peut entraîner une dégradation des performances du composant si le nombre de messages échan-

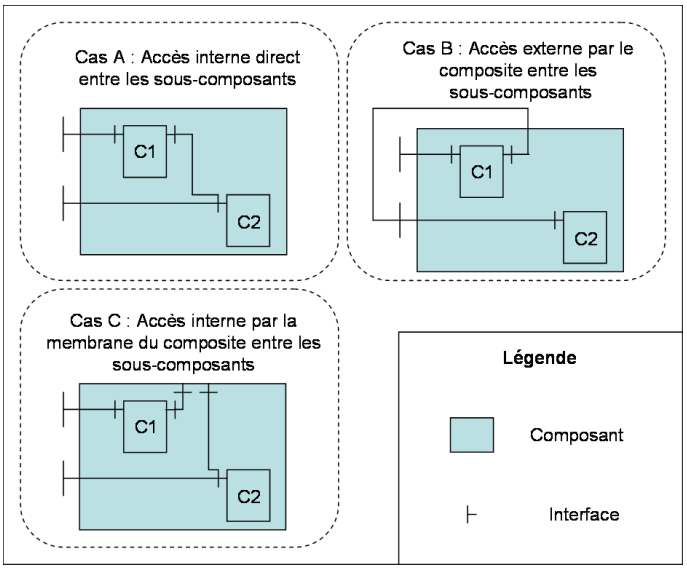

Figure 3.13 – Configuration de la propriété d'accessibilité interne aux services des sous-composants du composite issu de l'adaptation structurelle

gés est trop important. De ce fait, une telle stratégie ne peut être envisagée que si l'application le permet. Ainsi, la configuration de cette propriété dépend fortement de l'application qui contient le composant adapté donc elle ne doit être réalisée que par un acteur extérieur de l'adaptation ;

- *le cycle de vie du composite et de ses sous-composants*
  la gestion du cycle de vie est relative aux différentes stratégies envisageables lors de l'instanciation, de l'arrêt (*i.e.* dépendances comportementales) ou de la suppression (*i.e.* dépendances existentielles) du composite ou bien de l'un de ses sous-composants.

  – *Instanciation du composite résultat de l'adaptation*
    La propriété d'instanciation des composants fait référence à la stratégie d'instanciation du composite et de ses sous-composants adoptée lors du déploiement de l'application. L'approche choisie peut être descendante (*i.e.* d'abord le composite est instancié, ensuite les sous-composants), ascendante (*i.e.* d'abord les sous-composants ensuite le composite), mixte ou bien dynamique. La stratégie adoptée est étroitement liée au type de composant manipulé (boîte blanche, boîte grise, boîte noire ou mixte) :

    1. une approche ascendante
       L'instanciation ascendante d'un composant logiciel impose la création des instances de ses sous-composants avant celle du composite. Cette stratégie induit une certaine indépendance du composite vis-à-vis de ses sous-composants. L'instanciation ascendante est généralement utilisée pour assembler des composants existants et les encapsuler au sein d'un composite. Tant que le composite n'est pas instancié les sous-composants sont visibles et accessibles directement ; ce qui posent des problèmes de sécurité si le composant composite n'est pas spécifié comme étant « boîte blanche » ;

2. une approche descendante
   Lors d'une instanciation descendante, le composite est instancié avant ses sous-composants. Une telle approche induit une forte encapsulation des sous-composants dans le composite. Contrairement à l'instanciation ascendante, si les acteurs externes de l'adaptation le souhaitent, à aucun moment il est possible d'accéder aux instances des sous-composants. Cette approche est donc à privilégier pour la création de composants « boîtes noires » ou de composants « boîtes grises » ;

3. une approche mixte
   Une stratégie mixte entre l'instanciation ascendante et descendante peut être réalisée. Par exemple, si l'on prend le cas des composants mixtes où l'on à un ensemble de sous-composants visibles et accessibles ainsi qu'un ensemble de sous-composants cachés et inaccessibles, l'on peut instaurer une stratégie d'instanciation des sous-composants qui serait la suivante : tout d'abord, on instancie les composants visibles (*i.e.* ce qui permet d'utiliser des composants déjà instanciés), ensuite le composite et pour finir les sous-composants restants (*i.e.* composants cachés). En fait cette stratégie consiste à instrumenter les sous-composants et le composite dans n'importe quel ordre ;

4. une approche par instanciation dynamique des sous-composants
   Une stratégie d'instanciation dynamique des composants permet d'instancier un composant dès le premier appel à un de ses services. Tant qu'un composant n'est pas utilisé, il n'est pas instancié. Cette méthode peut se révéler judicieuse pour des applications destinées à être exécutées dans des environnements à faibles ressources. Plusieurs stratégies peuvent être établies en fonction du moment de création de l'instance du composite. Celle-ci peut être créée dès l'appel à n'importe quel service de l'un de ses sous-composants ou bien uniquement lorsque l'un de ses services est appelé.

Le choix de la stratégie d'instanciation des composants dépend d'une part, de l'environnement de déploiement de l'application (par exemple, dans le cadre d'environnement à ressources limitées, il est préférable d'opter pour une instanciation dynamique) et d'autre part de la politique de sécurité mise en place par l'administrateur de l'application (par exemple, contrairement à l'instanciation ascendante, l'approche descendante permet de garantir l'encapsulation totale des sous-composants ; de ce fait, la sécurité en est renforcée).

– Gestion des dépendances existentielles
  Les dépendances existentielles se traduisent par deux propriétés : d'une part, la dépendance des liens de composition (*i.e.* propriété permettant d'établir le processus à appliquer lors de la suppression du composite) et, d'autre part, la prédominance des liens de composition (*i.e.* propriété permettant de spécifier le processus à mettre en œuvre lorsqu'un sous-composant est supprimé). L'existence des composites est fortement liée à celle de ses composants et réciproquement.

  1. Propriété de dominance existentielle des liens de composition
     Cette propriété permet d'établir le processus à appliquer lors de la destruction du composite. Plusieurs stratégies sont possibles (voir Figure 3.14) :

(a) la destruction totale du composant
La première solution consiste à détruite tous les sous-composants du composite (voir Figure 3.14, cas 1). Dans ce cas, le composite est vu comme une unité indivisible. Si un de ses sous-composants est détruit, le composite n'a plus de sémantique et donc l'existence de ses autres sous-composants est mise en question. Cette stratégie à forte sémantique de la composition peut être couplée avec les propriétés de boîtes noires ou de boîtes grises où la préservation de la sécurité est mise en avant. Étant donné que les sous-composants n'étaient pas accessibles directement, leur destruction permet de conserver cette propriété ;

(b) la destruction de la membrane
La deuxième stratégie réside uniquement dans la destruction de la membrane du composite (voir Figure 3.14, cas 2). Dans ce cas, le composite n'existe plus mais les sous-composants deviennent des composants (*i.e.* pouvant être composite) accessibles directement par les autres composants de l'application. Cette stratégie n'est envisageable que pour les composites boîtes blanches car ses sous-composants peuvent être accessibles directement ;

(c) la destruction partielle du composant
La dernière solution consiste à détruire une partie des sous-composants (voir Figure 3.14, cas 3). Les composants non détruits restent accessibles directement. Cette solution peut être mise en œuvre pour les composites boîte blanche où les composants à conserver seront spécifiés parmi l'ensemble de tous les sous-composants mais également pour les composants boîtes grises où les composants à conserver seront spécifiés parmi l'ensemble des sous-composants accessibles de l'extérieur du composite (*i.e.* les composants inaccessibles seront automatiquement détruits).

2. Propriété de prédominance existentielle des liens de composition
La propriété de prédominance des liens de composition fait référence à la relation entre un sous-composant et son composite. Elle permet de spécifier le processus à mettre en œuvre lorsqu'un sous-composant est détruit. Les stratégies possibles diffèrent essentiellement par la conservation ou la destruction du composite (voir Figure 3.15).
Tout d'abord, le choix de destruction du composite présuppose une forte relation sémantique entre les sous-composants. Ainsi, si un sous-composant est supprimé, le composite n'a plus de sens d'un point de vue sémantique (*i.e.* encapsulation forte). Cette stratégie peut avoir des répercutions sur la propriété de dépendance. En effet, la suppression d'un sous-composant peut donc entraîner la suppression de tous les autres sous-composants du composite (*i.e.* le composite sera totalement détruit).
Une autre stratégie consisterait à détruire uniquement le sous-composant concerné et à conserver le composite. Cependant, la destruction d'un sous-composant peut avoir des répercutions dans le reste du composite au niveau des services. Par exemple, si le sous-composant fait partie d'un assemblage permettant de fournir un service, ce dernier doit être supprimé des services fournis par le composite. De plus, si d'autres sous-composants utilisent un des services fournis par le composant supprimé ou bien sont utilisés exclusivement dans le cadre d'un service nécessitant le composant supprimé, il peut être envisageable de

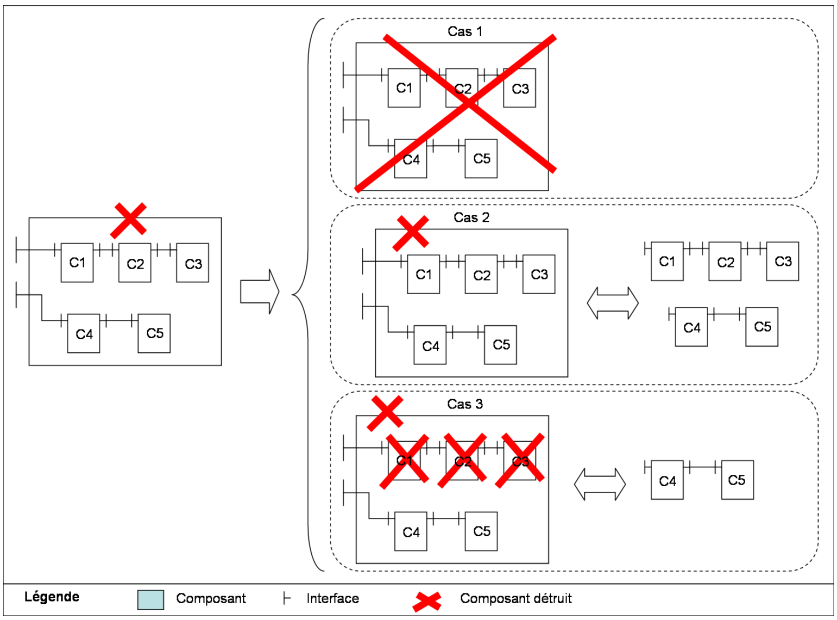

Figure 3.14 – Configuration des propriétés de dominances existentielles du composite issu de l'adaptation structurelle

supprimer les services faisant appels au moins à un service du composant supprimé ou bien de supprimer les sous-composants nécessitant le composant supprimé et ce récursivement. Par ailleurs, si le composite est boîte blanche, il doit pouvoir être possible de conserver les composants internes déconnectés.

– Gestion des dépendances comportementales
  Concernant les dépendances comportementales, le principe est le même que pour les dépendances existancielles à la différence qu'au lieu d'étudier les conséquences de la suppression du composite ou de l'un de ses sous-composants, sont spécifiées les répercussions de l'arrêt du composite ou de l'un de ses sous-composants.

- *la propagation des services*
  un composite peut propager certains de ses services[2] vers un de ses sous-composants et vice-versa. Cette propriété relative à la propagation des services peut être configurée de la manière suivante : tout d'abord, du composite vers ses sous-composants (*i.e.* un service du composite est partagé par ses sous-composants), ou bien d'un sous-composant vers son composite (*i.e.* un service d'un sous-composant est partagé avec son composite).
  Cette propriété peut être utilisée notamment dans le cadre de composants « boîtes noires » afin d'autoriser l'accès à un service d'un sous-composant par l'intermédiaire du composant (*i.e.* ce service sera alors partagé entre le sous-composant et le composite). Également, dans le cadre de composants « boîtes blanches », la possibilité d'invoquer un service du composite en accédant directement à un sous-composant sera donnée à l'utilisateur (*i.e.* propagation du sous-composant vers le composite).

[2]Les services non fonctionnels ainsi que ceux créés lors de la transformation ne peuvent pas être propagés.

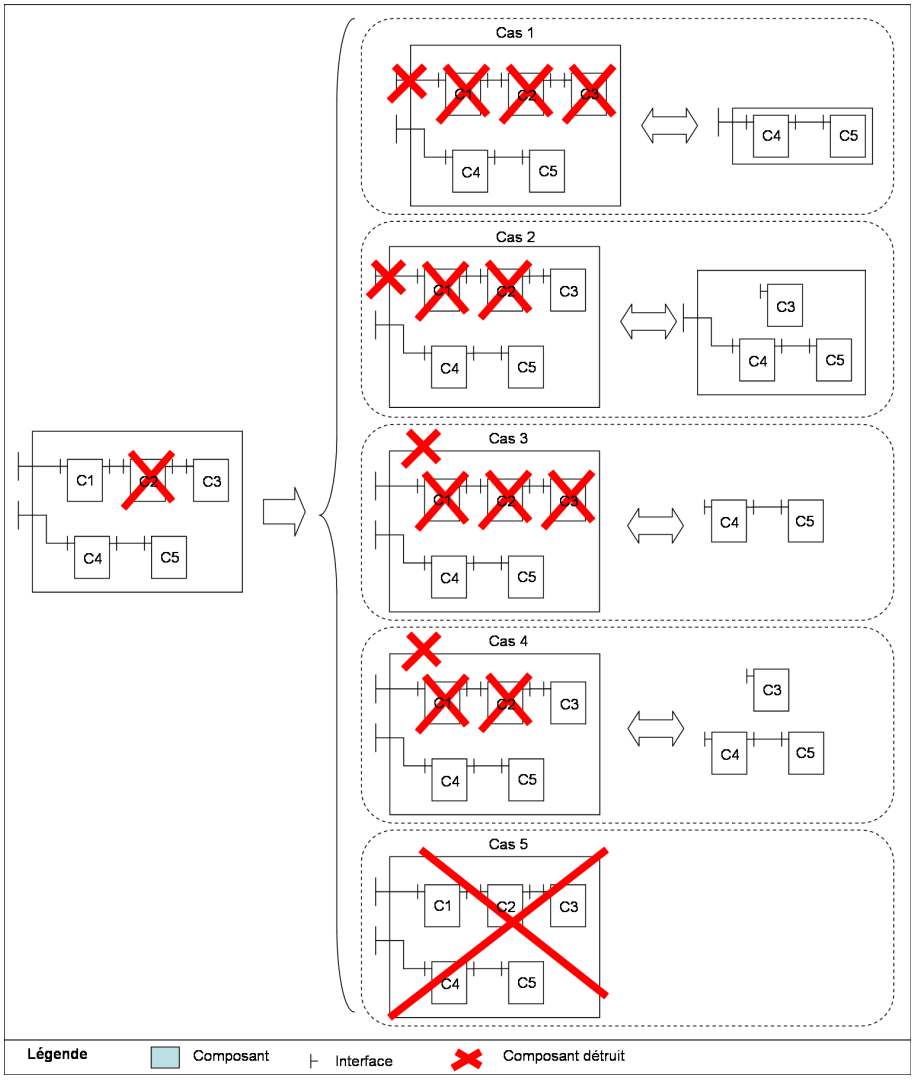

Figure 3.15 – Configuration des propriétés de prédominances existentielles du composite issu de l'adaptation structurelle

3.2.4.2 Intégration de mécanismes de distribution dans le composite issu de la transformation structurelle

Nous avons pu constater que parmi les motivations de l'adaptation structurelle, la majorité envisage une distribution du composant après sa restructuration. Or, tel qu'il est obtenu en appliquant le processus de transformation structurelle que nous avons défini précédemment, un composant ne peut être distribué sur plusieurs sites car tous ses sous-composants sont encore liés au travers de liaisons locales. En fait, dès lors qu'il existe des composants distants, certaines ressources ou services ne peuvent plus être accédés via des références directes. Il devient alors indispensable de gérer les communications entre les composants locaux et distants (*i.e.* les sous-composants sont reliés par des liaisons pouvant être des références locales ou distantes entre les interfaces fournies et requises).

Afin de créer un composant composite distribué[3] nous devons, dans un premier temps, proposer un modèle de distribution permettant d'assurer les communications entre les sous-composants déployés sur des sites différents. Ce modèle doit être un modèle étendu du modèle de composants structurellement adaptés (modèle de composants correspondant au résultat du processus de transformation structurelle). Le modèle que nous avons conçu, est détaillé dans la section 3.3. Son principe réside dans la création de plusieurs copies du composite à distribuer ; chacune devant être déployée sur un site. Chaque copie présente la même structure externe que le composant initial. Cependant, la structure interne de chacune est différente. En fait, chaque copie contient un ensemble de composants locaux (ceux destinés à être déployés sur la machine en question) et un ensemble de composants dit « virtuels » qui sont chargés d'assurer les communications entre les différentes copies du composite (qui sont déployées sur des sites différents). Ces derniers sont couplés à des composants spécifiques dits « de contrôle » (non fonctionnels) chargés de réaliser les communications de bas niveaux : un composant de transport pour gérer les protocoles réseaux (TCP/IP, UDP, etc.) et un composant de nommage pour gérer l'adressage des copies du composite.

Pour obtenir un composant conforme au modèle de composants structurellement adaptés et distribués, à partir d'un composant résultat de notre processus de transformation structurelle, nous devons étendre ce dernier afin d'introduire les mécanismes de distribution (voir Figure 3.35). Une telle extension passe par l'intégration de trois nouvelles étapes :

- *spécification de la configuration de déploiement*
  Tout d'abord, un acteur externe de l'adaptation doit spécifier la nouvelle configuration de déploiement distribuée du composant ; pour chaque composant, il faut spécifier son site d'implantation. Cette étape peut être réalisée lors de la spécification de la nouvelle structure du composant à obtenir à l'issue du processus de transformation structurelle ;

- *génération des différentes copies du composant structurellement adapté*
  Une fois que le processus de transformation structurelle est terminé, il faut générer les différentes copies du composant structurellement adapté qui vont être déployées en fonction de la spécification fournie. Cette opération passe par la réalisation de trois sous-étapes qui sont :

  1. *le paquetage des différentes copies d'un composant composite distribué*
     La première étape de notre processus consiste à séparer le code source correspondant aux composants locaux et à les intégrer dans les différentes copies du composant distribué.

[3] Composant dont les sous-composants peuvent être déployés sur différents sites.

Comme nous l'avons évoqué précédemment, le code de ces composants n'est pas modifié, cependant, il doit être intégré au paquetage correspondant aux copies les contenant. Cette opération est réalisée par l'analyse de la spécification contenant la nouvelle structure du composant à générer. Pour chaque site de déploiement, un paquetage contenant les composants locaux qui doivent y être déployés est créé ;

2. *la génération des composants virtuels*
   Une fois que les paquetages correspondant aux copies du composant composite distribué ont été construits, il faut générer les composants virtuels. Cette opération est réalisée par l'analyse des composants locaux correspondant aux composants virtuels et par la génération de code. En fait, le code correspondant à un composant virtuel a la même structure que celui du composant local qu'il représente. Seul le code des méthodes correspondant aux services qu'il implémente est modifié : il est remplacé par du code de contrôle permettant d'invoquer le service, en question, fourni par un composant local appartenant à une autre copie du composite distribué et déployé sur un site distant. Lors de cette étape, du code d'adaptation peut être introduit avant ou après l'invocation ;

3. *l'insertion des composants de contrôle*
   Enfin, la dernière étape de notre processus de génération du composite distribué consiste à introduire dans chacune des copies, générées précédemment, des composants de contrôles (composant de transport et composant de nommage) permettant aux composants virtuels d'assurer la communication entre les différentes copies existantes.

- *déploiement des différentes copies*
  Une fois que les différentes copies du composite ont été générées, elles sont déployées sur les nœuds de l'infrastructure distribuée disponible en fonction de la spécification fournie précédemment. Cette opération peut être réalisée manuellement par un acteur de l'adaptation ou bien automatiquement par un outil spécifique tel que InstallShield Developer [49] ou bien Java Web Start [121].

3.3 Modèle de composants support de l'adaptation structurelle

Comme nous l'avons vu précédemment, la réalisation du processus d'adaptation structurelle entraîne la ré-ingénierie du composant à adapter. Le résultat de ce processus est un composant structurellement adapté. Un tel composant est en fait un composant composite dont la structure externe (ports, interfaces fournies et requises, etc.) et le comportement sont identiques à ceux du composant initial. Aussi, afin de garantir ces obligations, le composant composite issu du processus de ré-ingénierie doit être conforme au modèle de composants structurellement adaptés que nous définissons ci-dessous.

3.3.1 Présentation de notre modèle de composants support de l'adaptation structurelle

Tout d'abord, notre modèle de composants structurellement adaptés doit être un modèle de composants hiérarchiques. En fait, un composant structurellement adapté est un composant dont la structure externe est la même que celle du composant initial (*i.e.* les services fonctionnels fournis par le composant adapté sont identiques à ceux du composant initial) mais sa structure interne est, quant à elle, différente

étant donnée qu'elle devient conforme à une spécification fournie par un acteur externe de l'adaptation. Ainsi, un composant structurellement adapté contient un ensemble de sous-composants (ceux spécifiés par un acteur de l'adaptation et générés par la fragmentation du composant initial) fournissant des services (chaque sous-composant fournit un sous ensemble de services fournis par le composant initial) et nécessitant d'autres services pour fonctionner (ces services peuvent être soit fournis par d'autres composants issus de la fragmentation, soit requis par le composite).

Afin d'assurer l'intégrité fonctionnelle et structurelle des sous-composants ainsi que le partage de ressources, les sous-composants doivent être dotés de nouveaux mécanismes assurant cette intégrité. Comme nous l'avons mentionné précédemment, l'introduction de ces mécanismes se traduit par l'insertion de nouvelles interfaces non-fonctionnelles au niveau des sous-composants. Ces interfaces sont de deux types :

- *les interfaces de communication*
  Lors du partitionnement de l'implémentation du composant initial, nous avons pu constater que l'intégrité fonctionnelle des composants générés n'est pas garantie. En effet, certaines méthodes dans l'implémentation d'un composant peuvent invoquer des méthodes qui sont devenues des services fournis par d'autres composants générés. De ce fait, il est indispensable d'introduire pour chaque composant un mécanisme lui permettant de réaliser ces invocations. L'introduction de ces mécanismes passe par la définition de nouvelles interfaces chargées de la communication entre les composants générés lors de la fragmentation du composant initial. Ces interfaces sont appelées interfaces de communication ;

- *les interfaces de coordination*
  Lors du partitionnement de l'implémentation du composant initial, certaines ressources pouvaient être utilisées par plusieurs nouveaux sous-composants générés. De ce fait, pour assurer la cohérence du composant issu de l'adaptation, il est nécessaire d'introduire dans les sous-composants générés des mécanismes de gestion de ressources partagées. Ces mécanismes sont introduits sous la forme d'interfaces de coordination.

Par ailleurs, afin de contrôler l'encapsulation de l'assemblage précédemment obtenu, nous devons introduire au niveau du composant composite des interfaces non fonctionnelles lui permettant d'une part, de gérer les nouveaux sous-composants et d'autre part de paramétrer certaines propriétés configurables et notamment, celles liées à la relation de composition. Ce paramétrage a pour objectif d'optimiser le fonctionnement du composite généré en termes de flexibilité et d'adaptabilité (*i.e.* fournir des facilités en vue d'une éventuelle future adaptation fonctionnelle). Il est mis en œuvre au travers des interfaces de configuration.

Notre modèle de composants structurellement adaptés répondant à ces contraintes est décrit dans la figure 3.16. Nous le détaillerons dans les sections suivantes.

3.3.2 Gestion des dépendances fonctionnelles entre composants : les interfaces de communication

La gestion des dépendances fonctionnelles entre les composants générés lors de la fragmentation est réalisée par l'intermédiaire d'interfaces dédiées appelées interfaces de communication.

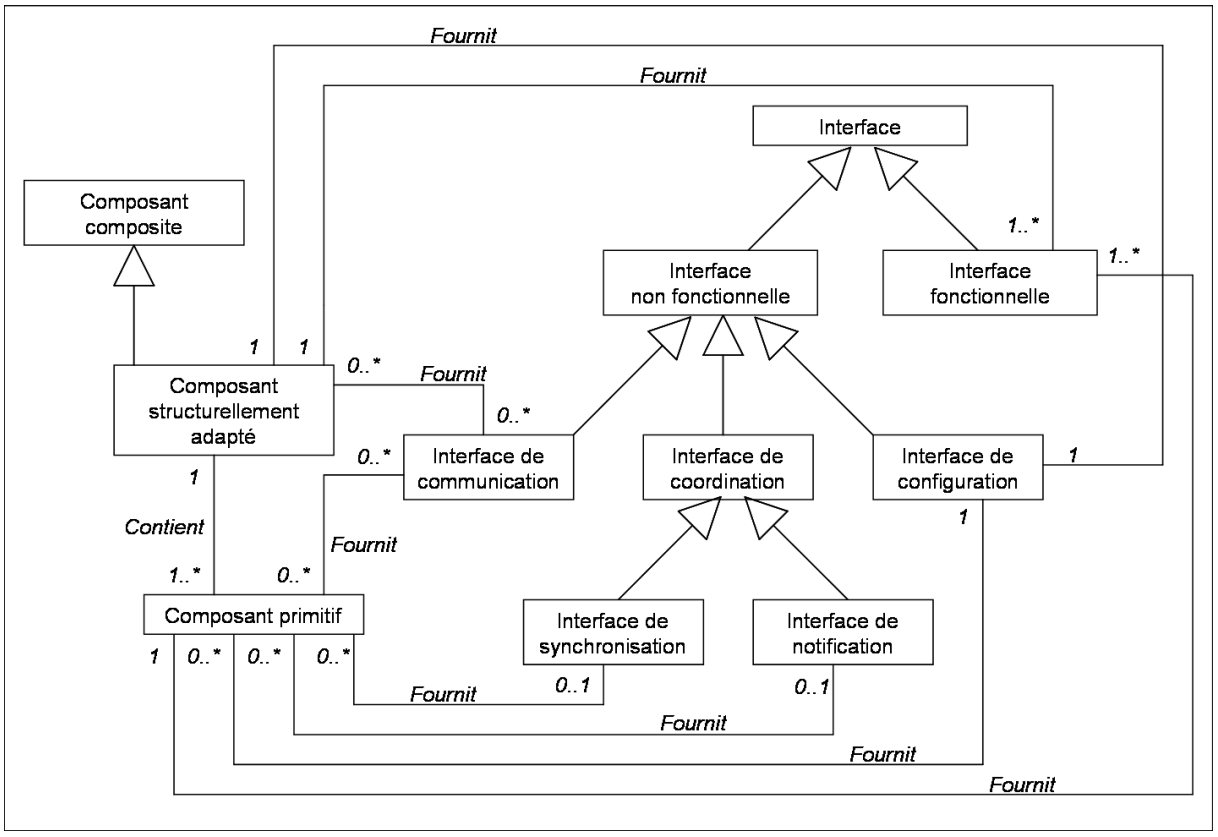

Figure 3.16 – Modèle de composants structurellement adaptés

3.3.2.1 Présentation des interfaces de communication

Les interfaces de communication sont chargées d'assurer l'intégrité fonctionnelle des composants issus de la fragmentation du composant initial. En fait, elles permettent à un composant généré d'accéder aux entités structurelles qu'il a besoin pour fonctionner et qui sont définies dans l'implémentation d'autres composants qui ont été générés lors de la fragmentation.

Exemple de l'agenda-partagé
Dans le cadre de notre exemple d'agenda-partagé, le composant *GestionnaireDAgenda* est lié structurellement au composant *GestionnaireDeReunion* parce qu'il utilise le service proposé par le composant *GestionnaireDeReunion*. Aussi, le composant *GestionnaireDeReunion* est lié à travers un lien comportemental avec le composant *GestionnaireDAbsence* parce que la méthode *confirmer_reunion* définie dans le composant *GestionnaireDeReunion* fait appel à la méthode *est_absent* définie dans le composant *GestionnaireDAbsence*. Dans ce cas, les interfaces de communication vont permettre l'invocation de la méthode *est_absent* par la méthode *confirmer_reunion* dès lors qu'elle sera appelée (voir Figure 3.17) : lorsque la méthode *confirmer_reunion* du composant *GestionnaireDeReunion* est invoquée (1), son code correspondant est exécuté (2). Lors de cette exécution, une méthode (*est_absent*) implémentée par un autre composant (*GestionnaireDAbsence*) est invoquée. Cette invocation est réalisée au travers des interfaces de communication (3). La méthode *est_absent* pourra alors être exécutée (4). Le diagramme de séquences correspondant à la réalisation des interfaces de communication est fourni dans la figure 3.18.

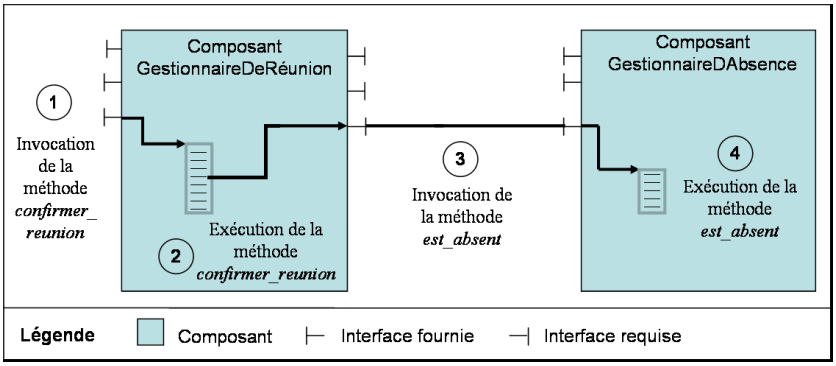

Figure 3.17 – Exemple d'interfaces de communication entre deux composants générés

3.3.2.2 Mise en œuvre des interfaces de communication

Comme nous l'avons évoquée précédemment, la mise en œuvre des interfaces de communication passe par la transformation d'appels de méthodes en appels de services fournis par un sous-composant. Cette transformation est réalisée par analyse et instrumentation du code orienté objet des composants générés.

En fait, le corps de chacune des méthodes contenues dans la classe d'implémentation d'un composant est analysé de manière à repérer les méthodes qui sont appelées et à déterminer par quels composants elles sont implémentées. De ce fait, si lors de l'analyse d'une méthode M_1, elle fait appel à la méthode M_2, trois cas sont possibles :

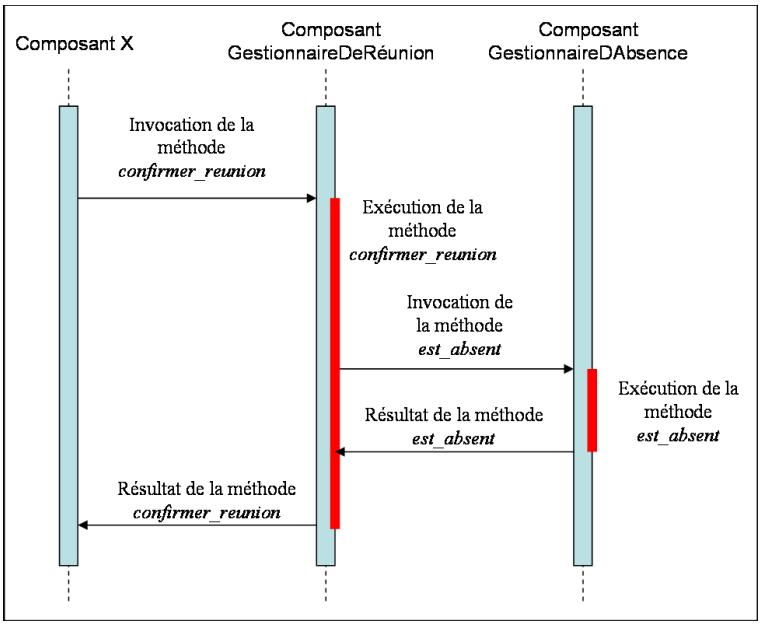

Figure 3.18 – Diagramme de séquences montrant la communication entre deux sous-composants générés

Cas 1 : la signature de cette méthode apparaît dans une interface fournie par le composant.
Dans ce cas, la dépendance fonctionnelle ne nécessite aucun traitement que ce soit au niveau architectural ou au niveau implémentatoire.

Cas 2 : la signature de cette méthode apparaît dans une interface fournie par un autre composant.
Dans ce cas, l'interface contenant la signature de la méthode M_2 doit être considérée comme un service requis par le composant contenant la méthode M_1. Dans ce cas, la description d'architecture de l'application ainsi que le code source du composant adapté doivent être modifiés. Du point de vue du code source, l'appel de la méthode M_2 dans M_1 doit être transformé en appel du service M_2 dans M_1.

Exemple de l'agenda-partagé
Par exemple, si l'on reconsidère le composant *Agenda-partagé* : le code ci-dessous correspond à l'implémentation d'un nouveau composant généré à savoir le composant *GestionnaireDeReunion*.

```
1  ...
2  public class GestionaireDeReunionImpl implements Reunion, ... , BindingController {
3  ...
4  public boolean confirmer_reunion(Date jour,Vector personne){
5  ...
6  if (est_en_reunion(service_appelle,(String)personne.elementAt(i), jour) ||
7      est_absent((String)personne.elementAt(i), jour)) {...}
8  ...
9  }
10 }
```

Comme nous pouvons le constater dans le code fourni ci-dessus, la méthode *confirmer_reunion* fait appel à une méthode (*est_absent*) associée à un service fourni par un autre composant généré

(ligne 7). De ce fait, l'appel de la méthode *est_absent* doit être transformé en l'appel du service *est_absent*. Pour cela, la classe doit définir un nouvel attribut correspondant à l'interface requise par le composant à savoir l'interface *Absence* (ligne 12). Puis, les appels de méthodes doivent être transformés en appel de service (ligne 7). Enfin, les interfaces de contrôle de liaisons (*BindingController*) doivent être implémentées ou instrumentées (si elles avaient déjà été créées) (lignes 14-41).

```
1  ...
2  public class GestionnaireDeReunionImpl implements Reunion, ... , BindingController {
3  ...
4  public boolean confirmer_reunion(Date jour,Vector personne){
5  ...
6  if (est_en_reunion(service_appelle,(String)personne.elementAt(i), jour) ||
7      absence.est_absent((String)personne.elementAt(i), jour)) {...}
8  ... }
9
10 //Interface requise
11 private Absence absence;
12
13 //Implémentation de l'interface de contrôle BindingController
14 public String[] listFc () {
15 String[] l_key=(String[])notify.keySet().toArray(new String[notify.size()]);
16 String[] l_inter=new String[] { "It_absence", ... };
17
18 int taille=l_key.length+l_inter.length;
19 String[] liste=new String[taille];
20 for (int i=0;i<liste.length;i++){
21  for (int j=0;j<l_inter.length;j++) {liste[i]=l_inter[j];}
22  for (int j=0;j<l_key.length;j++) {liste[i]=l_key[j];}
23 }
24 return liste; }
25
26 public Object lookupFc (final String cItf) {
27 if (cItf.equals("It_absence")) {return absence;}
28 ...
29  return null;}
30
31 public void bindFc (final String cItf, final Object sItf) {
32 if (cItf.equals("It_absence")) {absence = (Absence)sItf;}
33 ... }
34
35 public void unbindFc (final String cItf) {
36 if (cItf.equals("It_absence")) {absence = null;}
37 ... }
38 }
```

Ces transformations ont également des impacts sur le niveau architectural du composant en question. En effet, la détection d'une liaison entre deux composants doit être répercutée sur la description de l'architecture de l'application : un composant dont au moins une des méthodes définies dans son implémentation utilise une autre méthode définie par un autre composant doit définir une interface requise correspondant à celle définissant la méthode utilisée. Ici, l'interface contenant la signature de la méthode M_2 doit être définie comme requise par le composant implémentant la méthode M_1.

Exemple de l'agenda-partagé

Si nous reprenons l'exemple de l'agenda partagé, le composant *GestionnaireDeReunion* doit définir l'interface *Absence* en temps qu'interface requise (voir Figure 3.19). La nouvelle description de l'architecture de l'application relative au composant *GestionnaireDeReunion* sera alors :

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/standard.dtd">
4  ...
5  <component name="GestionnaireDeReunion">
6  <interface name="It_reunion" role="server" signature="Reunion"/>
7  <interface name="It_absence" role="client" signature="Absence"/>
8  ...
9  <interface name="It_Mail_sender" role="client" signature="Mail_sender"/>
10 <content class="GestionnaireDeReunionImpl"/>
11 </component>
12
13 <component name="GestionnaireDAbsence">
14 <interface name="It_absence" role="server" signature="Absence"/>
15 ...
16 <content class="GestionnaireDAbsenceImpl"/>
17 </component>
18
19 <binding client="GestionnaireDeReunion.It_absence" server="GestionnaireDAbsence.It_absence"/>
20 ...
```

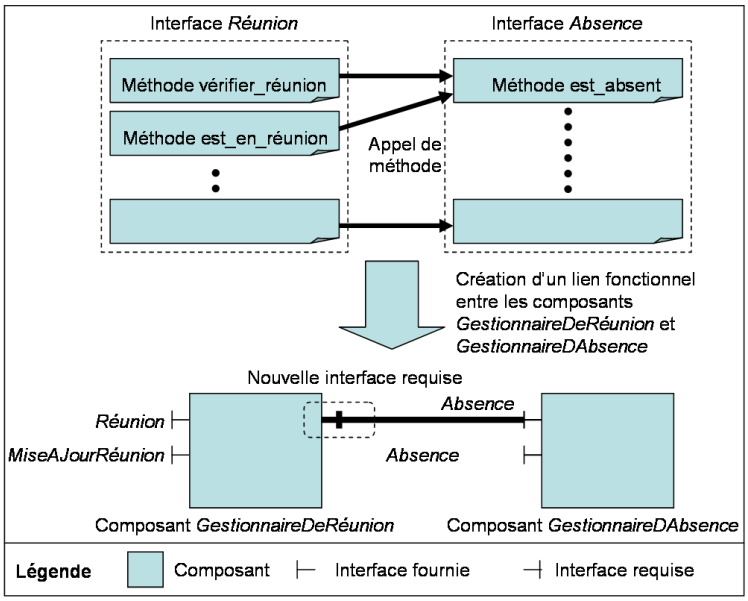

Figure 3.19 – Exemple d'interfaces de communication pour gérer la communication entre les composants générés

Le processus est identique lorsque le service requis est fourni par un autre composant de l'application.

Cas 3 : la signature n'apparaît dans aucune interface.

Si la méthode appelée n'apparaît dans aucune interface, il s'agit d'une méthode interne au composant à adapter. La gestion des dépendances fonctionnelles relative à ces méthodes dépend de la stratégie utilisée pour gérer l'intégrité structurelle. Si la solution optée consiste à dupliquer le code de ces méthodes, aucun traitement n'est nécessaire car l'utilisation de ces méthodes coïncide

avec un appel de méthode classique. Si la solution optée est celle de la centralisation des méthodes internes, il est nécessaire d'appliquer les mêmes traitements que dans le cadre du cas 2.

3.3.3 Gestion de la cohérence des états des composants : les interfaces de notification

La gestion de la cohérence des états des composants générés lors de la fragmentation est réalisée par l'intermédiaire d'interfaces dédiées appelées interfaces de notification.

3.3.3.1 Présentation des interfaces de notification

Une ressource partagée peut être manipulée par l'ensemble des composants dans lesquels elle est définie. Il est donc nécessaire de garantir la cohérence de l'état de cette ressource lors d'un accès multiple. La communication entre les différents composants définissant une ressource partagée doit être réalisée de manière à pouvoir préserver des états cohérents de cette ressource. Cette tâche est accomplie au travers des interfaces de notification.

Ainsi, les interfaces de notification d'un composant sont chargées, d'une part, de notifier au reste des composants partageant avec lui une ressource logicielle le changement d'état de cette dernière et, d'autre part, de mettre à jour l'état d'une ressource partagée après qu'elle ait été mise à jour par un autre composant (*i.e.* prise en compte des notifications de modification d'une ressource partagée) ;

Exemple de l'agenda-partagé
La figure 3.20 montre un exemple d'interfaces de notification utilisées pour la gestion de la ressource *nb_jours_libres* qui est partagée par les composants *GestionnaireDAbsence* et *GestionnaireDAgenda*. Quand la ressource *nb_jours_libres* est mise à jour par le composant *GestionnaireDAbsence* (1), une notification est envoyée au composant *GestionnaireDAgenda* (2) afin que la nouvelle valeur puisse être prise en compte par ce composant (3). Le diagramme de séquences correspondant à la réalisation des interfaces de notification est fourni dans la figure 3.21.

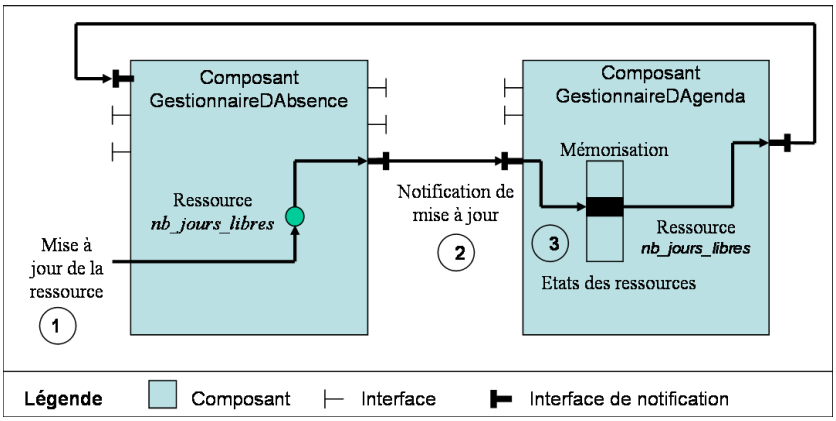

Figure 3.20 – Exemple d'interfaces de notification des états des ressources partagées entre plusieurs composants générés

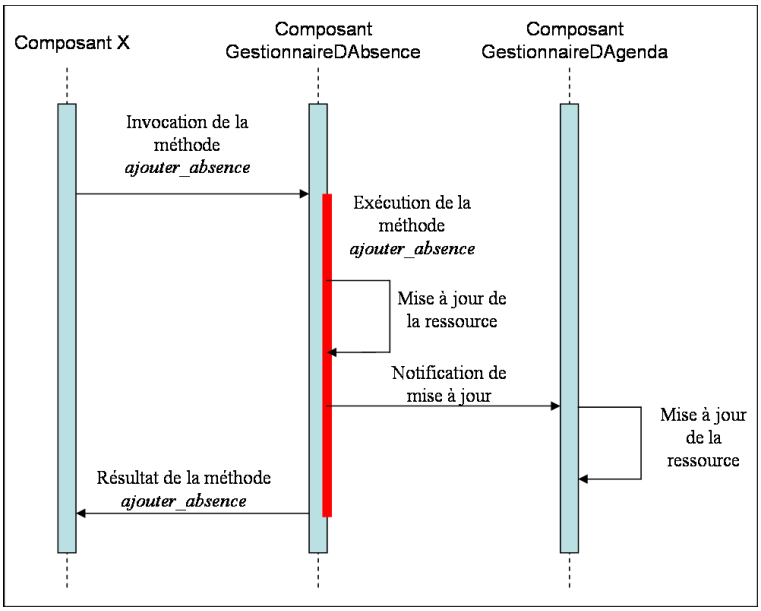

Figure 3.21 – Diagramme de séquences représentant la notification des états des ressources partagées entre plusieurs composants générés

3.3.3.2 Mise en œuvre des interfaces de notification

Les interfaces de notification doivent être insérées au niveau de la description de l'architecture de l'application puis ces modifications doivent être répercutées sur le code source du composant, de par, l'analyse et la génération de code.

Mise en œuvre des interfaces de notification au niveau architectural Au niveau architectural, la mise en œuvre des interfaces de notification se traduit par la définition de deux nouvelles interfaces pour chaque composant utilisant une ressource partagée :

- la première interface permet au composant en question de notifier au reste des composants partageant avec lui une ressource le changement d'état de cette dernière. Cette interface est définie comme étant requise, synchrone et de cardinalité collection. Ainsi, le composant ayant mis à jour l'état de la ressource ne peut continuer son exécution qu'à partir du moment où les autres composants partageant cette ressource aient pris effectivement ce changement d'état en compte ;

- la deuxième interface, définie comme fournie, permet de mettre à jour l'état d'une ressource partagée après qu'elle ait été mise à jour par un autre composant (*i.e.* prise en compte des notifications de modification d'une ressource partagée).

La description de l'architecture de l'application doit alors être mise à jour de par l'intégration de ces interfaces de notification à chaque composant partageant une ressource logicielle.

Exemple de l'agenda-partagé

Reprenons l'exemple de la gestion de la ressource *nb_jours_libres* qui est partagée par les composants *GestionnaireDAbsence* et *GestionnaireDAgenda*, présenté dans la figure 3.20. La nouvelle description d'architecture de l'application relative au composant *Agenda-partagé* incluant les interfaces de notification est donc la suivante :

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/standard.dtd">
4
5    <component name="GestionnaireDAgenda">
6      <interface name="It_agenda" role="server" signature="Agenda"/>
7      <interface name="It_miseajouragenda" role="server" signature="MiseAjourAgenda"/>
8        ...
9      <interface name="It_c_notify_GestionnaireDAgenda" role="client" signature="Notify"
10         cardinality="collection"/>
11     <interface name="It_s_notify_GestionnaireDAgenda" role="server" signature="Notify"/>
12   </component>
13
14   <component name="GestionnaireDAbsence">
15     <interface name="It_absence" role="server" signature="Absence"/>
16     <interface name="It_miseajourAbsence" role="server" signature="MiseAjourAbsence"/>
17     ...
18     <interface name="It_c_notify_GestionnaireDAbsence" role="client" signature="Notify"
19         cardinality="collection"/>
20     <interface name="It_s_notify_GestionnaireDAbsence" role="server" signature="Notify"/>
21   </component>
22
23 <binding client="GestionnaireDAbsence.It_c_notify_GestionnaireDAgenda"
24         server="GestionnaireDAgenda.It_s_notify_GestionnaireDAgenda"/>
25 <binding client="GestionnaireDAgenda.It_c_notify_GestionnaireDAbsence"
26         server="GestionnaireDAbsence.It_s_notify_GestionnaireDAbsence"/>
27 ...
```

Mise en œuvre des interfaces de notification au niveau implémentatoire Après la mise à jour d'une ressource logicielle par un composant généré, l'état de cette ressource doit être communiqué aux autres composants qui partagent, avec lui, cette ressource. Pour cela, il est nécessaire, d'une part, de déterminer comment et où sont réalisées les mises à jour de ressources partagées, puis d'instrumenter le code source des composants concernés afin qu'ils puissent notifier une modification d'état et prendre en compte des mises à jour réalisées par d'autres composants. Ces opérations doivent être réalisées différemment qu'il s'agisse de ressources primitives ou de ressources complexes (*i.e.* objets).

La première étape pour mettre en œuvre l'instrumentation du code source pour la notification consiste à déterminer comment une ressource peut être mise à jour. Cette opération peut être réalisée en utilisant une instruction de mise à jour mettant en jeu une référence directe ou indirecte de la ressource en question. Par exemple, dans le cadre de Java, où une ressource peut être un objet, celle-ci peut être mise à jour directement ou en utilisant une référence.

La deuxième étape consiste à repérer les instructions de mise à jour d'une ressource. En fait, les mises à jour de ressources primitives sont réalisées en utilisant des instructions d'affectation alors que les ressources qui sont des instances d'objets, sont mises à jour par des appels de méthodes sur cet objet. Ainsi, la gestion de la notification pour des ressources complexes doit être individualisée, liée à la manipulation des instances concernées et ne doit pas affecter les autres instances. Pour cela, nous introduisons dans la classe de manipulation de cette ressource, deux opérations qui doivent être insérées respectivement avant et après toute opération de manipulation de cette ressource. La première opération consiste à

sauvegarder l'état de la ressource et la deuxième à retourner les attributs de l'instance de la ressource qui ont subi un changement. L'opération de notification est réalisée dans le cas du changement d'au moins un attribut de l'instance de la ressource. Les deux opérations respectivement de sauvegarde et de retour seront ajoutées à l'ensemble des méthodes de la classe de définition de la ressource. L'instrumentation du code afin d'assurer la cohérence de ressources complexes se fait en trois temps :

1. dans un premier temps, il est nécessaire de générer le code permettant de sauvegarder l'état de la ressource. Pour cela, il faut déterminer l'ensemble des attributs primitifs de la ressource complexe et ce de manière récursive (*i.e.* une ressource complexe peut contenir d'autres ressources complexes), puis récupérer les valeurs de tous les attributs de la ressource en question et enfin sauvegarder dans des variables spécifiques la valeur de chaque attribut ;

2. ensuite, le code de l'opération de retour doit être généré. Ce qui permet de récupérer les valeurs de tous les attributs de la ressource en question et de les comparer avec les valeurs sauvegardées ;

3. enfin, il est nécessaire de générer le code de notification associé à la modification d'une ressource. En fait, si une des nouvelles valeurs est différente de la valeur qui a été sauvegardée, alors il faut envoyer une notification de la ressource à tous les composants partageant cette ressource afin qu'ils puissent prendre en compte ces modifications.

La dernière étape pour mettre en œuvre l'instrumentation consiste à déterminer à quel moment vont être envoyées les notifications de mise à jour d'une ressource. En fait, cette opération dépend du moment où les autres composants partageant une ressource pourraient prendre en compte une notification. Ainsi, étant donné que toutes les instructions de mise à jour d'une ressource sont rassemblées au sein d'une section critique (voir Section 3.2.3.1), bloquante pour les autres composants partageant cette ressource, il est inutile de notifier chaque mise à jour de ressource. En effet, seule la dernière notification d'une section critique est prise en compte par les autres composants. De ce fait, les notifications doivent être transmises juste avant la fin d'une section critique.

Exemple de l'agenda-partagé

Le code présenté ci-dessous montre l'instrumentation de la classe *GestionaireDAbsenceImpl* correspondant au composant *GestionaireDAbsence* réalisée afin d'intégrer les mécanismes de notification de la ressource *nb_jours_libres*.

```
1  ...
2  public class GestionaireDAbsenceImpl implements Absence, AbsenceAttribute, Notify, Update, ... {
3  ...
4  public boolean Ajouter_absence(Date jour_absent){
5  ...
6  nb_jours_libres--;
7  notify_values("nb_jours_libres",nb_jours_libres);
8  ...
9  }
10 ...
11 }
12
13 //************** Notification **************
14 private Map notify=new TreeMap(); //définition de l'interface de notification
15
16 public void notify_values(String resource, int value) //propagation de la notification
```

```
17 {
18 Iterator i=notify.values().iterator();
19 while (i.hasNext()) {((Notify)i.next()).modify_values(resource,value);}
20 }
21
22 public void modify_values(String resource, int value) //prise en compte de notifications
23 {
24 if (resource.equals("nb_jours_libres")) {nb_jours_libres=value;}
25 ...
26 }
27
28 //************** Connexions **************
29
30 public String[] listFc () {
31 String[] liste=new String[taille];
32 String[] l_key=(String[])notify.keySet().toArray(new String[notify.size()]);
33 ...
34 for (int j=0;j<l_key.length;j++)
35         {liste[i]=l_key[j];}
36
37         return liste;
38 }
39
40 public Object lookupFc (final String cItf) {
41 if (cItf.startsWith("notify_gestionairedagenda")) {return notify.get(cItf);}
42 ...
43 return null;
44 }
45
46 public void bindFc (final String cItf, final Object sItf) {
47 if (cItf.startsWith("notify_gestionairedagenda")) {notify.put(cItf,sItf);}
48 ...
49 }
50
51 public void unbindFc (final String cItf) {
52 if (cItf.startsWith("notify_gestionairedagenda")) {notify.remove(cItf);}
53 ...
54 }
55 ...
56 }
```

3.3.4 Gestion de la concurrence entre composants : les interfaces de synchronisation

La gestion de la concurrence entre composants générés lors de la fragmentation est réalisée par l'intermédiaire d'interfaces dédiées appelées interfaces de synchronisation.

Présentation des interfaces de synchronisation Les ressources partagées peuvent également être manipulées simultanément par l'ensemble des composants dans lesquels elles sont définies. Ainsi, il est nécessaire de garantir la cohérence de l'état des ressources lors d'un accès multiple et simultané. Pour cela, toutes les opérations de manipulation des ressources doivent être mutuellement exclusives. L'interdiction de l'accès multiple et simultané à une ressource est réalisée au travers des interfaces de synchronisation, définies dans chaque composant utilisant cette ressource.

Ainsi, les interfaces de synchronisation sont chargées de mettre en œuvre un système d'accès à une ressource partagée via des droits de lecture et d'écriture. En fait, lorsqu'un composant souhaite accéder à une ressource partagée, il doit demander aux autres composants qui partagent avec lui cette ressource, un droit d'accès qu'il devra céder dès qu'il en aura terminé avec elle.

Exemple de l'agenda-partagé
La figure 3.22 montre un exemple d'interfaces de synchronisation destinées à gérer l'accès à la ressource *nb_jours_libres* qui est partagée par les sous-composants suivants : *GestionnaireDAgenda*, *GestionnaireDeReunion* et *GestionnaireDAbsence*. La synchronisation de l'accès à cette ressource est gérée de la manière suivante : tout d'abord, le composant *GestionnaireDAbsence* qui a besoin de modifier la ressource *nb_jours_libres* (1) demande un droit d'accès aux autres composants qui partagent cette ressource (*GestionnaireDAgenda* et *GestionnaireDeReunion*) (2). Ensuite, dès que le composant *GestionnaireDAbsence* reçoit une notification provenant de tous les composants partageant la ressource, celui-ci peut alors procéder à la mise à jour de *nb_jours_libres* (3). Enfin, une fois que la mise à jour de la ressource est terminée, le composant *GestionnaireDAbsence* envoie une notification aux autres composants (*GestionnaireDeReunion* et *GestionnaireDAgenda*). Le diagramme de séquences correspondant à la réalisation des interfaces de synchronisation est fourni dans la figure 3.23.

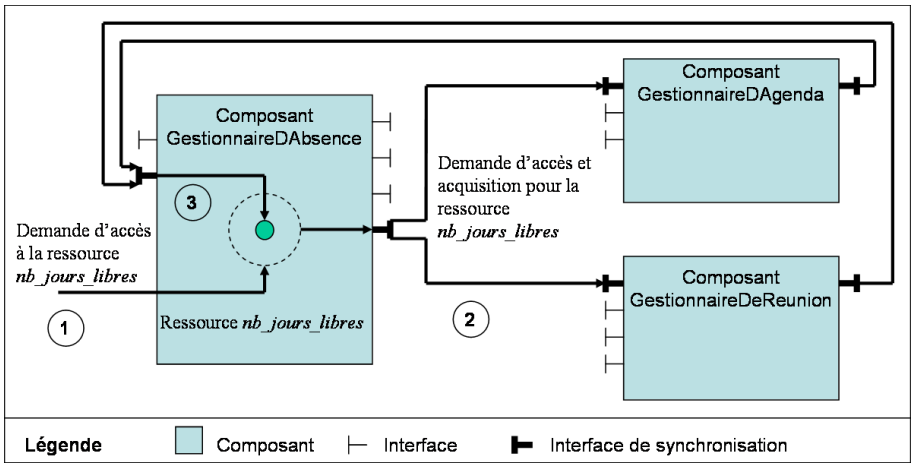

Figure 3.22 – Exemple d'interfaces de synchronisation pour l'accès à une ressource partagée

3.3.4.1 Mise en œuvre des interfaces de synchronisation

Les interfaces de synchronisation doivent être insérées au niveau de la description de l'architecture de l'application puis ces modifications doivent être répercutées sur le code source du composant, de par, l'analyse et la génération de code.

Mise en œuvre des interfaces de synchronisation au niveau architectural La mise en œuvre des interfaces de synchronisation au niveau architectural se traduit par la définition de deux nouvelles interfaces pour chaque composant utilisant une ressource logicielle partagée :

- la première interface définie comme requise et synchrone lui permet d'acquérir le droit d'accès à la ressource. Cette interface garantit qu'à un instant donné, un et un seul accès à la ressource est possible ;

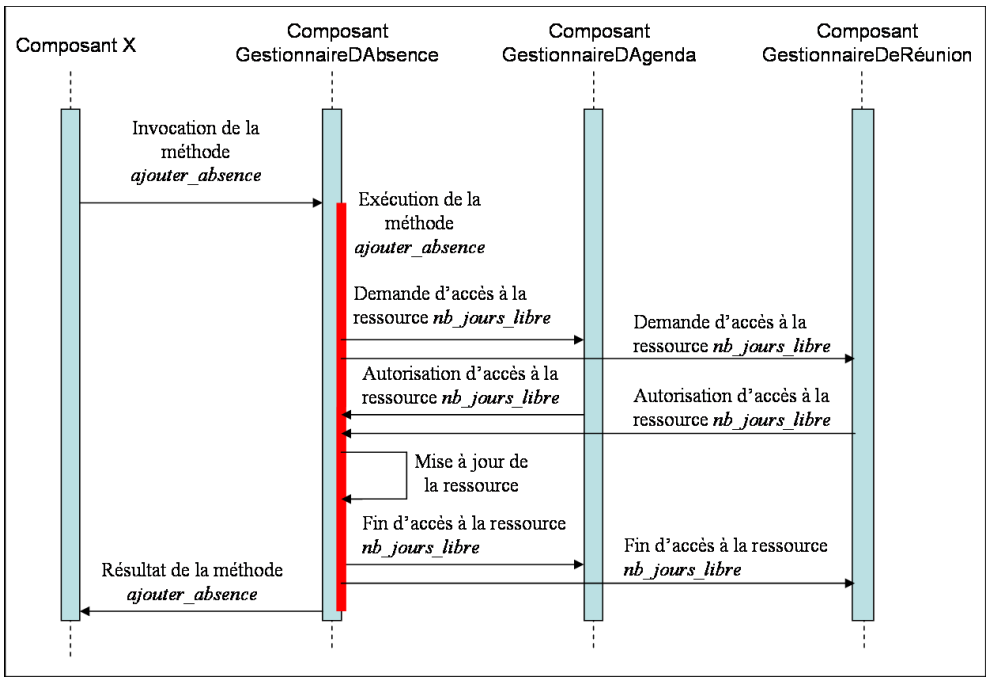

Figure 3.23 – Diagramme de séquences représentant la synchronisation pour l'accès à une ressource partagée

- la deuxième interface définie comme fournie et synchrone permet de libérer les droits de mise à jour sur la ressource partagée.

La description de l'architecture de l'application doit alors être mise à jour de par l'intégration des interfaces de notification.

Exemple de l'agenda-partagé
Reprenons l'exemple d'interfaces de synchronisation destinées à gérer l'accès à la ressource *nb_jours_libres* qui est partagée par les sous-composants suivants : *GestionnaireDAgenda*, *GestionnaireDeReunion* et *GestionnaireDAbsence*, présenté dans la figure 3.22. La nouvelle description d'architecture de l'application relative au composant *Agenda-partagé* incluant les interfaces de synchronisation est donc la suivante :

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
3    "classpath://org/objectweb/fractal/adl/xml/standard.dtd">
4
5    <component name="GestionnaireDAgenda">
6      <interface name="It_agenda" role="server" signature="Agenda"/>
7      <interface name="It_miseajouragenda" role="server" signature="MiseAjourAgenda"/>
8        ...
9      <interface name="It_c_synchronize_GestionnaireDAgenda" role="client"
10         signature="Synchronize" cardinality="collection"/>
11     <interface name="It_s_synchronize_GestionnaireDAgenda" role="server"
12         signature="Synchronize"/>
```

```
13    </component>
14
15    <component name="GestionnaireDAbsence">
16      <interface name="It_absence" role="server" signature="Absence"/>
17      <interface name="It_miseajourAbsence" role="server" signature="MiseAjourAbsence"/>
18      ...
19      <interface name="It_c_synchronize_GestionnaireDAbsence" role="client"
20           signature="Synchronize" cardinality="collection"/>
21      <interface name="It_s_synchronize_GestionnaireDAbsence" role="server"
22           signature="Synchronize"/>
23    </component>
24
25  <binding client="GestionnaireDAbsence.It_c_synchronize_GestionnaireDAgenda"
26           server="GestionnaireDAgenda.It_s_synchronize_GestionnaireDAgenda"/>
27  <binding client="GestionnaireDAgenda.It_c_synchronize_GestionnaireDAbsence"
28           server="GestionnaireDAbsence.It_s_synchronize_GestionnaireDAbsence"/>
29  ...
```

Mise en œuvre des interfaces de synchronisation au niveau implémentatoire La mise en œuvre des interfaces de synchronisation permet d'interdire tout accès simultané à une ressource. Elle repose sur le placement des instructions de mise à jour des ressources au sein de sections critiques implémentées en utilisant des sémaphores.

Mise en place de sections d'accès concurrents aux ressources partagées L'interdiction de l'accès multiple et simultané à une ressource est gérée au moyen de sections critiques placées au niveau des services. On définit comme étant une section critique d'un service la fraction de code contenant tous les accès en écriture ou en lecture à une ressource que ce soit directement, en utilisant l'affectation, ou indirectement, de par l'appel de services modifiant la ressource.

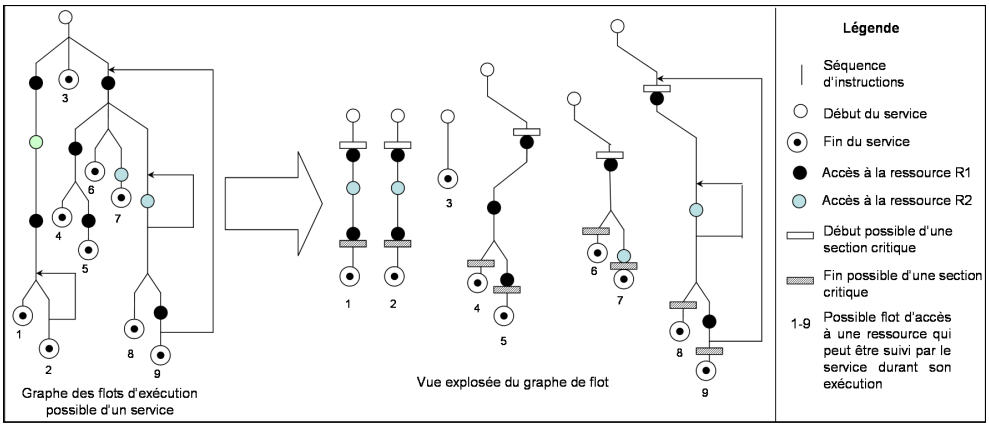

Figure 3.24 – Mise en place des sections critiques sur les services

Ces sections critiques se traduisent par la mise en place de verrous placés au sein même des services implémentés par le composant. Le positionnement des verrous est réalisé par analyse et instrumentation du code. Pour cela, il faut donc déterminer le premier et le dernier accès à une ressource au sein d'un service. Ces deux points constituent respectivement le point d'entrée et le point de sortie d'une section critique. La taille de cette dernière doit être réduite au minimum afin de favoriser le parallélisme. Pour cela, il faut, pour chaque service, déterminer tous les points d'entrée et de sortie possibles. La mise en

place de cette section ne peut être réalisée que dynamiquement (*i.e.* pendant l'exécution) pour être la plus fine possible. Ainsi, le positionnement d'une section critique au sein d'un service se fait en deux étapes :

1. *détection des points d'entrée en section critique*
   D'une part, il faut déterminer tous les points d'entrée possibles pour la section critique du service. Pour cela, il est nécessaire d'analyser le code source de chaque méthode tout en recherchant tous les premiers accès possibles à une ressource, que ce soit un accès direct (*i.e.* accès par le nom de la ressource) ou indirect (*i.e.* accès au moyen de méthodes). Ensuite, il faut instrumenter le code de manière à pouvoir poser dynamiquement un verrou juste avant le premier accès à une ressource. Pendant l'exécution, lors du passage sur un point d'entrée en section critique, il suffira de vérifier si le verrou a bien été posé. Si ce n'est pas le cas, alors le service devra entrer en section critique et donc poser un verrou (voir Figure 3.24) ;

2. *détection des points de sortie de section critique*
   La deuxième étape consiste à déterminer les points de sortie de la section critique. Ainsi, pour chaque service, il faut déterminer les points de modification d'une ressource (voir Section 3.2.3.1) et construire le graphe de flot d'instructions. Les sommets de ce graphe représentent les instructions qui peuvent être de deux types : les instructions standard ou les instructions modifiant une ressource. Les arcs représentent leur enchaînement. Il suffit alors de balayer ce graphe pour déterminer les points de sortie de la section critique. Le cas idéal serait de libérer le verrou juste après le dernier point d'accès à une ressource. Cependant, il faudrait anticiper le déroulement de l'application, ce qui est impossible. Notre algorithme pour placer les outils de libération du verrou est le suivant (voir Figure 3.25) : il faut parcourir tous les chemins du graphe et pour chaque sommet déterminer s'il existe au moins un chemin partant de ce sommet, arrivant à un point de sortie et passant par un point d'accès à une ressource. S'il n'en existe pas, alors ce point est considéré comme point de sortie possible de section critique, sinon il faut continuer le parcours du chemin. Lorsque tous les chemins ont été explorés, nous obtenons l'ensemble des points pour lesquels le service devra libérer le verrou.

```
Pour chaque chemin
    Pour chaque noeud N
        P = {chemin p / p part du noeud N,
                        le dernier élément du chemin est un point de sortie possible et
                        p contient un point d'accès à une ressource}
        Si P=∅ alors
            ajouter(N,Liste des points de sorties possibles de section critique) ;
        Fin Si
    Fin Pour
Fin Pour

ajouter(N,L) : méthode qui permet d'ajouter l'élément N dans la liste L.
```

Figure 3.25 – Algorithme de détection des sorties possibles de sections critiques

De plus, dans l'optique d'optimiser les sections critiques, nous pouvons réduire le nombre de ressources verrouillées (voir Figure 3.26). Quand le service démarre une section critique, toutes les ressources qui sont susceptibles d'être utilisées après ce point sont verrouillées. Puis, après chaque possibilité de branchement dans le graphe de flot du service, toutes les ressources qui n'ont pas été utilisées et

qui ne le seront pas peuvent être déverrouillées. A la fin de la section critique toutes les ressources qui n'ont pas été déverrouillées doivent l'être. Ainsi, ces opérations permettent de diminuer dans certains cas la période de verrouillage de certaines ressources.

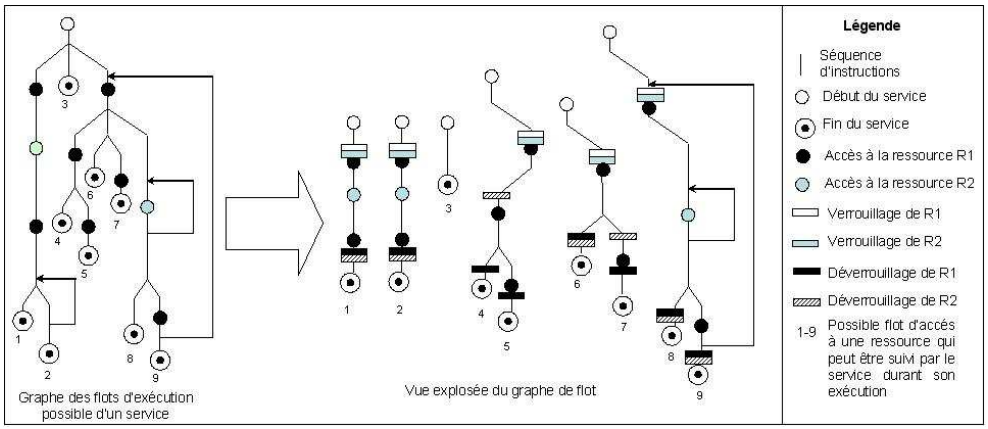

Figure 3.26 – Optimisation de la taille de la section critique

Procédure d'entrée aux sections d'accès L'entrée d'un service en section critique se fait par l'acquisition d'une autorisation matérialisée par un jeton. Deux approches sont possibles pour la gestion des jetons.

La première technique dite « centralisée » consiste à utiliser une pile située sur un composant pour déterminer le service qui possède le jeton. Ce dernier sera celui qui se trouve en haut de la pile. Chaque fois qu'un service demande un jeton, son identifiant sera placé en bas de la pile. Ainsi, lorsqu'un service devra entrer en section critique, il aura à vérifier si son identifiant se trouve en haut de la pile. La libération du verrou consiste à supprimer le premier élément de la pile.

La deuxième stratégie est basée sur le concept de la décentralisation. Elle nécessite l'utilisation d'une pile pour chaque composant et d'un séquenceur. La pile d'un composant doit contenir la liste des services ayant demandés le jeton. Le premier élément de pile contient l'identifiant du service qui possède le jeton et le dernier est l'identifiant du service qui a demandé en dernier le jeton. Lorsque le service qui possède le jeton sort de section critique, ce dernier doit être supprimé de la pile et le second possède alors le jeton. Un séquenceur qui se balade de composant en composant est utilisé pour ordonner les opérations de demande d'entrée en section critique de manière à assurer la cohérence de chaque pile.

Le processus d'obtention d'un jeton est le suivant (voir Figure 3.27) : (1) lorsqu'un service veut entrer en section critique, il demande au séquenceur un numéro. Ensuite, (2) il va enregistrer dans sa pile ces informations et (3) envoyer sa demande aux autres composants en fournissant son identifiant ainsi que le numéro donné par le séquenceur. (4/5/6) Lorsqu'un composant reçoit sa demande, (7/8/9) il va positionner l'identifiant du service demandant le jeton dans sa pile en fonction du numéro fourni et (10/11/12) renvoyer un accusé de réception. (13) Un composant possédera le jeton uniquement lorsqu'il sera en haut de sa pile et qu'il aura reçu un accusé de réception de la part de tous les autres composants.

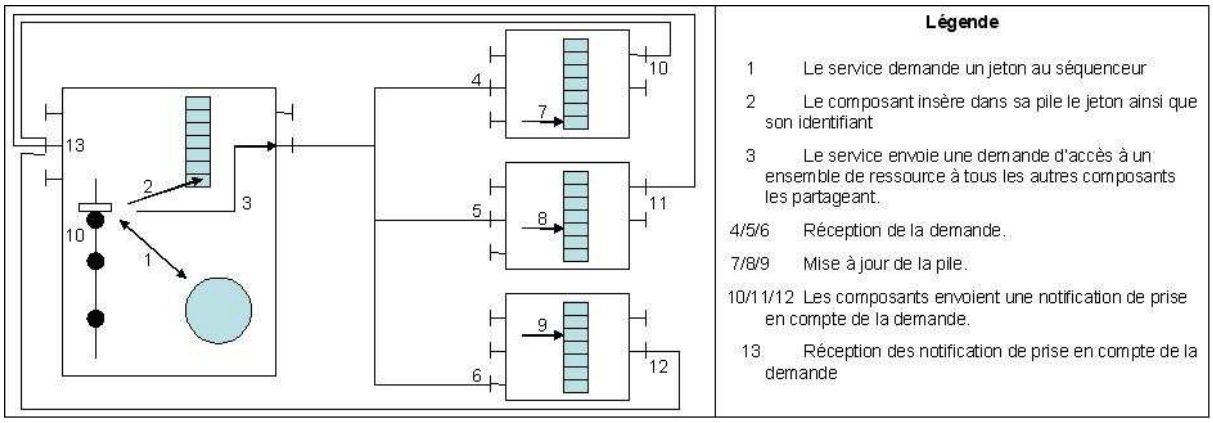

Figure 3.27 – Fonctionnement des jetons

Procédure de transition de droits d'entrées aux sections d'accès concurrents Etant donné que l'on applique la section critique au niveau des services, il faut prévoir le passage du jeton d'une méthode à une autre. En effet, un service peut faire appel à un autre service du composant ou même d'un autre composant. Ainsi, chaque méthode et service doivent connaître dans quel cadre ils sont appelés (*i.e.* par quel service). Pour réaliser cela, il faut que les méthodes puissent se transmettre l'identifiant du service qui les appelle afin qu'elles puissent vérifier qu'elles ont bien le bon jeton et qu'elles le libèrent correctement.

Par exemple, si on a deux composants C_1 et C_2 fournissant respectivement les services A et B, le service A du composant C_1 fait appel au service B du composant C_2, il faut que le service B sache qu'il est appelé dans le cadre du service A (voir Figure 3.28). En effet, si le service A verrouille les ressources avant d'appeler le service B, lorsque le service B s'exécute, il devra savoir qu'il possède le jeton (*i.e.* afin de libérer le verrou) car le service B est inclus dans le service A. Cette opération ne peut être réalisée que dynamiquement.

Il faut donc que chaque service puisse connaître son contexte d'exécution à savoir quel est le service instigateur de la requête : un service doit pouvoir déterminer s'il est appelé par un autre service ou bien s'il est appelé de manière indépendante (*i.e.* premier appel au service). Pour réaliser cette opération, il est nécessaire d'introduire un nouveau paramètre optionnel à chaque service qui va permettre de déterminer dans quel contexte est appelé ce service. Lorsque ce paramètre est indiqué, le service appelé sera dans le cadre d'un service en cours d'exécution (*i.e.* le paramètre contiendra le nom du service qui utilise ce service). Si ce paramètre n'est pas indiqué, il s'agit d'un service fourni par le composant en premier appel. Il faut également pouvoir identifier de manière unique les services de base pour éviter les appels concurrents à un service. Une autre contrainte qui s'impose, est le fait que l'on ne puisse pas modifier les interfaces du composant à adapter car celles-ci peuvent être utilisées par d'autres composants de l'application. Il faut donc créer de nouvelles interfaces contenant les services internes au composant. Ces dernières doivent permettre de transférer le jeton de service en service.

Nous pouvons noter que la stratégie de gestion de la synchronisation nous permet de préserver une cohérence forte sur toutes les ressources partagées. De plus, elle nous garantit l'absence d'interblocage. Cependant, elle est relativement coûteuse comme nous pourrons l'observer dans la section 3.5. Ainsi, il pourrait être envisagé d'appliquer différentes solutions de maintien de la cohérence en fonction des

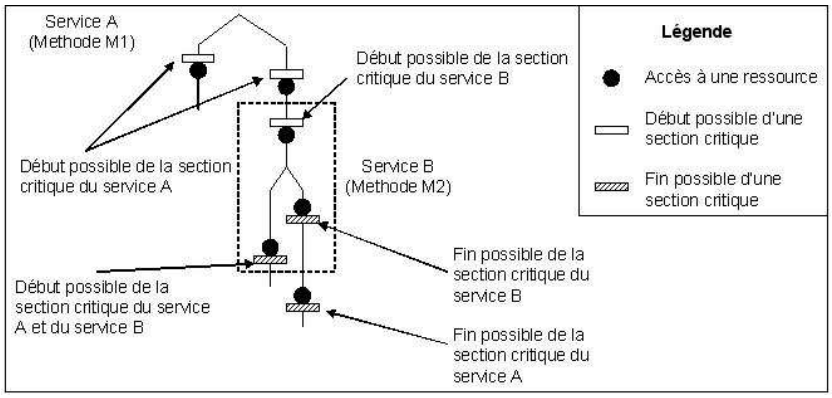

Figure 3.28 – Passage de jeton de services en services

ressources concernées (*i.e.* cohérence faible sur certaines ressources) qui permettraient d'en réduire son coût. Une telle gestion de la synchronisation doit être spécifique au composant adapté. Dans ce cas, elle ne peut être automatisée et donc nécessite l'intervention d'un développeur. L'étude des stratégies de gestions des ressources partagées n'étant pas un des objectifs prioritaires de cette thèse, nous invitons le lecteur à consulter [24, 109].

3.3.5 Gestion de la configuration du composant résultat des transformations

Afin de garantir l'interopérabilité du résultat de l'adaptation et la transparence de l'adaptation, l'assemblage des composants issus de la décomposition du composant adapté est encapsulé dans un composant composite. Cette encapsulation nécessite la mise en œuvre de mécanismes permettant au composite de communiquer avec ses sous-composants. Ces mécanismes passent par la définition d'interfaces dédiées appelées interfaces de communication entre le composite et les sous-composants générés. Par ailleurs, afin d'intégrer de nouveaux points de variabilité au composant issu de l'adaptation, nous avons proposé d'intégrer au composite résultat des mécanismes lui permettant de configurer certaines propriétés liées à la relation de composition. Pour permettre à un acteur externe de configurer ces propriétés, nous dotons le composite d'une interface de configuration.

Interface de communication entre le composite et les sous-composants générés Afin de contrôler les interactions entre le composite généré et ses sous-composants, nous introduisons dans le composite des interfaces de communication. Ces interfaces permettent, entre autre, de transférer les messages du composite vers les nouveaux composants générés (*i.e.* sous-composants) pour l'invocation de services fournis par l'un des sous-composants ou bien des sous-composants vers le composite pour l'invocation de services fournis par d'autres composants (autres que les sous-composants du composite résultat de l'adaptation). Ces interfaces peuvent également permettre d'insérer du code d'adaptation en vue d'une éventuelle adaptation fonctionnelle ; ce code s'insérant avant ou après les invocations de services.

Exemple de l'agenda-partagé
La figure 3.29 montre un exemple de communication entre le composant *Agenda-partagé* et l'un de ses sous-composants (*GestionnaireDeRéunion*) : lorsque le composant *Agenda-partagé* reçoit une invocation de l'un de ses services (1), il va transmettre cette invocation au sous-composant fournissant le

service concerné en l'occurrence le composant *GestionnaireDeRéunion* (2). Au cours de cette invocation, le message transmis peut éventuellement être transformé afin de procéder à une adaptation fonctionnelle. Ensuite, lors de l'exécution du service appelé (3), il se peut que le composant invoque un service fourni par un autre composant de l'application. Dans ce cas, le sous-composant *GestionnaireDeRéunion* devra passer par le composant *Agenda-partagé* pour réaliser cette invocation (ceci afin de préserver l'encapsulation des sous-composants générés). Ainsi, le composant *GestionnaireDeRéunion* devra invoquer une interface de communication du composite (4) pour accéder à des services fournis par d'autres composants de l'application. Lors de cette invocation, le message envoyé peut également être modifié en vue d'une adaptation fonctionnelle. Le diagramme de séquences correspondant à la réalisation de ces interfaces de communication entre le composite et ses sous-composants est fourni dans la figure 3.30.

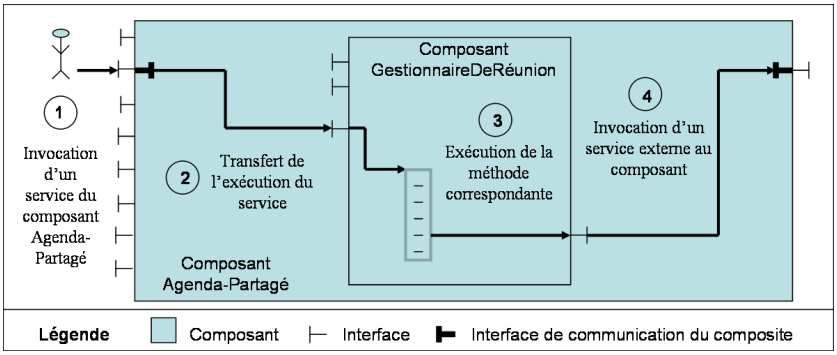

Figure 3.29 – Exemple d'interfaces de communication entre le composite et les sous-composants générés

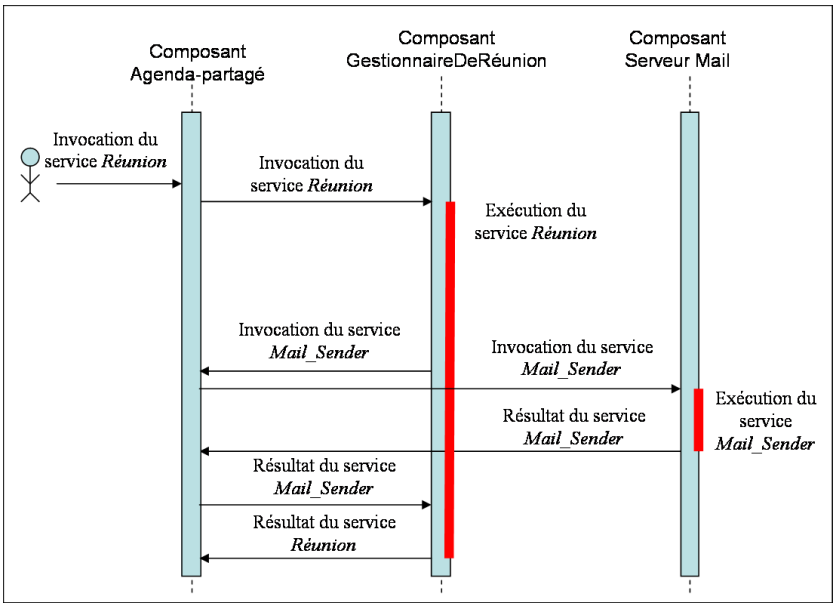

Figure 3.30 – Modèle dynamique de communication entre le composite et les sous-composants générés

Interface de configuration du composant composite Ces interfaces vont permettre de configurer certaines propriétés du composite qui sont liées à la relation de composition. La configuration de ces propriétés se fait en deux étapes : tout d'abord, un acteur externe de l'adaptation doit spécifier les valeurs de chaque propriété. Ensuite, la génération du composite sera réalisée en fonction des choix établis.

Spécification des propriétés configurables La configuration de ces propriétés paramétrables du composite issu de l'adaptation est réalisée au travers d'une spécification formulée dans un langage de type XML dont la DTD est fournie par la figure 3.31. Cette spécification doit être fournie par un acteur externe de l'adaptation au service d'adaptation structurelle.

```
1  <!ELEMENT component (interface*,component*,binding*,content?) >
2  <!— interface : list of component interfaces, component : list of sub−components,
3       binding   : list of connection between component provided interfaces and
4                       other component required interfaces
5       content   : implementation classes of the component —>
6
7  <!ATTLIST component
8    name CDATA #REQUIRED
9    definition CDATA #IMPLIED
10   cardinality CDATA #IMPLIED
11   inv_cardinality CDATA #IMPLIED
12   composition_lvl CDATA #IMPLIED
13   encapsulation type ("White−box" | "Black−box" | "Mix−box")
14   internal−access type ("Direct" | "Composite" )
15   service−propagation type ("CompositeToComponent" | "ComponentToComposite")
16 >
```

Figure 3.31 – Spécification des propriétés configurables du composite

Mise en œuvre des propriétés configurables Certaines propriétés, en l'occurrence celles liées à la structure du composite, à savoir la cardinalité (*i.e.* nombre de sous-composants pouvant appartenir à un objet composite *via* un lien de composition), la cardinalité inverse (*i.e.* nombre de composites qui peuvent se partager à un même sous-composant *via* un lien de composition) et le niveau de composition, peuvent être directement contrôlés au moment de l'étape de spécification du résultat de l'adaptation.

En ce qui concerne les propriétés liées au comportement du composite comme l'encapsulation, l'accessibilité interne ou le cycle de vie, la mise en œuvre de certaines configurations peut entraîner des traitements (*i.e.* analyse et instrumentation) au niveau des codes sources générés par le processus d'adaptation.

- *Le degré d'encapsulation des sous-composants*
  L'encapsulation des sous-composants fait référence à leur visibilité et à leur accessibilité vis-à-vis des autres composants de l'application. Cette propriété doit être mise en œuvre au cours de l'étape d'intégration du processus d'adaptation structurelle. Si les acteurs externes de l'adaptation ont paramétré le composite comme étant « boîte blanche », les sous-composants sont accessibles et manipulables directement. Cette configuration implique une faible considération de la notion d'encapsulation des composants. Ainsi, les autres composants de l'application (*i.e.* autres que ceux contenus dans le composite) peuvent accéder aux services issus de l'adaptation. Ce qui est en contradiction avec la première propriété sur le contrôle de l'intégration (*i.e.* propriété de transparence). Pour remédier à ce problème, il est nécessaire de mettre en place un système d'authenti-

fication des composants. Chaque fois qu'un composant fait appel à un service issu de l'adaptation, il doit s'identifier afin de poursuivre l'exécution de ce service. Chaque sous-composant doit donc connaître la liste des autres sous-composants du composite afin de pouvoir les identifier. Cette solution passe par l'instrumentation du code de ces services.
Le cas des composants mixtes est similaire à celui des « boîtes blanches ». En effet, étant donné que certains de ses sous-composants sont accessibles directement, la mise en place d'un système d'authentification sur les services issus de l'adaptation qui sont fournis par les composants visibles est nécessaire.
Concernant les composants « boîtes noires » et « boîtes grises », le composite fait l'office de filtre vis-à-vis des services fournis par les sous-composants (*i.e.* forte encapsulation). Les autres composants de l'application ne peuvent donc pas accéder aux services issus de l'adaptation. Dans ce cas, l'authentification n'est pas obligatoire.

- *L'accessibilité interne entre sous-composants*
  La propriété d'accessibilité interne permet de spécifier comment un sous-composant accède à un autre sous-composant du composite. Cette propriété peut être configurée de la manière suivante : l'accès peut être effectué *via* le composant composite ou bien *via* une référence directe vers le sous-composant en question.
  Le premier cas (*i.e.* accès au sous-composant par l'intermédiaire du composite) permet de préparer le composant à une possible adaptation fonctionnelle. En effet, comme toutes les invocations de services transitent par le composite, il devient alors facile d'analyser les interactions entre les sous-composants et de procéder à des pré ou des post-traitements. Le composite fournit ainsi les services d'un connecteur (*i.e.* centralisation des traitements). Dans ce cas, la mise en œuvre de cette propriété nécessite l'introduction de nouvelles interfaces au niveau du composite permettant de réaliser les connexions. En fait, les interfaces requises du sous-composant vont être connectées aux nouvelles interfaces fournies du composite qui vont permettre de déléguer les invocations de services au travers de liens d'exportation. Cependant afin de respecter la propriété de transparence, les nouvelles interfaces ainsi créées ne devront pas être accessibles aux autres composants de l'application. Ainsi, des systèmes d'identification de composants devront être mis en place afin de filtrer l'accès à ces services.
  Dans le deuxième cas (*i.e.* accès direct entre les sous-composants), le traitement des interactions pourra être réalisé au niveau des connecteurs entre les composants (*i.e.* décentralisation des traitements). Contrairement à la première approche, aucune interface additionnelle n'est requise.

- *Le cycle de vie du composite et de ses sous-composants*
  La gestion du cycle de vie est relative aux différentes stratégies envisageables lors de l'instanciation, de l'arrêt (*i.e.* dépendances comportementales) ou de la suppression (*i.e.* dépendances existentielles) du composite ou bien de l'un de ses sous-composants.
  La mise en œuvre des propriétés configurables liées au cycle de vie du composant composite et de ses sous-composants passe par la mise en place de connecteurs de coordination entre les différentes entités mises en jeu. Le rôle de ces connecteurs va être d'appliquer la stratégie choisie par un acteur externe de l'adaptation lors de la destruction (pour le cas des dépendances existentielles) ou de l'arrêt (pour le cas des dépendances comportementales) du composite ou de l'un de ses sous-composants. Par exemple, lorsqu'un sous-composant est détruit, le connecteur de coordination détecte cet évènement et le notifie aux autres sous-composants et au composite qui seront alors détruits ou non en fonction de la politique adoptée.

- *La propagation de services*
  Si la propriété de propagation de services est spécifiée comme étant du composite vers ses sous-composants, les services fournis par le composite doivent être fournis par ses sous-composants. Ainsi, de nouvelles interfaces correspondant aux services du composite doivent être introduites dans les sous-composants. Celles-ci vont permettre de déléguer les invocations de services d'un sous-composant vers le composite.
  Si la propriété de propagation de services est spécifiée comme étant des sous-composants vers le composite, les services fournis par les sous-composants doivent être fournis par le composite. Ainsi, de nouvelles interfaces correspondant aux services des sous-composants doivent être introduites dans le composite. Celles-ci vont permettre de déléguer les invocations de services du composite vers le sous-composant le fournissant.

3.4 Modèle étendu du composant support à l'adaptation structurelle pour la gestion de la distribution

Comme nous l'avons évoqué précédemment, la majorité des applications de l'adaptation structurelle prévoit un déploiement distribué du composant issu de la restructuration. Or, un composant conforme à notre processus de transformation structurelle ne peut être distribué de manière *ad-hoc*. De ce fait, nous proposons dans cette section un modèle étendu permettant de prendre en compte la distribution.

3.4.1 Analyse des besoins par rapport au modèle de la distribution

Un composant composite distribué est un composant dont les sous-composants peuvent être déployés sur différents sites. Le modèle de composants distribués que nous proposons doit répondre aux propriétés définies dans le cadre de notre processus (*i.e.* intégrité, cohérence, transparence, interopérabilité et autonomie). Par ailleurs, afin d'améliorer le résultat de la distribution en termes de maintenabilité, nous fixons de nouvelles propriétés :

- *visibilité de la structure interne*
  Pour des raisons de maintenabilité, la structure interne du composant structurellement adapté distribué doit rester visible. Ainsi, sur chaque site de déploiement du composant, sa structure interne en termes de composants et de connexions entre ces composants doit être identique ;

- *encapsulation des composants distribués*
  La distribution du composant issu de l'adaptation ne doit pas altérer les propriétés d'encapsulation du composite ; ceci pour des raisons de sécurité. Ainsi, les sous-composants déployés sur des sites différents de celui initialement prévu ne doivent pas être directement accessibles par les autres composants de l'application ;

- *transparence de la distribution*
  La mise en œuvre de la distribution doit être réalisée de manière transparente pour les sous-composants qui ont été générés lors de la fragmentation. Ainsi, les sous-composants doivent pouvoir s'envoyer des messages sans savoir qu'ils sont déployés sur des machines. De plus, l'implémentation des composants destinés à être déployés sur une machine ne doit pas être modifiée.

Nous distinguons alors trois modèles de distributions possibles pour le composant structurellement adapté obtenu suite à l'application de notre processus de transformation structurelle :

- *un modèle de distribution à faible niveau d'encapsulation*
  La première solution possible (voir Figure 3.32, cas B) consiste à déployer les sous-composants sur différents sites et le composite (dont l'instance ne contient aucun sous-composant) sur le site principal. Des interfaces de connexion sont alors utilisées pour transférer les messages du composite vers les sous-composants pouvant être déployés sur le site local ou bien sur un site distant (*i.e.* liens d'exportation). Les sous-composants sont alors connectés entre eux par l'intermédiaire de liaisons pouvant être locales ou distantes.
  Cette stratégie impose une accessibilité directe des sous-composants sur leur site de déploiement étant donné qu'ils ne sont plus encapsulés dans un composant composite ; ce qui est contraire aux principes de composants « boîtes noires » ou « boîtes grises ». Par ailleurs, la structure interne du composite reste floue car ce dernier ne contient que des connecteurs. Cette stratégie ne peut donc pas être envisageable ;

- *un modèle de distribution à base de proxys*
  La deuxième solution (voir Figure 3.32, cas C) consiste à utiliser des composants « *proxys* » (*i.e.* représentation du composant distant) au sein du composant composite tout en déployant les sous-composants de manière *ad-hoc* sur les différents sites. Les *proxys* créés sont utilisés afin d'accéder aux composants distants. Cette stratégie permet d'améliorer la visibilité de la structure du composant composite. Cependant, l'encapsulation des composants déployés sur des sites distants reste inexistante. De ce fait, cette stratégie ne peut pas être utilisée ;

- *un modèle de distribution par copies du composite*
  La dernière solution (voir Figure 3.32, cas A), que nous avons choisie, réside dans la création d'un composant composite sur chaque site d'implantation du composant. Cette solution permet de préserver une encapsulation forte des composants créés. Une instance du composant composite est chargée sur chaque site qui contient une partie de ce composant (*i.e.* au moins un sous-composant). Néanmoins, le composant composite entier n'est pas instancié sur chaque site. En fait, différentes copies du composant composite sont instanciées. Chaque instance se compose d'un ensemble de composants locaux fournissant les services proposés par le composite et d'un ensemble de composants virtuels utilisés comme composants de connexion entre les instances du composite déployées sur différents sites.

  Nous pouvons noter que les composants virtuels ne sont pas appelés composants « proxys » car leur rôle ne se limite pas à de l'invocation de services. En fait, les composants virtuels sont des réifications de connecteurs de communication. Au delà de leur rôle de base qui est la transmission des invocations de services à des composants distants, ils peuvent être utilisés notamment pour introduire du code d'adaptation en vue de réaliser une adaptation fonctionnelle du composant adapté (au travers de pré-conditions ou de post-conditions).

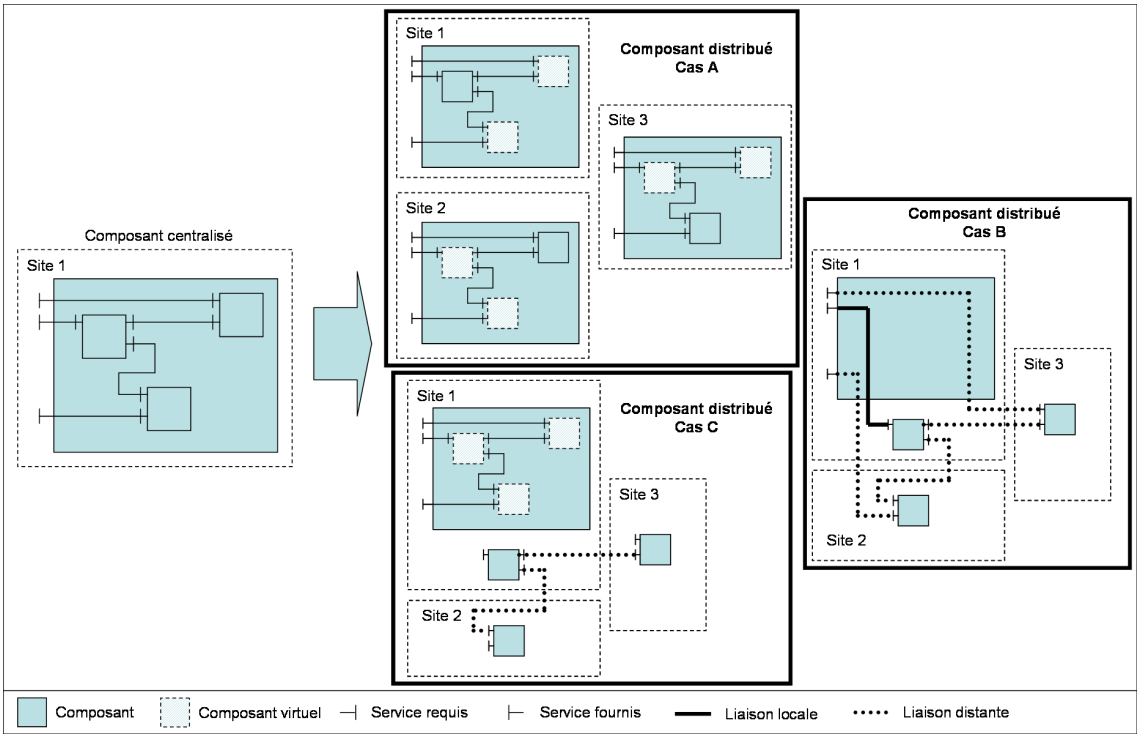

Figure 3.32 – Transformation d'un composant centralisé en un composant distribué

3.4.2 Modèle de gestion de la distribution

Le modèle de distribution de composants composites que nous proposons est constitué de deux parties (voir Figure 3.33). La première partie est consacrée à la gestion de la distribution au niveau du contenu du composant composite et la seconde, au contrôle de la distribution. Concernant la partie « contenu », un composant composite distribué est constitué de plusieurs copies différentes de ce composant (voir Figure 3.35). En fait, une copie du composite doit être instanciée sur chaque site d'implantation en fonction de la spécification de la distribution. Chacune de ces copies doit contenir un ensemble de composants locaux (*i.e.* composants déployés sur le site en question) ainsi qu'un ensemble de composants virtuels (*i.e.* composant de connexion). En ce qui concerne la partie « contrôle », celle-ci est constituée d'un composant de distribution permettant d'assurer les communications entre les différentes copies de ce composant composite distribué. Ce composant est constitué de deux sous-composants qui sont : un composant de nommage et un composant de transfert.

3.4.2.1 Composants virtuels

Pour réaliser les communications distantes entre les composants locaux des différentes copies du composite issu de l'adaptation, et déployées sur différents sites, il est nécessaire d'introduire de nouveaux mécanismes. Ces mécanismes sont mis en œuvre par un nouveau type de composants appelés « composants virtuels ».

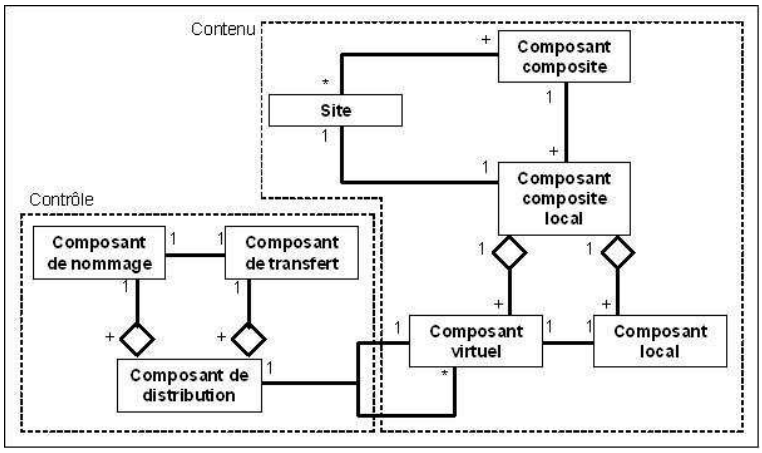

Figure 3.33 – Modèle de composants composites distribués

Rôle et comportement des composants virtuels Un composant virtuel est utilisé comme un connecteur entre un composant local et un composant distant. En effet, un service d'un composant local peut appeler un service distant comme si celui-ci était fourni par un composant local (*i.e.* le code fonctionnel des composants locaux n'est pas modifié). Le composant virtuel fournit les mêmes services que le composant distant. Néanmoins, les services fonctionnels ne sont pas exécutés sur ce composant. Les composants virtuels sont utilisés afin de transférer des messages entre les composants locaux et distants (*i.e.* services de délégation). Ainsi, les connexions distantes sont réalisées seulement d'un composant virtuel vers un autre composant virtuel car seulement ces composants ont la capacité d'envoyer et de recevoir des messages dans une infrastructure distribuée. Imaginons, par exemple, que lorsqu'un service du composant *GestionnaireDeReunion* appelle un service qui est fourni par un composant distant *GestionnaireDAgenda*, le composant *GestionnaireDeReunion* envoie un message au composant virtuel de *GestionnaireDAgenda*. Puis, cet appel est transformé en appel du composant virtuel de *GestionnaireDeReunion* au composant *GestionnaireDAgenda*. Cette transformation est réalisée à travers une connexion distante entre le composant virtuel de *GestionnaireDAgenda* et le composant virtuel de *GestionnaireDeReunion* (*i.e.* sur le site distant).

Mise en œuvre et implémentation des composants virtuels Un composant virtuel fournit les mêmes interfaces que ceux du composant distant cependant son implémentation (*i.e.* code des services) est différente. En fait, le code fonctionnel est remplacé par du code de contrôle permettant d'invoquer des services distants du composant distribué. Deux interfaces sont ajoutées à ce composant virtuel : l'une est requise et permet au composant d'envoyer des messages au composant distant et l'autre est fournie et permet au composant de recevoir des messages du composant distant. Ces deux interfaces assurent les communications distantes. Les connexions entre les composants virtuels sont générées par l'analyse de la description de l'architecture de l'application.

Exemple de l'agenda-partagé

Par exemple, quand un composant local *GestionnaireDAgenda* déployé sur le site 1 est lié à un composant distant *GestionnaireDeReunion* déployé sur le site 2 (*i.e.* une interface requise du composant *Gestion-*

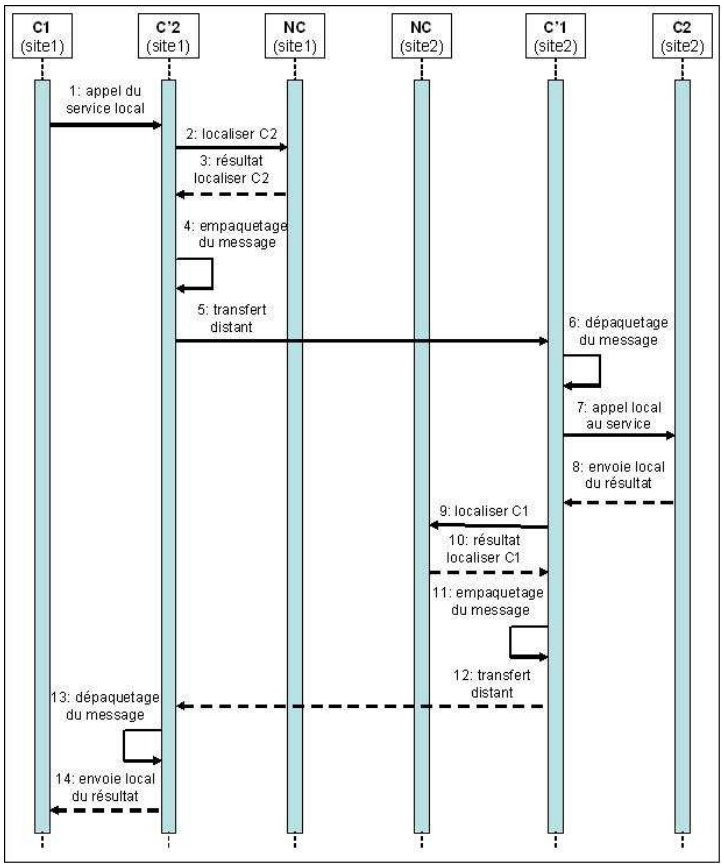

Figure 3.34 – Diagramme de séquences du processus de distribution entre deux composants

naireDAgenda est liée à une interface fournie du composant *GestionnaireDeReunion*), deux composants permettant d'assurer la connexion sont générés : l'un relie l'interface fournie du composant *VirtualGestionnaireDeReunion* (*i.e.* composant virtuel du *GestionnaireDeRunion* sur le site 1) à l'interface requise du composant *VirtualGestionnaireDAgenda* (*i.e.* composant virtuel de *GestionnaireDAgenda* sur le site 2) et l'autre, l'interface fournie du composant *VirtualGestionnaireDAgenda* à l'interface requise de *VirtualGestionnaireDeReunion*. Les communications entre les composants *VirtualGestionnaireDAgenda* et *VirtualGestionnaireDeReunion* sont réalisées par l'intermédiaire de ces deux nouvelles interfaces.

Cette stratégie d'utilisation de composants virtuels pour gérer les appels distants de services fournis par des composants déployés sur des machines distantes facilite la mise en œuvre d'une éventuelle adaptation fonctionnelle (*i.e.* adaptation du comportement du composant). En effet, du code d'adaptation peut être inséré avant ou après les invocations des services distants. Ainsi, les composants virtuels peuvent être considérés comme une réification en composants des connecteurs de communication (*i.e.* gestion des communications entre les composants locaux et distants) et des connecteurs d'adaptation.

3.4.2.2 Composants de distribution

Pour que ces composants virtuels puissent communiquer entre les différentes copies du composant, ils doivent utiliser des mécanismes de communication réseau de bas niveau : ils doivent connaître les adresses réseaux de chaque site de déploiement des différentes copies et pouvoir accéder à des protocoles de communication réseau (TCP/IP, UDP, etc.) pour le routage des messages échangés.

Ces services sont introduits au travers d'un composant non fonctionnel dédié, déployé sur chaque site et connecté aux composants virtuels qui y sont présents. Ce composant de contrôle, appelé composant de distribution est donc chargé d'assurer des communications distantes entre les composants virtuels. Il est composé de deux sous-composants :

- un composant de transport : il permet aux composants virtuels de mettre en œuvre les communications distantes. En effet, les services fournis par le composant du transport permettent d'empaqueter et déballer les messages qui sont échangés entre les composants locaux et distants, d'initialiser les connexions à travers les protocoles réseaux disponibles, etc. ;

- un composant de nommage : il permet au composant de transport de trouver l'adresse du site sur lequel les services composants locaux sont instanciés. En effet, les services fournis par le composant de nommage permettent de rechercher et de localiser les composants distants.

Comme nous avons expliqué précédemment, différentes instances de composant sont chargées sur des sites de déploiement de l'application. Comme une copie du composant composite est créée sur chaque site, les services non fonctionnels (*i.e.* services permettant de contrôler le contenu du composant, services permettant de contrôler les connexions entre les composants, services permettant de contrôler le cycle de vie du composant, etc.) sont dupliqués. Ainsi, nous devons assurer la communication et la cohérence entre les instances des composants au niveau de la couche de contrôle afin de préserver l'intégrité du composant logiciel. Par exemple, quand le composant composite est activé (*i.e.* appel aux services du contrôleur du cycle de vie), les autres instances qui sont chargées sur les sites distants doivent activer leur propre copie du composite. Cette opération peut être réalisée en utilisant l'instrumentation du code des services de contrôle.

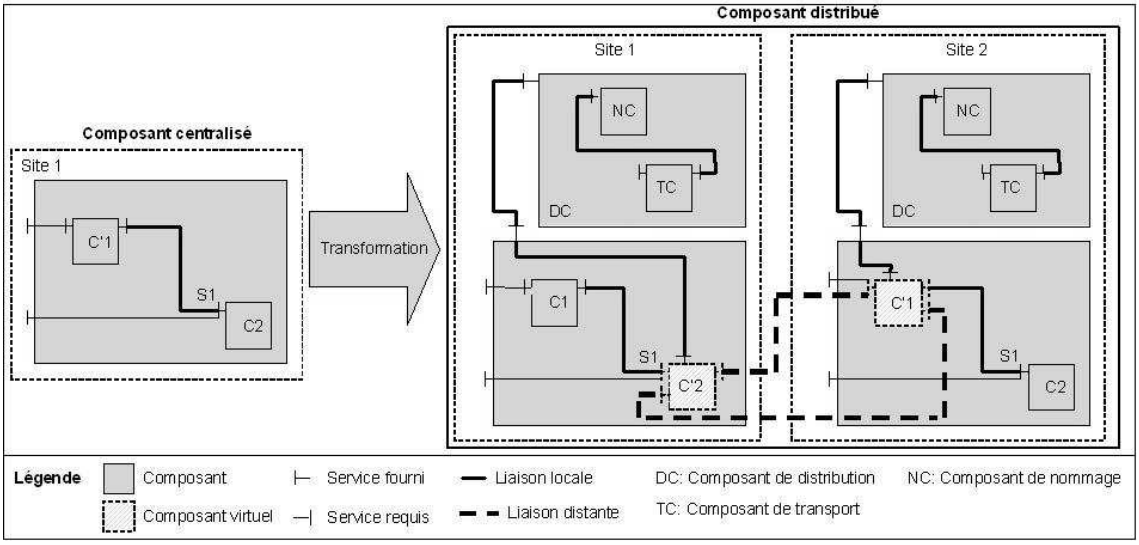

Figure 3.35 – Transformation d'un composant centralisé en un composant distribué

3.5 Étude et analyse des performances du modèle proposé

Comme nous l'avons mentionné précédemment dans nos motivations, notre approche a pour principal objectif de permettre le déploiement de composants logiciels dans des environnements dans lesquels ils n'auraient pas pu être déployés, du fait des ressources trop limitées. Cependant, la mise en œuvre de notre processus entraîne un surcoût, notamment lié à la gestion de la communication et de la synchronisation entre les différents sous-composants issus de l'adaptation bien que l'apport lié à la parallélisation du composant ne soit pas négligeable. En effet, le surcoût lié à l'adaptation structurelle du composant (C_{global}) correspond à la somme des coûts engendrés par l'assemblage des nouveaux composants générés moins le gain dû à la parallélisation du code (G_{paral}) (3.1). En fait, durant l'assemblage du nouveau composant, résultat de l'adaptation, nous avons introduit de nouvelles opérations nous permettant de garantir le comportement du composant généré : ces opérations sont dues à la matérialisation des dépendances comportementales (C_{comp}) et aux opérations de notification (C_{notif}) ainsi que de synchronisation (C_{synch}).

$$C_{global} = C_{comp} + C_{notif} + C_{synch} - G_{paral} \tag{3.1}$$

Le surcoût lié à la mise en place des dépendances comportementales (3.2) est essentiellement dû à la transformation d'un appel de méthode au sein d'une classe d'implémentation à un appel de service fourni par un autre composant qui a été généré lors de la décomposition du composant initial ainsi que du coût de transfert de l'invocation du service si ce dernier est déployé sur un composant distant.

$$C_{comp} = m * a * \Delta \qquad tel \qquad que \qquad \Delta = C_{S/C} - C_{M/O} + C_{TS} \tag{3.2}$$

m : nombre de services fournis par le composant initial

a : nombre d'appels de service

$C_{S/C}$: coût d'un appel d'un service

$C_{M/O}$: coût d'un appel d'une méthode

C_{TS} : coût de transfert d'un message vers un composant distant

Ce surcoût relatif à la mise en œuvre des dépendances comportementales est donc fortement lié au modèle utilisé pour implémenter les composants logiciels qui doivent être adaptés. En effet, le coût de l'appel d'un service fourni par un composant est différent suivant le modèle de composants utilisé et sa plate-forme d'implémentation. Par exemple, si l'on considère le modèle de composants Fractal et sa plate-forme Julia, le surcoût lié à la transformation d'un appel de méthode en appel de service est négligeable car de nombreux mécanismes de raccourcis sont mis en œuvre. En effet, nous avons constaté que le rapport entre la durée d'invocation d'une méthode et celle d'un service déployé sur une même machine est en moyenne de $3.42 \cdot 10^{-4}$. Concernant le coût de transfert d'un message vers un composant distant, ce dernier dépend des caractéristiques techniques de l'infrastructure distribuée (bande passante, etc.) ainsi que de son taux d'utilisation. Ce coût n'est pas négligeable mais est nécessaire dans la mise en place de la distribution du composant.

A cause du partage de ressources entre plusieurs composants, nous avons introduit des opérations de notification (3.3) et de synchronisation (3.4) afin de préserver la cohérence du composant. La notification permet à un composant d'informer les autres composants partageant une même ressource de la mise à jour de celle-ci, en envoyant un message à la fin de chaque section critique. Le coût de ces échanges dépend du type de réseau utilisé et de la bande passante disponible. Les opérations de synchronisation permettent de gérer l'accès multiple et simultané à une ressource par la création des sections critiques qui donnent l'exclusivité des droits d'accès à un service pendant une certaine durée. L'objectif est donc de diminuer au minimum la durée de ces sections critiques afin d'augmenter le gain lié au parallélisme.

$$C_{notif} = n * m * r * c * C_{R/C} \qquad tel \qquad que \qquad C_{R/C} = n * 2 * C_{S/C} + C_{R/M} \tag{3.3}$$

$$C_{sync} = n * m * c * C_{Sc/C} \qquad tel \qquad que \qquad C_{Sc/C} = n * 4 * C_{S/C} + C_{Asc/M} \tag{3.4}$$

n : nombre de composants générés

m : nombre de services fournis par le composant initial

r : nombre de ressources partagées

c : nombre de sections critiques dans un service

$C_{R/C}$: coût de notification de mise à jour

$C_{R/M}$: coût de mise à jour de l'état d'une ressource

$C_{Sc/C}$: coût de la mise en place de la section critique

$C_{Asc/M}$: coût d'entrée en section critique

Le gain lié à la parallélisation du code dépend, quant à lui, des ressources fournies (vitesse du processeur, mémoire vive disponible, etc.) par les sites sur lesquels vont être déployés les services du composants en question ainsi que des performances du réseau (bande passante, taux d'occupation, etc.). Ainsi, ce gain dépend fortement de l'architecture matérielle sur laquelle est déployée l'application.

Bien que le gain lié à la parallélisation du code ne soit généralement pas négligeable, le surcoût lié à l'assemblage du composant généré et au maintien de la cohérence peut être supérieur. Cependant, comme nous l'avons précisé dans nos motivations, notre objectif n'est pas d'améliorer les performances du composant mais plutôt de lui assurer une continuité de service quel que soit son contexte d'exécution. De plus, la parallélisation du code requise par la situation implique obligatoirement la mise en place de processus de maintien de la cohérence des ressources partagées par les différents fragments de l'application. Ainsi, le surcoût réel (*i.e.* surcoût ne prenant pas en considération le coût engendré par les opérations à réaliser obligatoirement pour assurer la continuité de service de l'application) lié à notre approche de ré-ingénierie est négligeable.

3.6 Conclusion

Nous avons présenté dans ce chapitre une approche permettant de réaliser l'adaptation structurelle de composants existants par leur ré-ingénierie. Cette adaptation est réalisée par l'intermédiaire d'un processus de transformation structurelle semi-automatique constitué de deux étapes nécessitant le code source du composant à adapter. La première étape consiste à fragmenter le composant adapté en fonction d'une spécification fournie par un acteur de l'adaptation. La deuxième étape est quant à elle dédiée à l'assemblage des composants générés et à son intégration dans l'application. Une autre étape, optionnelle peut être insérer au processus. Elle offre la possibilité aux acteurs de l'adaptation de configurer certaines propriétés du composant résultat du processus de transformation ou de lui intégrer de nouvelles propriétés telle que la distribution. Ce processus repose sur un modèle de composants auquel tout composant structurellement adapté doit être conforme afin de garantir son comportement.

Nous avons alors évalué notre approche d'adaptation structurelle par la ré-ingénierie de composants existants. Cette évaluation nous a permis de constater que notre approche d'adaptation structurelle engendre un surcoût pouvant être diminué par la parallélisation des services et la prise en compte des ressources disponibles sur chaque site de déploiement d'un composant. Cependant, ce surcoût est indispensable pour garantir la continuité de service d'un composant ou d'une application.

Dans ce chapitre, la transformation d'un composant existant pour le rendre structurellement adapté est réalisée de manière statique ; ce qui impose l'arrêt du composant pour réaliser l'adaptation. Néanmoins, il est clair que, dans certains cas, où la continuité de service est primordiale, l'approche statique que nous avons proposée, ne peut répondre à ce type d'attente. En se basant sur ces considérations, notre objectif, par la suite, va être de proposer une approche permettant la restructuration dynamique d'un composant logiciel. Un composant pourrait alors adapter sa structure en fonction des variations de son contexte d'exécution (voir Chapitre 4).

De plus, nous avons pu noter que la spécification du résultat de l'adaptation (*i.e.* spécification de la structure interne attendue) est fournie par un acteur externe de l'adaptation ; ce qui peut être un inconvénient majeur dès lors que l'adaptation est réalisée dynamiquement dans des environnements où le contexte d'exécution change en permanence tels que les environnements ubiquitaires. Ainsi, notre objectif dans le chapitre suivant est d'automatiser les actions de spécification du résultat de l'adaptation mais aussi de déclenchement de phases d'adaptation.

CHAPITRE 4

Auto-adaptation structurelle de composants logiciels

4.1 Introduction

Dans le chapitre précédent, nous avons étudié l'adaptation structurelle par la ré-ingénierie de composants logiciels existants. Cette stratégie que l'on peut qualifier de « statique » impose l'arrêt du composant pour réaliser l'adaptation. Néanmoins, il est clair que, dans certains cas, où la continuité de service est primordiale, l'approche que nous avons proposée ne peut répondre à ce type d'attente. Ainsi, notre objectif, dans ce chapitre, est de proposer une approche permettant la restructuration d'un composant logiciel pendant son exécution.

Dans certains types d'environnements, tels que les environnements ubiquitaires, le contexte d'exécution d'une application évolue en permanence. De ce fait, une application ou un composant logiciel exécuté dans un tel contexte, peut fournir un ensemble de services à un instant et, à l'instant suivant, il peut ne plus être en mesure de les fournir car, par exemple, les ressources requises à leur exécution sont devenues insuffisantes ou bien les services sont devenus inappropriés à la situation.

Du fait de l'évolution permanente du contexte d'exécution d'une application, il ne peut être envisagé une adaptation manuelle de chaque composant à cause d'une part, du temps de réaction (relatif au déclenchement et à la paramétrisation de l'adaptation) qui peut être variable en fonction des situations et d'autre part, du coût lié à la maintenance. En effet, un administrateur doit, pour garantir la continuité de service de l'application, analyser le contexte en permanence afin de déclencher une phase d'adaptation, de la paramétrer et de vérifier son résultat. Or, compte tenu de la rapidité d'évolution du contexte, cette stratégie n'est envisageable.

Ainsi, la seule solution envisageable dans ce type de situation est l'automatisation de la prise de décisions et de la réalisation de l'adaptation. Dans ce cas, l'application doit elle-même prendre en considération les changements de son contexte d'exécution en s'adaptant automatiquement à des situations nouvelles.

En se basant sur ces considérations, notre objectif, dans ce chapitre, est de proposer une approche d'auto-adaptation structurelle de composants logiciels. Une telle approche impose aux composants d'être dotés de mécanismes permettant d'automatiser les prises de décisions et la réalisation de l'adaptation de manière dynamique. Nous appliquerons alors ces mécanismes d'auto-adaptation à un type particulier d'environnement : les environnements ubiquitaires.

La mise en œuvre de l'auto-adaptation sur un composant (*i.e.* adaptation niveau micro) peut avoir des impacts sur les autres composants de l'application. De ce fait, nous envisageons également d'étendre notre stratégie afin d'adapter une application d'un point de vue global (*i.e.* adaptation niveau macro).

4.2 Objectif et proposition : l'auto-adaptation structurelle dynamique

4.2.1 Objectif général et analyse des besoins

Nous définissons l'auto-adaptation structurelle d'un composant logiciel comme étant la capacité du composant à modifier automatiquement sa structure en fonction de son contexte d'exécution courant. Ainsi, un composant disposant d'une telle capacité doit être capable d'acquérir des informations sur son contexte afin de générer une structure adaptée à la situation. Un composant structurellement auto-adaptatif doit donc être capable de réaliser lui même ces opérations. Pour cela, nous devons proposer **un modèle de composants structurellement auto-adaptatifs** intégrant ces trois phases. De plus, afin de prendre en compte les composants existants, nous devrons proposer **un processus de ré-ingénierie permettant d'obtenir un composant conforme à ce modèle à partir d'un composant existant**.

1. *Acquisition du contexte*
   Tout d'abord, un composant auto-adaptatif doit disposer de mécanismes lui permettant d'acquérir des informations sur son contexte d'exécution qui peuvent avoir des impacts sur la structure du composant à adapter. De ce fait, il est indispensable de définir précisément **un modèle de contexte pertinent** (*i.e.* contexte ayant une influence sur la structure d'un composant) afin de faciliter la prise de décisions au niveau du déclenchement et de la spécification de l'adaptation. Le modèle de contexte pertinent que nous proposons est détaillé dans la section 4.3.1 ;

2. *Prise de décisions*
   Un composant auto-adaptatif doit disposer de mécanismes lui permettant de traiter les informations contextuelles acquises afin de prendre les décisions nécessaires à son adaptation. La prise de décisions fait référence à deux activités entrant dans le cadre de l'auto-adaptation :

   (a) *le déclenchement de l'adaptation*
   La première des activités de prise de décisions dans le cadre de notre approche d'auto-adaptation est relative au déclenchement de l'adaptation structurelle du composant. Cette activité doit être réalisée en fonction du contexte d'exécution du composant : si le composant détecte un changement significatif dans son contexte, celui-ci doit déclencher une adaptation structurelle. Ainsi, il est nécessaire de concevoir **une stratégie de déclenchement de l'adaptation** permettant de définir les variations de contexte au delà desquelles une adaptation doit être réalisée ;

   (b) *la spécification du résultat de l'adaptation*
   Une fois l'adaptation déclenchée, le composant doit déterminer lui même une structure adaptée à son nouveau contexte d'exécution. Pour cela, il est nécessaire de proposer **une stratégie de génération automatique de la spécification d'une structure pour le composant**, qui soit adaptée à la nouvelle situation.

3. *Réalisation de l'adaptation structurelle*
   Enfin, la dernière phase de l'auto-adaptation consiste à modifier de manière dynamique la structure du composant afin de la rendre conforme à la spécification générée lors de la phase précédente et à se redéployer si nécessaire. Ainsi, le composant doit être structurellement et dynamiquement adaptable.

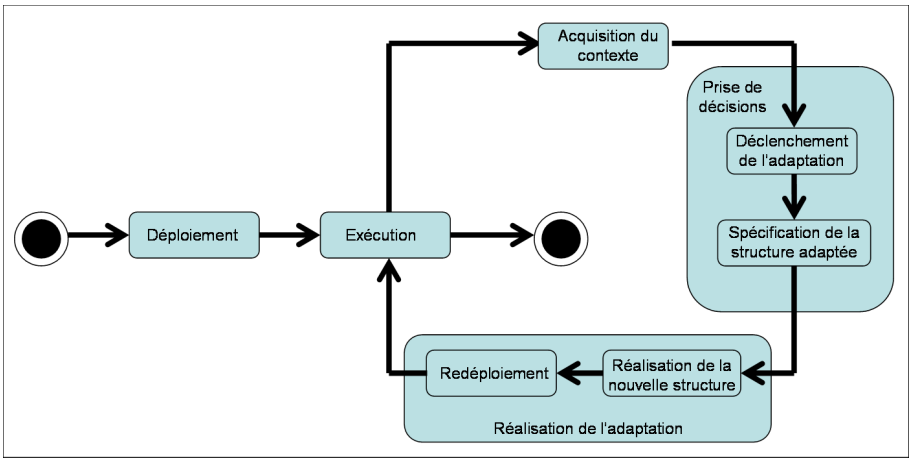

Figure 4.1 – Processus d'auto-adaptation structurelle dynamique

4.2.2 Objectif et analyse des besoins relatifs aux environnements ubiquitaires

Dans le cadre de notre application aux environnements ubiquitaires, notre objectif est de proposer une stratégie d'auto-adaptation permettant à un composant de prendre en compte certaines propriétés de l'ubiquité telles que l'hétérogénéité des moyens d'exécution et de communication, les variations des ressources disponibles ainsi que l'ouverture du système, tout en maintenant une certaine qualité de service pour l'utilisateur de l'application.

En fait, l'auto-adaptation structurelle d'un composant logiciel dans un tel environnement a pour objectif de réaliser l'adéquation entre son architecture logicielle (*i.e.* sous-composants, connecteurs et configuration) et l'architecture matérielle disponible (*i.e.* composants matériels[1], connecteurs matériels et configuration) tout en tenant compte de l'utilisateur et des éventuelles déconnexions de sa machine et des autres sites de déploiement de l'application. Cette adéquation doit être réalisée en garantissant la continuité de service mais également en assurant la meilleure qualité de service possible.

Concernant le premier objectif, un composant doit pouvoir détecter si les ressources matérielles fournies par son site de déploiement sont suffisantes pour garantir sa continuité de service. C'est pourquoi il doit comparer, durant son exécution, les ressources matérielles qu'il requiert pour fonctionner avec celles disponibles sur son site de déploiement. Si ces ressources matérielles se révèlent insuffisantes, il doit s'adapter automatiquement de manière à assurer sa continuité de service. L'adaptation réside alors dans sa fragmentation en sous-composants fournissant chacun un sous-ensemble de services fournis par le composant adapté. Ces sous-composants seront alors redéployés sur les différents nœuds de l'infrastructure distribuée disponible, en fonction du contexte ; le choix des sites de déploiement étant primordial pour garantir la continuité de service.

La qualité de service peut, quant à elle, être garantie en assurant une meilleure répartition des charges liées à l'utilisation de ses services. De la même manière que précédemment, un composant peut être fragmenté en sous-composants distribués sur l'infrastructure disponible. Le choix de la répartition des services fournis par chaque sous-composant ainsi que leur site de déploiement est crucial pour assurer une répartition des charges optimale. Ainsi, la nouvelle structure du composant doit être déterminée en fonction de son contexte courant.

[1] Nœuds de l'infrastructure distribuée.

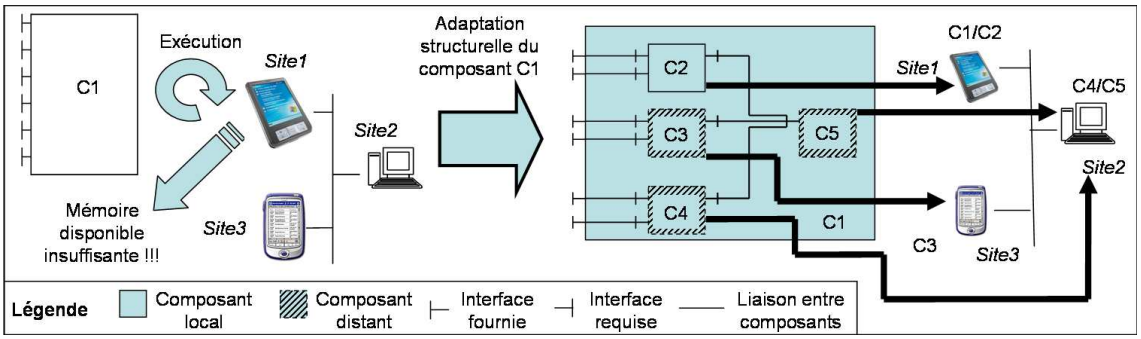

Figure 4.2 – Adaptation structurelle dynamique en fonction du contexte

Par exemple, la figure 4.2 montre un composant C_1 qui est déployé sur une machine à ressources limitées ($site1$). Imaginons que pendant l'exécution du composant C_1, la répartition des charges existante soit remise en question du fait de l'évolution de l'infrastructure de déploiement (ici, la capacité mémoire disponible sur *Site1* devient insuffisante pour garantir la continuité des services du composant C_1 sur ce site). Dans ce cas, C_1 doit réagir en s'adaptant à ces nouvelles conditions d'utilisation. Une solution consiste à fragmenter C_1 en quatre nouveaux composants appelés C_2, C_3, C_4, C_5. Ensuite, ces derniers sont redéployés sur des nœuds de l'infrastructure déterminés en fonction du contexte : C_2 est déployé sur *site1* car, d'une part, les ressources disponibles sur *site1* sont suffisantes pour le déployer et, d'autre part, les services qu'il propose font partie des services les plus utilisés par l'utilisateur de ce site. Il est donc préférable de déployer le composant C_2 sur *site1* car en cas de déconnexion de ce site à l'infrastructure, les services qu'il fournit resteront disponibles à l'utilisateur ; les composants C_4 et C_5 sont déployés sur *Site2* qui est une unité fixe de l'infrastructure (donc le risque de déconnexions de *Site2* à l'infrastructure distribuée est minimum) dotée de caractéristiques largement supérieures à celles proposées par *site1* (par exemple, meilleure capacité de stockage, plus grande vitesse de processeur, etc.) et dont les ressources sont suffisantes pour déployer ces deux composants ; et enfin, le composant C_5 est déployé sur *Site3* car d'une part, ce dernier dispose des ressources nécessaires à son déploiement et à son exécution et d'autre part, sur ce site, est déjà déployé un ensemble de composants contenus dans l'application et utilisant fortement les services fournis par le composant C_5. Ainsi, le déploiement de C_5 sur *Site3* permettra de diminuer la fréquence de communications distantes entre *Site1* et *Site3* et donc, améliorer les performances de l'application.

Ainsi, l'adaptation structurelle du composant C_1 aura permis de conserver la continuité de service de l'application mais aussi d'améliorer sa qualité de service. En effet, les composants C_4 et C_5 sont déployés sur une machine dotée de meilleures caractéristiques techniques donc le temps d'exécution des services fournis par ces deux composants sera diminué (ce qui peut compenser les pertes liées à la communication distante entre les deux machines : *Site1* et *Site2*). De plus, le composant C_5 est déployé *Site3* ; ce qui permet de réduire la fréquence des connexions distantes entre les composants de l'application.

4.3 Démarche d'auto-adaptation structurelle dynamique

Tout d'abord, nous devons établir un modèle dynamique de l'auto-adaptation structurelle afin de montrer les différents états dans lesquels un composant peut se trouver au cours de son adaptation ainsi que les transitions entre ces états. Ce modèle est présenté dans la figure 4.3.

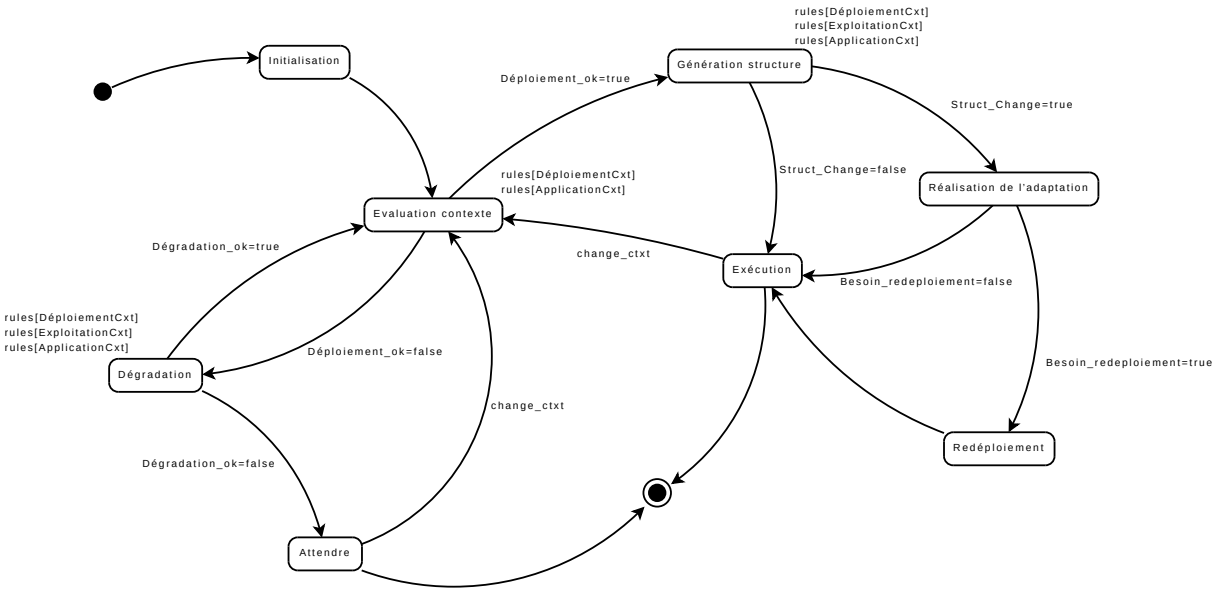

Figure 4.3 – Modèle dynamique de l'adaptation

L'état *Initialisation* correspond au démarrage d'un composant. Cette phase est immédiatement suivie d'une évaluation de la compatibilité des différentes contraintes de ce composant avec l'architecture matérielle disponible (les ressources disponibles sont-elles suffisantes pour garantir la continuité de service du composant). Cette évaluation correspond à l'état *Evaluation contexte*.

Si la configuration de l'infrastructure disponible ne permet pas de déployer le composant correctement (*i.e.* les ressources disponibles sont insuffisantes pour garantir le déploiement du composant), le composant peut passer en mode dégradé (*i.e.* état *Dégradation*), c'est-à-dire que certains services (les moins appropriés au contexte d'exécution) ne seront pas déployés de manière à optimiser l'occupation des ressources. Tant que le composant ne peut être déployé, la dégradation du composant sera effective. Si aucune dégradation ne peut être envisagée, le composant passe en phase d'attente jusqu'à un changement dans le contexte d'exécution.

Si la configuration de l'infrastructure disponible permet de déployer le composant correctement (*i.e.* les ressources disponibles sont suffisantes pour garantir le déploiement du composant), le composant passe dans l'état de sélection de la structure adaptée au contexte courant (*i.e.* état *Génération structure*). Cette phase de spécification d'une structure adaptée est détaillée dans la section 4.5.2. Elle nécessite la mise en œuvre de règles de sélection basée sur les éléments du contexte. Une fois que la spécification de la nouvelle structure est générée, le composant passe dans l'état de *réalisation de l'adaptation*. Au cours de cette phase, la structure du composant est modifiée de manière à la rendre conforme à la spécification générée lors de la phase précédente. Ensuite, si besoin est, le composant est redéployé (*i.e.* état *redéploiement*). Le composant peut alors passer dans l'état d'exécution jusqu'à ce qu'un changement de contexte nécessite une réévaluation de la satisfaction de ses besoins en ressources disponibles (*i.e.* état *Evaluation contexte*).

4.3.1 Acquisition du contexte : modèle de contexte pertinent pour l'adaptation structurelle

Afin de concevoir des composants capables d'adapter automatiquement leur structure, nous devons répertorier les éléments du contexte ayant un impact direct ou indirect sur la structure d'un composant logiciel. Nous avons alors isolé trois types de contexte (*ContextPart*) répondant à ce critère : le contexte de déploiement (*DéploiementCxt*) qui fait référence aux propriétés de l'architecture matérielle disponible, le contexte d'exploitation (*ExploitationCxt*) qui fournit des informations sur les caractéristiques de l'utilisateur et de l'environnement d'utilisation et enfin, le contexte applicatif (*ApplicationCxt*) qui décrit les propriétés de l'architecture logicielle. La figure 4.4 présente le méta-modèle que nous avons retenu. Chaque partie contient un certain nombre d'éléments de contexte (*ContextElement*) organisés hiérarchiquement. Un *ContextElement* représente une information élémentaire du contexte ou un ensemble d'informations hiérarchiques (*ContextCategory*). La prise en compte de ces différents contextes va permettre au composant de déclencher des phases d'adaptation et de déterminer une nouvelle structure, adaptée à la nouvelle situation.

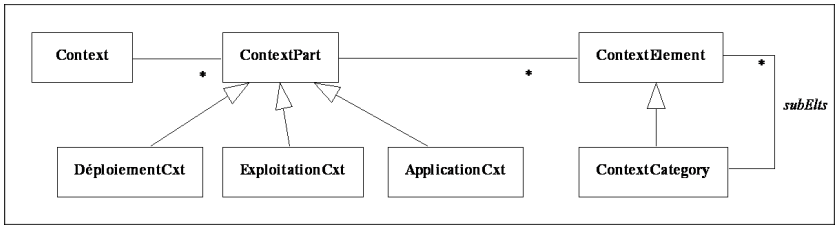

Figure 4.4 – Méta-modèle du contexte

Comme nous l'avons évoqué précédemment, nous avons isolé trois types de contexte ayant un impact direct ou indirect sur la structure des composants. Ces trois types de contexte sont les suivants :

- *le contexte de déploiement (DéploiementCxt)*

  Description : le contexte de déploiement fait référence aux caractéristiques techniques (propriétés des composants matériels et des connecteurs matériels) de l'infrastructure disponible (*i.e.* architecture matérielle) et à sa configuration.
  Nous pouvons distinguer deux types de caractéristiques techniques relatives à l'architecture matérielle : d'une part, les caractéristiques fixes et d'autre part les caractéristiques évolutives. Les caractéristiques fixes correspondent aux données qui ne vont pas évoluer au cours de l'exécution de l'application. Il s'agit par exemple de la vitesse du processeur, du système d'exploitation, de la mémoire maximale disponible, etc. Leur prise en compte par le composant est généralement réalisée au moment de son déploiement.
  Contrairement aux caractéristiques fixes, les caractéristiques évolutives varient tout au long du cycle de vie de l'application. Nous pouvons citer à titre d'exemple : le taux d'utilisation du processeur, le niveau de batterie restant, la bande passante du réseau, etc. Afin de garantir un bon fonctionnement du composant, les données de ce type doivent être contrôlées tout au long du cycle de vie du composant. Cette opération est réalisée par l'intermédiaire des sondes qui sont généralement constituées de capteurs logiciels ou matériels.

Raisons de la prise en compte : tout d'abord, la structure d'un composant doit être la plus adaptée à son infrastructure de déploiement. Par exemple, un composant monolithique risque de ne pas pouvoir être déployé sur une machine alors qu'un composant composite peut être réparti sur l'infrastructure distribuée disponible. En fait, un composant monolithique peut être déployé sur une machine seulement si les ressources matérielles disponibles le permettent. De plus, étant données que les ressources matérielles disponibles sur une machine de déploiement sont en perpétuelle évolution, il se peut qu'au cours de son exécution, la continuité de service du composant ne soit plus garantie (car les ressources matérielles nécessaires sont devenues insuffisantes). Dans ce cas, tous les services qu'il fournit deviennent indisponibles. Le composant peut alors, grâce à l'adaptation structurelle, modifier sa structure et distribuer les services qu'il fournit sur l'infrastructure disponible (dans le cas où le transfert du composant dans son intégralité, sur un site de déploiement distant, ne peut être envisagé pour diverses raisons telles que les déconnexions trop fréquentes de la machine de l'utilisateur).

Prise en compte : le contexte de déploiement et plus particulièrement les caractéristiques évolutives, est utilisé par le composant pour déclencher des phases d'adaptation. Par exemple, si le niveau de batterie ou bien la mémoire disponible dépasse un certain seuil (défini par le concepteur dans le cadre du contexte applicatif), une phase d'adaptation doit alors être déclenchée afin de garantir une continuité de service.
Le contexte de déploiement est également utilisé pour déterminer sur quels sites peuvent être déployés les services des composants en fonction des ressources matérielles qu'ils requièrent pour garantir une continuité de service ou maintenir la qualité de service ;

- *Le contexte d'exploitation (ExploitationCxt)*

  Description : le contexte d'exploitation contient deux types de données contextuelles : d'une part, des informations sur le profil de l'utilisateur (caractéristiques et préférences utilisateur) ; le profil de l'utilisateur comprend des données personnelles (âge, etc.) et des données relatives à son activité (profession, etc.). Il est généralement fourni par l'utilisateur lui-même.
  D'autre part, des informations sur les conditions d'utilisation des services du composant ; les informations sur les conditions d'utilisation permettent à l'application d'évaluer la situation dans laquelle se trouve l'utilisateur au moment de son interaction. Par exemple, l'environnement d'exécution est bruyant, sombre, etc.

  Raisons de la prise en compte : la structure d'un composant peut également dépendre de son contexte d'exploitation. En effet, il apparaît plus judicieux, dans certains cas, de déployer un maximum de services susceptibles d'être les plus utilisés (en fonction des ressources matérielles disponibles), sur la machine de l'utilisateur du fait des risques de déconnexion à l'infrastructure distribuée. De ce fait, l'utilisateur pourra accéder aux services qu'il a le plus besoin malgré d'éventuelles déconnexions. Pour cela, le choix des services à déployer sur la machine de l'utilisateur est déterminant. Il doit donc tenir compte de l'utilisateur et de la situation dans laquelle il se trouve lors de l'interaction avec l'application.

  Prise en compte : Le contexte d'exploitation est utilisé pour établir une classification des services fournis par le composant en fonction des besoins liés à leur utilisation, de manière à en déployer un maximum sur la machine de l'utilisateur ;

- *Le contexte applicatif (ApplicationCxt)*

  Description : tout d'abord, le contexte applicatif regroupe les propriétés relatives aux besoins des services telles que la taille mémoire requise pour exécuter un service, la fréquence du processeur requise, le système requis, etc. Ces données sont fournies par le concepteur de chaque service d'un composant.
  Les données relatives au comportement des composants contiennent des informations relatives aux appels de services (nombre de fois où un service est invoqué, probabilité d'invocation d'un service dans un contexte donné, etc.) ainsi qu'aux déclenchements de phases d'adaptation. Leur acquisition nécessite la mise en place d'un historique permettant de garder une trace du comportement du composant ainsi que des outils d'introspection sur le composant. L'historique de l'application permet de mémoriser les évènements qui se sont produits dans le passé (appels de services, déclenchement de phases d'adaptation, etc.), ainsi que leurs conséquences sur les ressources matérielles évolutives.

  Raisons de la prise en compte : l'adaptation de la structure d'un composant dépend également des besoins liés à l'utilisation de ses services ainsi que leur comportement.
  Comme nous l'avons évoqué précédemment, la structure d'un composant doit être mise en correspondance avec l'architecture matérielle disponible. Pour cela, les besoins liés à l'utilisation des services d'un composant doivent être exprimés par son concepteur. Ces informations contextuelles font parties du contexte applicatif car elles sont spécifiques aux composants logiciels concernés.
  L'adaptation de la structure d'un composant dépend également de son comportement. Par exemple, dans le cas où deux services sont fortement liés (appel d'un service lorsque l'autre est appelé), il est préférable de regrouper ces deux services dans un même composant déployé sur une même machine de manière à éviter les surcoûts liés à d'éventuelles communications distantes, et ainsi, assurer une certaine complétude des services déployés sur une même machine (*i.e.* les services les plus dépendants étant regroupés sur une même machine).

  Prise en compte : la prise en compte du contexte applicatif a pour objectif d'une part, de réaliser l'adéquation entre l'architecture logicielle et l'architecture matérielle en tenant compte des besoins en ressources de chaque service et d'autre part, d'optimiser le choix des composants et de leurs sites de déploiement en évaluant les dépendances entre les services et en proposant une répartition visant à minimiser les connexions distantes entre les services, ainsi que de tirer partie des évènements du passé afin d'anticiper le futur. Par exemple, si l'application constate que l'exécution d'un service a engendré une forte consommation d'énergie, ce service pourra être redéployé sur des unités disposant de batteries à forte autonomie.

4.3.2 Prise de décisions : stratégie de génération automatique de la spécification d'une structure pour le composant

La stratégie de génération automatique de la spécification d'une structure pour le composant, qui soit adaptée à la situation, consiste à fournir un ensemble de règles qui vont, en fonction des objectifs fixés initialement par les acteurs de l'adaptation, permettre d'obtenir un composant répondant aux nouvelles attentes liées à son utilisation. En fait, ces règles vont être utilisées pour associer à chaque service fourni par le composant à adapter, un site de déploiement. Chaque ensemble de services associé à un site

sera alors regroupé au sein d'un sous-composant qui sera déployé sur le site en question. Ainsi, nous obtiendrons une structure adaptée au contexte d'exécution du composant.

Dans le cadre de notre application aux environnements ubiquitaires, nous optons pour la stratégie suivante : comme nous l'avons évoqué précédemment, pour limiter les risques de non-disponibilité de services liés à une éventuelle déconnexion de la machine de l'utilisateur, nous avons pour objectif de déployer un maximum de service sur cette machine. Cependant, en cas d'impossibilité de déploiement de tous les services sur cette machine (à cause de ressources matérielles insuffisantes), la distribution des services est inévitable. Aussi, pour éviter les problèmes liés à la déconnexion des machines susceptibles de déployer une partie des services du composant adapté, le voisinage proche de la machine de l'utilisateur doit être privilégié de manière à garantir une certaine qualité de services. La proximité d'un voisin (nœud de l'infrastructure distribuée à laquelle la machine de l'utilisateur appartient) est calculée en fonction des caractéristiques des connexions disponibles entre les deux nœuds : les plus proches sont ceux qui possèdent une meilleure connexion (*i.e.* meilleure bande passante, etc.). Également, les entités fixes (*i.e.* serveurs) doivent être privilégiées, de manière à limiter les risques de déconnexion. Dans l'impossibilité de déployer les services distants sur de telles machines, il peut être envisagé des stratégies de duplication avec maintenance des états de chaque copie. Dans ce cas, les connecteurs du composant sous format canonique doivent être capables de sélectionner automatiquement une copie disponible du « composant primitif » fournissant le service à invoquer.

Dans le cas où des services sont déployés sur des sites distants et la machine de l'utilisateur est déconnectée de l'infrastructure distribuée, le composant peut passer alors en mode « dégradé ».Ainsi, certains services fournis par le composant dans ce mode sont rendus indisponibles :

- les services fournis par des « composants primitifs » déployés sur des machines distantes ;
- les services requérant directement ou indirectement des services fournis par des « composants primitifs » et déployés sur des machines distantes.

4.3.3 Réalisation de l'adaptation structurelle dynamique

L'auto-adaptation structurelle dynamique d'un composant logiciel par sa fragmentation en composants élémentaires (*i.e.* sous-composants) peut être réalisée suivant deux stratégies (voir Figure 4.5).

4.3.3.1 Adaptation basée sur la transformation à la volée du code

La première stratégie (voir Figure 4.5, cas A) consiste à générer, à la volée, le code binaire du composant adapté et à remplacer l'instance du composant à adapter par une nouvelle instance (correspondant au composant adapté), tout en assurant le transfert d'état entre ces deux instances.

Un nouveau composant dont la structure est adaptée au contexte courant est généré en utilisant le processus de transformation structurelle défini dans le chapitre précédent, pendant la phase d'exécution. L'instance du composant initial est alors remplacée par une instance du composant adapté en trois phases : (1) la première phase réside dans la déconnexion du composant à adapter, (2) ensuite, le composant adapté est connecté à l'application (*i.e.* remplacement de composants) et enfin, (3) l'état du composant initial est transféré au composant adapté qui peut alors être activé. L'état d'un composant fait référence aux ressources logicielles qui sont inclues dans sa structure. Ainsi, pour préserver la cohérence du composant, l'état des ressources qu'il utilise doit être le même avant et après son adaptation. Il est

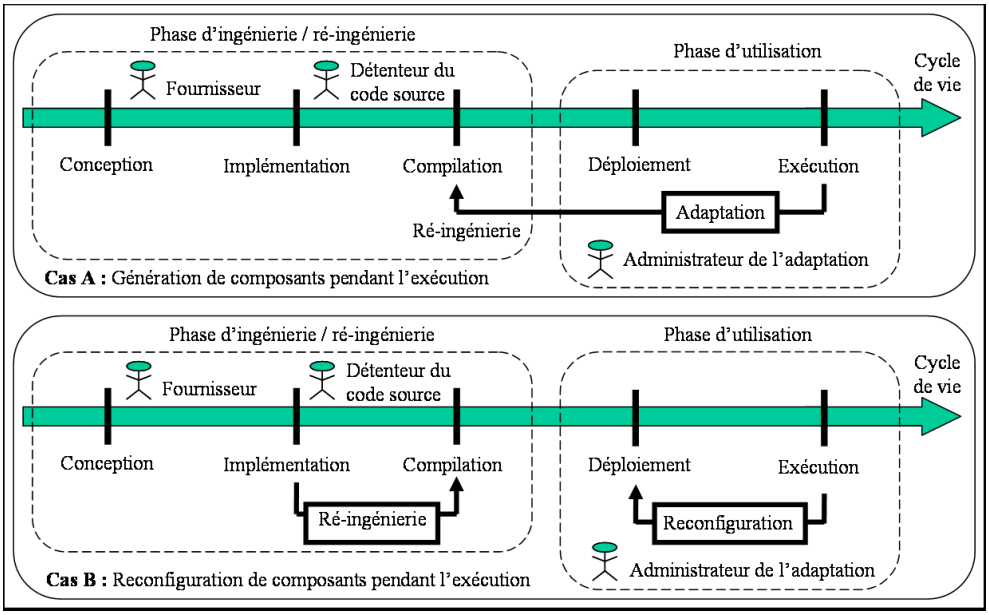

Figure 4.5 – Les stratégies d'adaptation structurelle dynamique

donc nécessaire de transférer l'état de chaque ressource du composant.

Cette stratégie d'adaptation structurelle dynamique par la génération à la volée de code binaire présente les avantages suivant :

- *possibilité d'adapter les composants existants*
  L'adaptation peut être réalisée sur tout type de composants. En effet, étant donné que la réalisation de l'adaptation est mise en œuvre au travers de notre processus de ré-ingénierie, l'approche est indépendante de tout modèle. De plus, elle peut être appliquée sur des composants existants ;

- *aucune spécificité sur l'infrastructure logicielle de déploiement requise*
  L'approche par la génération à la volée de composants ne nécessite pas de caractéristiques spécifiques au niveau de l'infrastructure logicielle de déploiement.

Cependant, cette stratégie souffre de certains désavantages. Parmi ceux-ci, nous pouvons citer, à titre d'exemple les suivants :

- *disponibilité du code source pendant la phase d'utilisation*
  Le code source du composant doit être disponible pendant toute la durée de la phase d'utilisation du composant. En effet, il doit être utilisé afin de générer le code source du composant conforme à la spécification demandée par l'administrateur ;

- *nécessité d'un environnement de compilation et de génération de code binaire*
  Un environnement de compilation et de génération de code binaire doit également être présent sur la machine de déploiement du composant à adapter. Il doit permettre de transformer le code source du nouveau composant en code binaire exécutable par la machine ;

- *impossibilité de vérification ou de modification du code généré*
  Étant donné que le composant est adapté pendant son exécution, donner au concepteur la possibilité de valider le code source généré n'est pas envisageable ;

- *temps d'adaptation du composant*
  Celui-ci inclut le temps de réalisation du processus de ré-ingénierie du composant initial (analyse et génération de code) ;

- *arrêt du composant requis*
  Le composant entier doit être arrêté pour être adapté. En effet, il est remplacé par un nouveau composant (le composant adapté). De ce fait, afin d'assurer la cohérence de l'état du composant, un transfert d'état du composant avant son adaptation vers le composant adapté doit être opéré. Ce transfert doit être réalisé sur l'ensemble de ressources logicielles contenues dans le composant.

4.3.3.2 Adaptation basée sur un modèle de composants dynamiquement reconfigurables

La seconde approche d'auto-adaptation structurelle dynamique est basée sur la proposition d'un modèle de composants dynamiquement et structurellement reconfigurables (voir Figure 4.5, cas B) basé sur la réification des interfaces en composants. Nous détaillons ce modèle dans la section 4.4.

L'adaptation structurelle dynamique d'un composant conforme à ce modèle est réalisée par un processus de reconfiguration qui consiste à modifier la structure interne du composant en se basant sur une spécification fournie par un acteur externe de l'adaptation. En fait, ce processus est chargé de créer dynamiquement les nouveaux sous-composants spécifiés par encapsulation de sous-composants et à les redéployer, si nécessaire.

Afin de garantir le comportement du composant issu de l'adaptation, le processus de reconfiguration dynamique doit préserver sa consistance. Pour cela, trois propriétés doivent être vérifiées :

Propriété 1 : *intégrité fonctionnelle*
Le composant doit conserver son intégrité fonctionnelle : les services du composant doivent être les mêmes avant et après une phase de reconfiguration ;

Propriété 2 : *états stables avant la configuration*
Les composants concernés par la reconfiguration doivent être dans des états mutuellement consistants (*i.e.* états stables) : si une phase de reconfiguration est initiée, les sous-composants concernés doivent être dans un état stable c'est-à-dire tous les services en cours d'exécution doivent être arrêtés avant de réaliser toute opération de reconfiguration ;

Propriété 3 : *états cohérents après la configuration*
Les opérations de reconfiguration ne doivent pas altérer l'état des entités concernées par la reconfiguration. Ainsi, l'état du composant et de ses sous-composants doit être le même avant et après une phase de reconfiguration. De ce fait, dans certains cas, des opérations de transfert d'état peuvent être nécessaires.

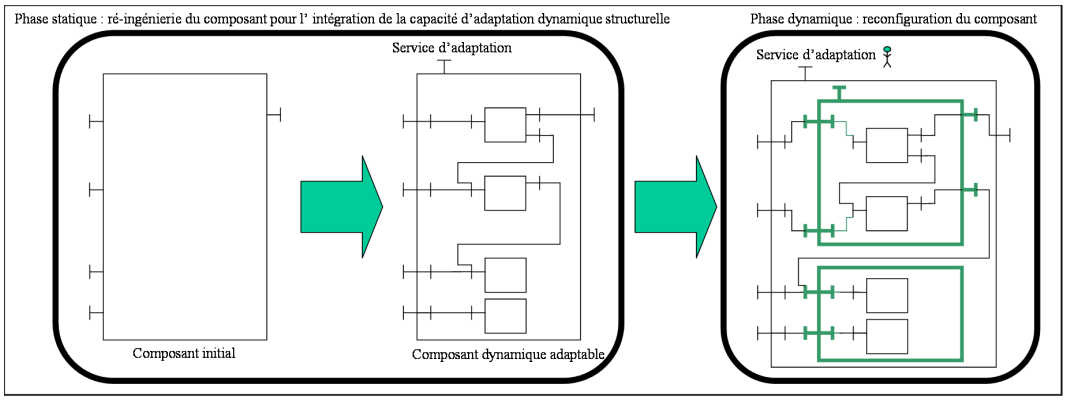

Figure 4.6 – Processus de reconfiguration du composant à adapter au moment de l'exécution

Cette stratégie d'adaptation structurelle dynamique basée sur un modèle de composants dynamiquement structurellement reconfigurables présente de nombreux avantages :

- *code source non nécessaire lors de la phase d'utilisation*
  Contrairement à la première stratégie, le code source du composant n'est pas nécessaire lors de la phase d'utilisation. En effet, l'étape de reconfiguration ne nécessite pas la génération de nouveaux codes. De ce fait, la disponibilité d'un environnement de transformation et génération de code binaire n'est pas nécessaire au moment du déclenchement de l'adaptation structurelle ;

- *possibilité d'agir sur des composants existants*
  Cette stratégie nécessite que le composant soit conforme à un modèle particulier. Cependant, nous proposons un processus de ré-ingénierie permettant de générer un composant conforme à ce modèle à partir d'un composant existant. Ainsi, il peut être envisagé que la ré-ingénierie soit réalisée par le concepteur du composant ou bien le créateur de l'application qui le contient de telle sorte que le code du composant à adapter ne soit pas accessible à l'utilisateur de l'application et au créateur de l'application ;

- *arrêt partiel du composant*
  Lors de l'adaptation d'un composant, ce dernier peut ne pas être entièrement stoppé. Seuls, les services mis en jeu lors de la reconfiguration de la structure du composant sont rendus temporairement indisponibles. De plus, le transfert d'état ne se fait plus au niveau du composant global mais au niveau des sous-composants réifiant les services mis en jeu.

Les inconvénients de cette stratégie sont les suivants :

- *surcoût permanent*
  Cette stratégie nécessite que le composant soit conforme à un modèle particulier que nous définissons. Cependant, comme nous pourrons le constater dans la section 4.8, ce modèle introduit nécessairement un surcoût pouvant entraîner une dégradation des performances du composant pouvant être compensé par la distribution des sous-composants générés ;

- *besoin de propriétés spécifiques de l'infrastructure logicielle de déploiement*
  Comme nous avons pu le constater dans notre état de l'art présenté dans le chapitre 1, les propriétés d'introspection et d'intercession sont présentes dans la grande majorité des infrastructures logicielles de déploiement existantes ; ce qui rend cette stratégie d'adaptation par reconfiguration applicable pour la plupart des modèles de composants existants dans la littérature. Dans le cas où l'introspection et l'intercession ne figurent pas parmi les propriétés de l'infrastructure de déploiement, seule la première stratégie peut être appliquée.

Compte tenu des avantages et des inconvénients de chaque stratégie, nous avons privilégié la stratégie d'adaptation dynamique par la reconfiguration au dépend de la première stratégie de génération de composants à la volée car elle offre plus de sureté en terme d'adaptation : d'une part le code source n'est pas accessible pendant la phase d'utilisation et d'autre part, la génération de code à la volée ne présente pas toutes les garanties car le code ne peut pas être vérifié. De ce fait, nous avons choisi de développer la stratégie d'adaptation basée sur un modèle de composants dynamiquement structurellement reconfigurables dans le reste de ce chapitre.

4.4 Modèle et architecture de composants structurellement et dynamiquement auto-adaptatifs

4.4.1 Modèle de composants structurellement dynamiquement adaptables

Un composant répondant à cette propriété est un composant conforme à un format canonique (voir Figure 4.7). Un composant sous format canonique est un composant dont les sous-composants appelés « composants primitifs » ne peuvent être fragmentés : ce sont des composants monolithiques. Les composants primitifs forment ainsi les briques de base pour l'adaptation structurelle dynamique. De ce fait, toute nouvelle structure pourra être obtenue par encapsulation de ces « composants primitifs » au sein de nouveaux composants considérés alors comme des unités de déploiement à part entière. Ces opérations d'encapsulation seront réalisées par reconfiguration dynamique.

Par défaut, afin de maximiser le nombre de spécifications possibles pour la structure du composant, les « composants primitifs » sont obtenus par réification des interfaces fournies par le composant. Dans ce cas, les « composants primitifs » sont appelés « composant interface » car ils fournissent un seul et unique service. La figure 4.8 montre un exemple de composant logiciel sous format canonique obtenu après ré-ingénierie.

Par ailleurs, chaque « composant primitif » doit être doté d'interfaces lui permettant de réaliser un transfert d'état entre deux instances de ce même composant. En effet, comme nous l'avons expliqué précédemment, le transfert d'état entre deux instances est nécessaire pour garantir la cohérence du composant adapté lors du redéploiement de certains de ses sous-composants. Nous définissons l'état d'un composant comme l'ensemble des valeurs des ressources logicielles qu'il définit. Ainsi, deux types d'interfaces sont nécessaires pour assurer le transfert d'état d'une instance vers une autre :

- *une interface de sauvegarde de l'état courant*
  Cette interface permet de sauvegarder les valeurs des ressources logicielles d'un composant ;

- *une interface de chargement d'un état*
  Cette interface permet de charger l'état d'un composant de même type.

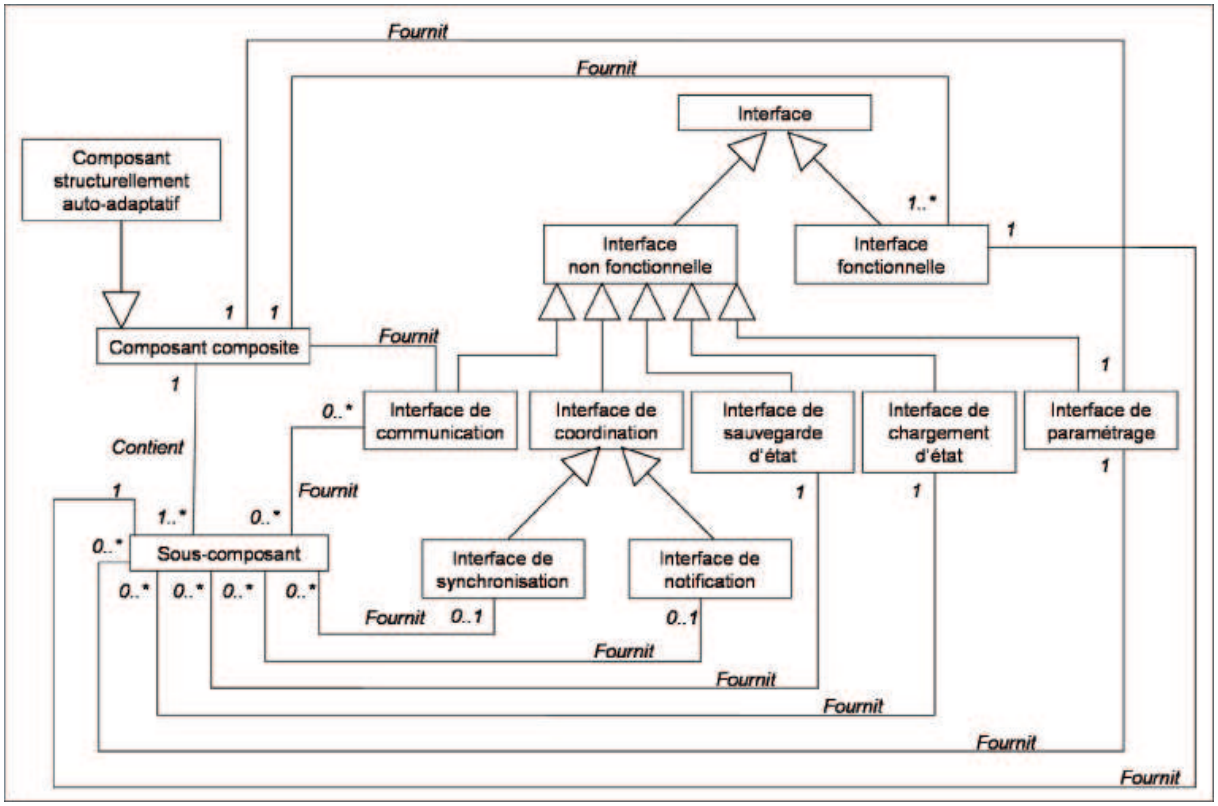

Figure 4.7 – Modèle de composants structurellement et dynamiquement adaptables

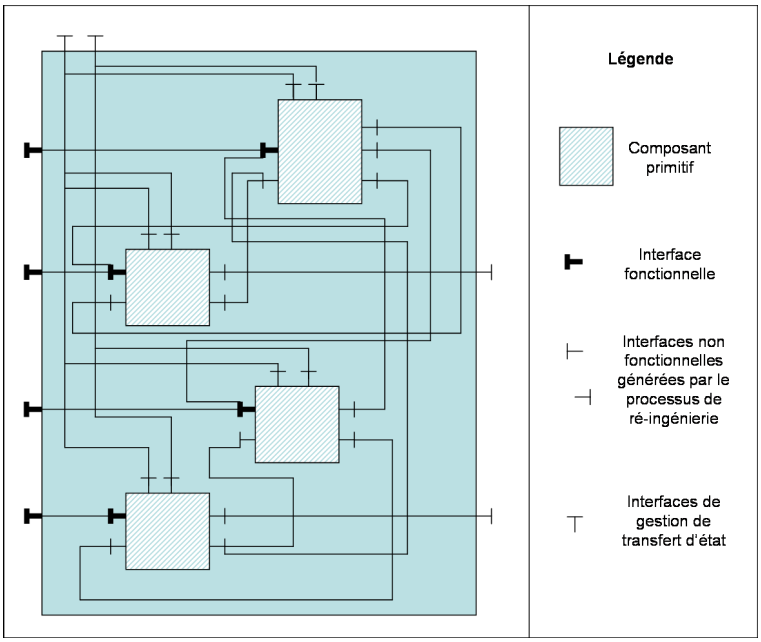

Figure 4.8 – Exemple de composant logiciel sous format canonique obtenu après ré-ingénierie

Infrastructure logicielle de déploiement Pour mettre en œuvre cette stratégie d'adaptation par la reconfiguration, l'infrastructure logicielle de déploiement des composants adaptés doit présenter les propriétés d'introspection et d'intercession. L'introspection va permettre au composant d'obtenir une vue sur sa structure courante ; ce qui va permettre de déterminer les opérations à réaliser pour obtenir la nouvelle structure du composant adapté à la situation. L'intercession va être utilisée pour modifier la structure courante afin de la rendre conforme à celle souhaitée au travers une spécification fournie par un acteur externe de l'adaptation. Ainsi, l'infrastructure logicielle de déploiement doit fournir un ensemble d'opérations primitives de reconfiguration dynamique :

- création/destruction de composant,
- création/destruction d'une liaison entre deux composants,
- la configuration d'un composant.

Ces opérations primitives vont servir de base à l'élaboration d'opérations de plus haut niveau permettant d'obtenir un composant dont la structure est conforme à une spécification fournie. La première opération, appelée *encapsulation* permet de générer de nouveaux composants par encapsulation de sous-composants existants. La deuxième opération appelée *éclatement* permet d'éclater un composant composite en le remplaçant par l'assemblage de sous-composants qu'il contient.

4.4.2 Architecture de composants structurellement auto-adaptatifs

Pour réaliser l'auto-adaptation de sa structure, un composant doit être capable de prendre en compte son contexte d'exécution courant afin de générer une spécification de structure adaptée à la situation puis il doit pouvoir reconfigurer sa structure actuelle afin de se conformer à la spécification obtenue. Ainsi, un tel composant doit être structurellement et dynamiquement adaptable. De plus, il doit disposer de mécanismes pour l'automatisation des étapes de prises de décisions.

4.4.2.1 Les composants

Les composants de notre modèle de composants structurellement et dynamiquement auto-adaptatifs sont de deux types : des composants métiers (*i.e.* composants fournissant les services fonctionnels de l'application le contenant) issus de la fragmentation du composant initial et des composants non-fonctionnels (*i.e.* composants fournissant des services transversaux à toute application) tels que les composants de gestion de contexte ou d'adaptation.

Les composants métiers font référence aux composants primitifs de notre modèle de composants structurellement et dynamiquement adaptables. L'auto-adaptation structurelle d'un composant logiciel va donc consister à générer automatiquement des nouveaux composants par encapsulation de ses sous-composants existants et à redéployer les composants générés en fonction de son contexte d'utilisation. De ce fait, plus le composant aura une architecture fragmentée, plus les possibilités de reconfiguration seront grandes. Ainsi, par défaut, nous nous basons sur le modèle canonique prévoyant la réification des interfaces en composants métiers.

Par ailleurs, un composant structurellement auto-adaptable doit être doté de fonctionnalités lui permettant de réaliser automatiquement l'adaptation. Pour cela, nous avons introduit trois nouveaux sous-composants dans son architecture initiale (voir Figure 4.12) :

- *un composant de gestion de contexte (GC)*
  D'une part, un composant auto-adaptatif doit être capable d'acquérir des informations sur sa structure et son comportement. Les informations sur la structure du composant peuvent être obtenues

par l'intermédiaire d'interfaces spécifiques (non-fonctionnelles) permettant de réaliser de l'introspection sur le composant. Concernant les informations comportementales, elles peuvent être obtenues par l'intermédiaire de connecteurs (CC) entre les composants métier (CM), chargés de collecter des informations sur les messages échangés entre composants tels que le nombre d'appels d'un service, etc.
Par ailleurs, il doit être capable de décrire les conditions d'utilisation de ses services. Ainsi, il doit être doté d'une interface appelée service de description (SD) connectée au gestionnaire de contexte, et permettant de décrire les services fournis par le composant. Chaque description doit contenir deux types d'informations : d'une part, la liste des ressources requises par le service pour fonctionner (*i.e.* contexte de déploiement) et d'autre part, les données relatives au profil de son utilisateur potentiel ainsi que les conditions d'utilisation (*i.e.* contexte d'exploitation). Les informations sur le contexte de déploiement doivent être fournies par les concepteurs de services alors que celles liées au contexte d'exploitation sont spécifiques à l'application et donc doivent être renseignées par l'administrateur ou le concepteur de l'application. Ces informations sont représentées sous la forme de données (voir Figure 4.9).

```
1  <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2  <!DOCTYPE componentStructure SYSTEM "/ScorpioData/xmldtds/context.dtd">
3
4   <context name="Context1">
5     <resource>
6         <system>MAC OS</system>  <cpu>450</cpu>
7     </resource>
8     <user>
9         <work>Engineer</work>  <level>2</level>
10    </user>
11    <condition>
12        <position>
13            <longitude>0.05377152</longitude> <latitude>0.879408752</latitude>
14            <altitude>200</altitude> <area>100</area>
15        </position>
16    </condition>
17  </context>
```

Figure 4.9 – Exemple de description de services

D'autre part, le composant doit être capable d'acquérir des informations sur son contexte externe (*i.e.* contexte de déploiement et contexte d'exploitation). L'acquisition de ce type d'information est réalisée par l'intermédiaire de capteurs pouvant être matériels (GPS, etc.) ou logiciels (indicateur du niveau de batterie, etc.).
Les données ainsi recueillies doivent être interprétées, agrégées puis transmises au composant décisionnel ;

- *un composant décisionnel (CD)*
  Ce composant fournit des mécanismes de décisions permettant au composant adapté de déterminer une spécification de sa structure (voir Figure 4.10) qui soit adaptée au contexte courant. La spécification est en fait une description de la structure interne du composant résultat de l'adaptation en termes de sous-composants à générer : pour chaque sous-composant, doivent être spécifiés les primitifs qu'ils contiennent ainsi que leur site de déploiement.

```
1  <?xml version="1.0" encoding="iso-8859-1" ?>
2  <!-- An XML DTD for Scorpio : adaptationScript.dtd -->
3  <!-- Component adaptation script -->
4
5  <!ELEMENT componentStructure (component*)>
6  <!ELEMENT component (component*, u-component*)>
7  <!ELEMENT u-component>
8  <!--
9  component : nouveau composant à générer par fragmentation
10 u-component : composant primitif dont le nom a été défini lors
11       de la génération du composant au format canonique
12 --!>
13 <!ATTLIST componentStructure
14  name CDATA #REQUIRED
15 >
16 <!ATTLIST component
17  name CDATA #REQUIRED
18  host CDATA #IMPLIED
19 >
20 <!ATTLIST u-component
21  name CDATA #REQUIRED
22 >
```

Figure 4.10 – Spécification de l'adaptation de composant logiciel

Exemple de l'agenda-partagé
Pour illustrer cette étape, considérons l'exemple du composant *Agenda-partagé*. Nous supposons que ce composant a été déployé sur un seul site. Imaginons que l'administrateur du composant souhaite, pour des raisons de répartition des charges, distribuer ce composant de la manière suivante : (1) les services *Agenda* et *MiseAjourAgenda* seront fournis par un composant appelé *GestionnaireDAgenda* et déployé sur *site1*, (2) les services *Réunion* et *MiseAjourReunion* seront fournis par un composant appelé *GestionnaireDeReunion* et déployé sur *site2* et enfin (3) les services *Absence*, *MiseAjourAbsence*, *Droit* et *MiseAjourDroit* seront déployés sur un composant appelé *GestionnaireDAbsence* et déployé sur *site3*. De ce fait, les composants primitifs *Agenda* et *MiseAjourAgenda* doivent être encapsulés dans un composant appelé *GestionnaireDAgenda* et déployé sur *site1*, etc. La spécification de l'adaptation permettant d'obtenir cette nouvelle architecture du composant *Agenda-partagé* est donné dans la figure 4.11.
Pour plus de détails sur la génération automatique de cette spécification, nous invitons le lecteur à consulter la section 4.5.2 ;

- *un composant d'adaptation (CA)*
  Une fois que la spécification du résultat de l'adaptation a été générée, le composant doit automatiquement reconfigurer sa structure et redéployer certains de ses sous-composants si nécessaire. Ces opérations sont réalisées respectivement par le composant de reconfiguration et le composant de redéploiement. Par ailleurs, le composant de redéploiement exige que chaque « composant interface » soit doté d'une interface de gestion d'état permettant de sauvegarder et de charger l'état d'un composant[2]. Nous considérons l'état d'un composant logiciel comme l'ensemble des états des ressources logicielles qu'il utilise.

[2]Pour plus de détails sur les mécanismes de transfert d'état existants, voir [126].

```
1  <?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
2  <!DOCTYPE componentStructure SYSTEM "/ScorpioData/xmldtds/adaptationScript.dtd">
3
4  <componentStructure name="Agenda-partage">
5   <component name="GestionnaireDAgenda" host="site1">
6      <u-component name="Agenda"/>
7      <u-component name="MiseAjourAgenda"/>
8    </component>
9   <component name="GestionnaireDeReunion" host="site2">
10     <u-component name="Reunion"/>
11     <u-component name="MiseAjourReunion"/>
12  </component>
13  <component name="GestionnaireDAbsence" host="site3">
14     <u-component name="Absence"/>
15     <u-component name="MiseAjourAbsence"/>
16  </component>
17 </componentStructure>
```

Figure 4.11 – Script d'adaptation du composant *Agenda-partagé*

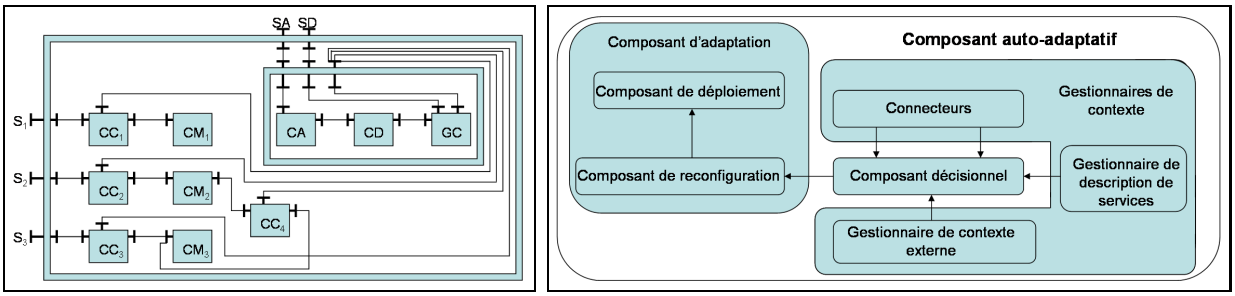

Figure 4.12 – Architecture d'un composant auto-adaptatif

4.4.2.2 Les connecteurs

Les connecteurs mis en jeu dans le cadre de notre modèle de composants dynamiquement et structurellement adaptables sont les connecteurs définis dans notre modèle statique (voir Section 3.3) auxquels nous avons introduit deux nouveaux types de connecteurs permettant de réaliser l'auto-adaptation dynamique :

- *les connecteurs d'acquisition contextuelle*
  Ces connecteurs sont utilisés pour acquérir des informations sur le comportement du composant. Ces données vont être stockées sous la forme d'un historique des communications échangées entre les composants. Les informations recueillies sont par exemple le nombre de fois qu'un service est appelé par un autre service, la taille des paramètres échangés, etc. Elles vont être transmises au gestionnaire de contexte qui va les interpréter afin de générer une spécification de la structure du composant prenant en compte son comportement ;

- *les connecteurs de redéploiement*
  Ces nouveaux connecteurs, appelés *connecteurs de redéploiement* sont intégrés aux connecteurs horizontaux ainsi qu'aux connecteurs verticaux. Ils ont pour rôle de gérer le redéploiement de composants logiciels destinés à être exécutés sur des sites distants de l'infrastructure distribuée. Ainsi, ces connecteurs sont chargés (1) d'intercepter les messages entrant dans le composant que

l'on veut redéployer sur un site distant, durant la durée d'indisponibilité de ce composant, (2) de réaliser le transfert d'état entre les deux instances du composant en question et enfin (3) d'agir sur les connecteurs de communications afin qu'ils transforment les appels locaux au composant transféré en appels distants vers le nouveau composant déployé.

4.5 Processus d'auto-adaptation structurelle dynamique

Le processus d'auto-adaptation du composant consiste tout d'abord à déclencher une phase d'adaptation, puis à déterminer une stratégie d'adaptation en fonction du contexte courant et enfin à réaliser cette adaptation par reconfiguration de la structure interne du composant. Toutes ces étapes doivent être entièrement automatiques.

4.5.1 Déclenchement automatique d'une phase d'adaptation

Une phase d'adaptation structurelle peut être déclenchée de deux manières : soit par l'administrateur de l'application, soit par l'application elle-même qui détecte une évolution soudaine de son contexte d'exécution engendrant des conséquences sur son fonctionnement.

Le déclenchement automatique d'une phase d'adaptation impose la mise en place préalable de stratégies d'adaptation en fonction du contexte. Ces stratégies sont généralement établies au moment de la conception de l'application ou pendant un processus de ré-ingénierie de l'application. Elles résident dans la création de règles de type ECA (évènement, condition, actions) qui vont en fonction des évènements produits lors de l'exécution de l'application, déclencher l'adaptation structurelle.

```
When <event> if <condition> do <adaptation structurelle>
```

Par exemple, si un service du composant est invoqué et que la mémoire disponible sur la machine de déploiement de l'application ne permet pas de l'exécuter, une adaptation de la structure du composant est déclenchée. Cette adaptation peut entraîner le redéploiement en fonction du contexte ; ceci afin de libérer de l'espace mémoire pour que le service puisse être exécuté.

```
When componentService.isCalled() if (AvailableMemory<1024) then structAdapt.start();
```

Dans le cas où l'administrateur déclenche lui-même une phase d'adaptation, deux cas de figure peuvent être envisagés :

Soit il est uniquement l'instigateur de l'adaptation et le choix stratégie d'adaptation est déterminé automatiquement. Ce cas de figure se produit dès lors qu'une situation nécessitant l'adaptation et n'ayant pas été prise en compte automatiquement apparaît (mauvaise gestion des règles de déclenchement de l'adaptation).

Soit il est l'instigateur et le superviseur de l'adaptation en spécifiant manuellement la structure du composant à obtenir. Ce cas de figure peut être envisagé lorsqu'il estime que l'application ne peut pas anticiper des évènements qui vont se produire dans le futur. Par exemple, si une partie de l'application est déployée sur une machine qui doit être déconnectée de l'infrastructure pendant une durée inconnue pour des raisons de maintenance, l'administrateur de l'application peut prendre la décision de restructurer l'application afin de la déployer sur d'autres nœuds de l'infrastructure (anticipation de déconnexion de matériels).

4.5.2 Spécification automatique de la nouvelle structure du composant

Une fois l'adaptation déclenchée, le composant doit déterminer une structure adaptée à son contexte d'exécution. La génération d'une telle structure doit être réalisée en fonction de la stratégie adoptée.

Pour générer une spécification de la nouvelle structure du composant adaptée à son contexte d'utilisation, nous avons identifié trois types de tâche permettant de classifier ses services de manière à obtenir une partition de services pour laquelle chaque élément est associé à une machine de déploiement différente. Ces tâches diffèrent selon la catégorie de contexte mise en jeu et leurs impacts sur la structure du composant. Ces tâches sont les suivantes :

1. *classification des services en fonction de leur priorité de déploiement sur la machine de l'utilisateur (prise en compte du contexte d'exploitation)*
   Tout d'abord, le contexte d'exploitation est utilisé pour classifier les services fournis par le composant à adapter en fonction de leur priorité de déploiement sur la machine de l'utilisateur de l'application. L'indice de priorité d'un service est calculé en fonction des points de similarité entre son contexte cible (défini par l'administrateur de l'application au travers des interfaces de description de services) et son contexte d'exploitation (acquis par l'intermédiaire de capteurs logiciels ou matériels). Plus le nombre de points de similarité est important, plus le niveau priorité du service est grand. Ces points de similarités sont établis par l'intermédiaire de règles de type $(< condition > \Rightarrow < action >)$. Un exemple de règle est donné ci-dessous :

   $$(\quad if(userRequiredLocation(S_i) = getuserLocation()) \quad then \quad incrementPriority(S_i) \quad)$$

   Ces règles doivent être définies par l'administrateur de l'application car elles ne peuvent être générées de manière automatique du fait des spécificités des évènements et des décisions à l'application ;

2. *association des services à des sites de déploiement potentiels (prise en compte du contexte de déploiement)*
   La seconde tâche consiste à sélectionner les machines susceptibles de déployer les services du composant à adapter, en tenant compte des ressources matérielles disponibles sur les différents nœuds de l'infrastructure distribuée (*i.e.* contexte de déploiement) ainsi que des ressources matérielles requises par les services pour permettre leur déploiement et leur assurer une continuité et une qualité de service. Cette tâche consiste à comparer les données relatives aux ressources requises par les services fournis par le composant (données définies par le concepteur du composant au travers des interfaces de description de services) avec les données contextuelles fournies par les capteurs déployés sur chaque machine de l'infrastructure distribuée. Étant donné que nous avons pour objectif de maximiser le nombre de services de forte priorité (définie lors de la tâche précédente) déployés sur la machine de l'utilisateur ainsi que sur son voisinage, cette tâche doit associer à chaque site de déploiement un ensemble de services qui pourront y être déployés tout en garantissant la continuité de service. Cependant, cette tâche de sélection des sites de déploiement pour les différents services fournis par le composant ne peut être entièrement automatisée du fait de la spécificité des ressources requises par chaque service. De ce fait, la sélection est dirigée par l'intermédiaire de règles définies par le concepteur du composant. Un exemple de règle de sélection est donné ci-dessous :

   $$(\quad if(requiredCPU(S_i) < providedCPU(Site_j)) \quad then \quad associate(S_i, Site_j) \quad)$$

3. *association des services à leur site de déploiement en fonction des dépendances entre les services (prise en compte du contexte applicatif)*
   La dernière tâche permettant d'obtenir une partition des services adaptée au contexte consiste à répartir les services en fonction de leur comportement et de l'architecture de l'application (*i.e.* contexte applicatif). Cette répartition a pour objectif d'optimiser la distribution des services en évaluant leurs dépendances. En fait, cette optimisation consiste à regrouper les services les plus dépendants au sein de sous-composants déployés sur une même machine, en tenant compte des dépendances avec les autres composants de l'application. Contrairement aux deux précédentes tâches, pour lesquelles un traitement spécifique non automatisable était requis, cette tâche peut être automatisée.

4.5.3 Auto-Spécification de la structure à générer en fonction des dépendances entre les services du composant

Cette règle a pour objectif de minimiser le surcoût lié aux communications entre les sous-composants générés. Elle consiste à regrouper les services les plus dépendants au sein de composants déployés sur une même machine. Comme nous l'avons vu précédemment, les dépendances entre les services sont de deux types : d'une part, les dépendances fonctionnelles (appels de services, etc.) et d'autre part, les dépendances liées au partage de ressources (mise en œuvre de sections critiques, etc.). Ces deux types de dépendances peuvent nécessiter la mise en œuvre de communications distantes ; ce qui peut se révéler très coûteux dans certains cas. Il est donc indispensable de sélectionner judicieusement les services de chaque composant à générer de manière à minimiser le nombre de communications entre les composants distants.

4.5.3.1 Éléments évaluables du contexte

Pour mettre en place un tel système de sélection, il est nécessaire d'évaluer quantitativement les dépendances entre les composants. Pour cela, nous devons conserver un historique des communications entre différents composants interfaces du composant conforme au format canonique : des mécanismes vont être mis en place au niveau des connecteurs entre composants afin de conserver des informations sur les communications échangées. Les données à conserver pour chaque service S_i du composant à adapter sont les suivantes :

- la probabilité que le service S_i appelle un autre service S_j tel que $S_j \in S_{fourni} \cup S_{requis}$, notée $P_{use}(S_i, S_j)$: nombre de fois où S_j est appelé lors de l'exécution de S_i (appel direct ou indirect) par rapport au nombre d'appels de S_i,

- le nombre moyen d'appels de S_i à un autre service S_j tel que $S_j \in S_{fourni} \cup S_{requis}$, noté $M_{moy}(S_i, S_j)$: nombre moyen de fois où S_j est appelé lors de l'exécution de S_i (appel direct ou indirect),

- le nombre moyen de paramètres utilisés lors de l'appel d'un service S_j par le service S_i, noté $Nb_{param}(S_i, S_j)$ ainsi que leur taille mémoire moyenne (en octets), notée $T_{param}(S_i, S_j)$,

- la probabilité de mise à jour d'une ressource dans S_i partagée avec un autre service S_j tel que $S_j \in S_{fourni}$, notées $P_{update}(S_i, S_j)$,

- la probabilité de déclencher une section critique dans S_i relative à une ressource partagée avec un autre service S_j tel que $S_j \in S_{fourni}$, notées $P_{critical}(S_i, S_j)$.

4.5.3.2 Évaluation des dépendances entre les services

Les données contextuelles ainsi récoltées sont utilisées pour établir des matrices de proximité entre les différents services fournis par le composant à adapter. La proximité entre deux services dépend de leur couplage qui représente l'évaluation des dépendances fonctionnelles et de leur cohésion qui représente l'évaluation des dépendances liées au partage de ressources.

Le couplage (équation 4.1) entre deux services différents S_i et S_j noté $C_{couplage}(S_i, S_j)$ est évalué en fonction du nombre probable d'appels du service S_j au cours de l'exécution de S_i et inversement (équation 4.2) pondéré par le nombre et le type de paramètres échangés entre ces deux services (équation 4.3). En fait, cette pondération est calculée en fonction du nombre moyen de paramètres utilisés pour l'appel du service et de la taille mémoire moyenne de ces paramètres. Ainsi, nous évaluons le couplage entre deux services S_i et S_j de la manière suivante :

$$C_{couplage}(S_i, S_j) = \alpha(S_i, S_j) * \beta(S_i, S_j) + \alpha(S_j, S_i) * \beta(S_i, S_j) \quad (4.1)$$

$$Tel \quad que \quad \alpha(x, y) = T_{param}(x, y) * (Nb_{param}(x, y) + 1) \quad (4.2)$$

$$\beta(x, y) = M_{moy}(x, y) * P_{use}(x, y) \quad (4.3)$$

La cohésion (équation 4.4) entre deux services différents S_i et S_j, notée $C_{cohesion}(S_i, S_j)$ est déterminée en fonction du nombre de sections critiques déclenchées dans chaque service (équation 4.5) et de la fréquence de mise à jour de ressources partagées de S_i vers S_j et inversement (équation 4.6), pondérée par le nombre et le type de ressources (équation 4.7). La pondération par rapport au type de ressources correspond à leur taille moyenne exprimée en octets et notée $T_{rp}(S_i, S_j)$. Ainsi, la valeur de la cohésion entre deux services S_i et S_j est obtenue comme suit :

$$C_{cohesion}(S_i, S_j) = \chi(S_i, S_j) + \gamma(S_i, S_j) * \eta(S_i, S_j) \quad (4.4)$$

$$Tel \quad que \quad \chi(x, y) = P_{critical}(x, y) + P_{critical}(y, x) \quad (4.5)$$

$$\eta(x, y) = P_{update}(x, y) + P_{update}(y, x) \quad (4.6)$$

$$\gamma(x, y) = Nb_{rp}(x, y) * T_{rp}(x, y) \quad (4.7)$$

Nous définissons la proximité entre deux services S_i et S_j dans un ensemble S comme une relation binaire, notée $Pr(S_i, S_j)$, exprimée de la manière suivante :

$$Pr(S_i, S_j) = \begin{cases} 1 & \text{si } S_i = S_j \\ \frac{1}{\alpha+\beta}(\alpha * C'_{couplage}(S_i, S_j) + \beta * C'_{cohesion}(S_i, S_j)) & \text{sinon} \end{cases}$$

Tel que

$$C'_{couplage}(S_i, S_j) = \begin{cases} 0 & \text{si } C_{couplage\_max} = 0 \\ \frac{C_{couplage}(S_i, S_j)}{C_{couplage\_max}} & \text{sinon} \end{cases}$$

$$C'_{cohesion}(S_i, S_j) = \begin{cases} 0 & \text{si } C_{cohesion\_max} = 0 \\ \frac{C_{cohesion}(S_i, S_j)}{C_{cohesion\_max}} & \text{sinon} \end{cases}$$

$$C_{couplage\_max} = max(\{C_{couplage}(x, y), \quad \forall x, y \in S \quad / \quad x \neq y\})$$

$$C_{cohesion\_max} = max(\{C_{cohesion}(x, y), \quad \forall x, y \in S \quad / \quad x \neq y\})$$

La proximité entre deux services varie de zéro, quand les deux services ne sont liés par aucune dépendance, à un. La valeur « un » signifie que les deux services sont identiques (par convention). Plus

la valeur de la proximité est proche de un, plus les deux services sont dépendants. α et β sont les facteurs d'impact respectivement du couplage et de la cohésion. L'administrateur peut, en fonction des besoins, faire varier ces facteurs d'impact, en donnant plus de poids au couplage ($\alpha > \beta$) ou bien à la cohésion ($\alpha < \beta$). Par défaut, on suppose que ces deux dépendances ont des impacts identiques ($\alpha = 1$ et $\beta = 1$).

4.5.3.3 Algorithme de regroupement de services

L'évaluation de la proximité entre les services fournis par le composant à adapter va permettre de les regrouper en sous-ensembles contenant ceux qui sont les plus dépendants (proximité élevée) les uns des autres ; chaque sous-ensemble constituera, par la suite, un composant dont les interfaces fournies sont celles contenues dans ce sous-ensemble.

En fait, pour obtenir une partition des interfaces fournies par le composant à adapter, pour laquelle chaque élément de la partition est associé à un site de déploiement, nous utilisons un algorithme de « *clustering* » hiérarchique (voir Figure 4.13) que nous adaptons afin qu'il réponde à nos attentes. Celui-ci prend en entrée une matrice contenant l'évaluation de la proximité entre les services fournis par le composant à adapter et avec les sites de déploiement de l'application. Chaque site représente le *cluster* correspondant à l'ensemble des services fournis par les composants qui y sont déployés et requis par le composant à adapter (*i.e. cluster* de site).

L'algorithme consiste à regrouper les services fournis par le composant à adapter, dans des *clusters*, en fonction de leur proximité, et à associer à chaque *cluster* un site de déploiement unique. Initialement, chaque ensemble de services fournis par le composant initial est placé dans un *cluster* (*i.e. cluster* de service). On calcule alors la matrice de proximité entre les différents *clusters* créés ainsi qu'avec les *clusters* représentant les sites de déploiement. Dans un premier temps, la valeur maximale de cette matrice est recherchée (valeur correspondant aux *clusters* les plus proches). Deux cas de figure peuvent se présenter : si cette valeur correspond à la proximité entre deux *clusters* contenant des services, ces *clusters* seront alors regroupés en un même *cluster*. Si cette valeur correspond à la proximité entre un *cluster* de service et un site de déploiement alors le *cluster* de service sera associé au site en question. Dans le cas où le *cluster* est déjà associé à un site de déploiement, cette valeur est ignorée et l'algorithme recherche la valeur maximale de cette matrice, exceptées celles déjà trouvées. Une fois que la valeur maximale est traitée, la matrice de proximité doit être recalculée en fonction du nouveau *cluster* créé. Deux solutions sont possibles pour calculer la proximité entre deux *clusters* : soit tous les calculs de proximité (couplage et cohésion) sont réalisés à nouveau en fonction des services contenus dans chaque *cluster*. Cette solution ne peut être envisagée du fait de sa complexité qui ne peut être acceptable lors d'une adaptation dynamique ; soit une approximation de la valeur de proximité entre deux *clusters* est réalisée. Nous avons choisi une stratégie qui consiste à conserver la moyenne des proximités entre les éléments du nouveau *cluster* créé :

$$Pr(C_{l_1}, C_{l_2}) = \frac{1}{|C_{l_1}||C_{l_2}|} \sum_{S_i \in C_{l_1}, S_j \in C_{l_2}} Pr(S_i, S_j)$$

L'algorithme est alors réitéré jusqu'à ce que chaque *cluster* soit associé à un site. Le résultat correspond alors aux différents *clusters* obtenus et à leur site associé.

4.5.3.4 Exemple de regroupement de services

Considérons l'application suivante : trois composants appelés C_1, C_2 et C_3 sont assemblés (voir Figure 4.14). C_1 fournit les ensembles de services S_1, S_2, S_3 et S_4 et requiert les ensembles de services S_5 et S_6 pour fonctionner ; chaque ensemble de services correspond à une interface du composant. C_1

$Cl_{init} \leftarrow Ensemble \quad des \quad clusters \quad initiaux$
Tant que $\exists Cl_i \in Clinit$ tel que $Cl_i \cap Sites \neq \oslash$ Faire
$\forall Cl_i \in Cl_{init}, \forall Cl_j \in Cl_{init} \cup Sites, T[Cl_i, Cl_j] \leftarrow Pr(Cl_i, Cl_j)$
Déterminer Cl_{maxi} et Cl_{maxj} tels que
$\forall Cl_i \in Cl_{init}, \forall Cl_j \in Cl_{init} \cup Sites, \quad Cl_i \neq Cl_j$
$T(Cl_{maxi}, Cl_{maxj}) \geq T(Cl_i, Cl_j)$ et $|Cl_{maxi} \cap Site| \leq 1$
$Cl_{init} = Cl_{init} \cup \{(Cl_{maxi}, Cl_{maxj})\}$
Si $Cl_{maxj} \in Cl_{init}$ alors
$Cl_{init} = Cl_{init} - \{Cl_{maxi}, Cl_{maxj}\}$
Si $Cl_{maxj} \in Sites$ alors
$Cl_{init} = Cl_{init} - \{Cl_{maxi}\}$
$Sites = Sites - \{Cl_{maxj}\}$
Retourner Cl_{init}

Figure 4.13 – Algorithme de regroupement des services en *clusters* associés à des sites

est déployé sur le site 1. Le composant C_2 qui fournit l'ensemble de services S_5 (requis par C_1) est également déployé sur le site 1. L'ensemble de services S_6 requis par C_1 est fourni par le composant C_3 qui est déployé sur le site 2.

Dès lors que l'adaptation est déclenchée par le composant C_1, la proximité entre les *clusters* de services ((S_1),(S_2),(S_3),(S_4)) et avec les *clusters* de sites ($Site_1 = \{S_5\}$, $Site_2 = \{S_6\}$) doit être évaluée. Nous obtenons les résultats suivants :

Pr	(S_1)	(S_2)	(S_3)	(S_4)	$Site_1 = \{S_5\}$	$Site_2 = \{S_6\}$
(S_1)	1	0.1203	**0.4972**	0	0.059	0
(S_2)	–	1	0.0409	0.1622	0.0218	0
(S_3)	–	–	1	0.0727	0	0.3454
(S_4)	–	–	–	1	0.1909	0

Première itération : nous observons que la valeur de la proximité maximale entre les *clusters*, correspond à la proximité entre deux *clusters* de services (S_1) et (S_3). Ainsi, ces deux *clusters* sont regroupés et la matrice de proximité est alors réévaluée.

Pr	(S_1, S_3)	(S_2)	(S_4)	$Site_1$	$Site_2$
(S_1, S_3)	1	0.0806	0.0363	0.0295	0.1727
(S_2)	–	1	0.1622	0.0218	0
(S_4)	–	–	1	**0.1909**	0

Deuxième itération : dans ce cas, nous observons que la valeur de la proximité maximale entre les *clusters*, correspond à la proximité entre un *cluster* de service $(S_4$ et un *cluster* de site ($Site_1$). Ainsi, le *cluster* (S_4) est associé au site 1.

Pr	(S_1, S_3)	(S_2)	(S_4)	$Site_2$
(S_1, S_3)	1	0.0806	0.0363	**0.1727**
(S_2)	–	1	0.1622	0
$(S_4), Site_1$	–	–	1	0

Troisième itération : le *cluster* (S_1, S_3) est associé au site 2.

Pr	(S_1, S_3)	(S_2)	(S_4)
$(S_1, S_3), Site_2$	1	0.0806	0.0363
(S_2)	–	1	**0.1622**
$(S_4), Site_1$	–	–	1

Quatrième itération : les deux *clusters* de services (S_2) et (S_4) sont regroupés dans un nouveau *cluster* associé au site 1 car (S_4) est déjà associé à ce site.

Pr	(S_1, S_3)	(S_2, S_4)
$(S_1, S_3), Site_2$	1	0.0585
$(S_2, S_4), Site_1$	–	1

Étant donné que tous les *clusters* de service sont associés à un site, l'algorithme se termine. Nous obtenons ainsi deux *clusters* : l'un contient les ensembles de services S_1 et S_3 et est associé au site 2 ; l'autre contient les ensembles de services S_2 et S_4 et est associé au site 1. Ainsi, le partitionnement du composant C_1 devra être réalisé en générant deux nouveaux composants appelés C_1' et C_1''. C_1' fournira les ensembles de services S_1 et S_3 et sera déployé sur le site 2. Et, C_1'' fournira les ensembles de services S_2 et S_4 et sera déployé sur le site 1. La spécification de l'adaptation ainsi obtenue sera alors utilisée par le processus d'adaptation structurelle dynamique pour reconfigurer la structure du composant afin de l'adapter à son contexte d'exécution (voir Figure 4.14).

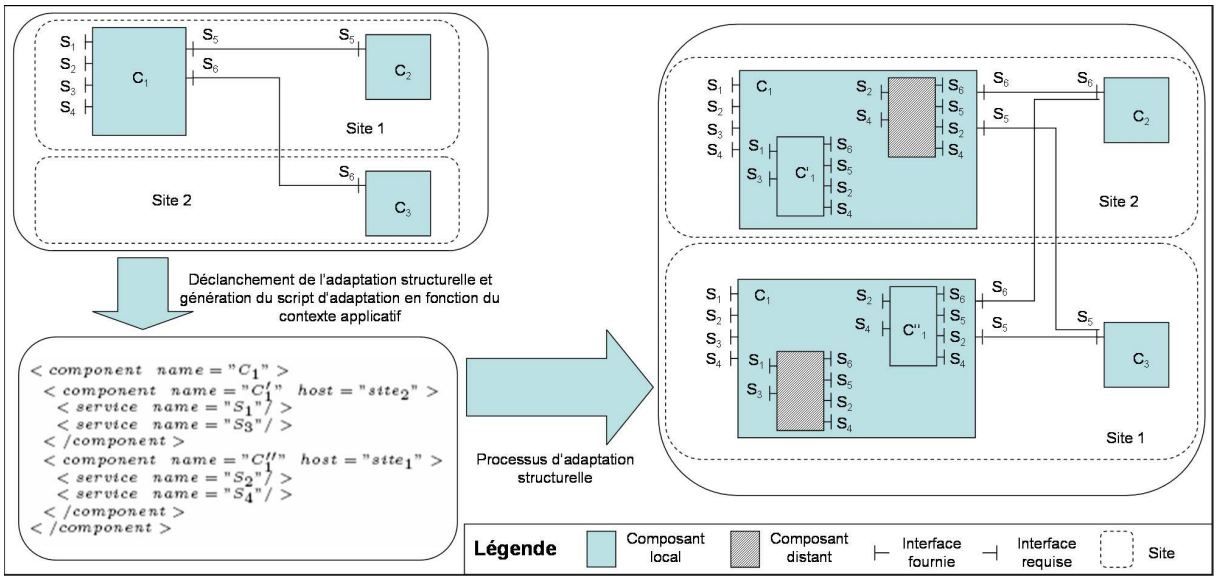

Figure 4.14 – Regroupement des services en fonction des dépendances existantes

4.5.4 Reconfiguration dynamique du composant structurellement adaptable

Le composant à adapter pour satisfaire les attentes liées à son utilisation, se trouve dans un format canonique au moment de son exécution. La génération de la nouvelle structure est ainsi obtenue à partir de ce format. Il s'agit de construire, à la volée, de nouveaux composants à partir des composants interfaces existants. Chaque nouveau composant deviendra ainsi un composant composite dont les sous-composants des composants primitifs. Chaque nouveau composant composite doit donc encapsuler toutes les interfaces fournies et requises par ses sous-composants. Ces derniers ne seront plus visibles et ils seront accessibles uniquement par les interfaces du composite les contenant. Ainsi, chaque composite fournit l'ensemble des services offerts par ses sous-composants. De plus, il définit comme interfaces requises,

celles requises par ses sous-composants, exceptées celles fournies par d'autres de ses sous-composants. Les nouveaux composants générés seront, par la suite, manipulés comme des unités de déploiement à part entière. Ainsi, il est nécessaire de définir une opération d'encapsulation de composants primitifs pour réaliser la reconfiguration.

Cependant, l'encapsulation n'est pas toujours suffisante pour obtenir la structure souhaitée. En effet, si une phase de reconfiguration a déjà été opérée sur le composant à adapter, certains composants primitifs peuvent être encapsulés dans un composite. Or, si la nouvelle spécification requiert l'encapsulation de plusieurs composants déjà encapsulés dans des composites différents, il est nécessaire de casser ces composites afin de procéder à leur encapsulation dans un composite répondant à la spécification. De ce fait, il est indispensable de définir une opération d'éclatement des composites issu des opérations d'encapsulation menées dans le cadre de la reconfiguration du composant.

La reconfiguration, à la volée, du composant sous format canonique nécessite donc la définition de deux nouvelles opérations de reconfiguration. Ces opérations sont les suivantes :

1. *l'encapsulation de composants primitifs*
   Cette opération consiste à encapsuler plusieurs « composants primitifs » dans un même composant appelé « composant généré » fournissant les services de l'ensemble des composants qu'il contient. Le nouveau composant ainsi créé a le même comportement que l'assemblage des « composants primitifs » qu'il contient ; de ce fait, l'intégrité du composant adapté est préservée. Cette opération de reconfiguration est réalisée de la manière suivante (voir Figure 4.16) :

   (a) *extraction des composants à encapsuler*
   Tout d'abord, (1) l'assemblage de « composants primitifs » qui va être encapsulé dans un composant composite généré à la volée doit être extrait du composite initial. Pour cela, les liaisons entre les interfaces (fournies/fournies, fournies/requises et requises/requises) doivent être détruites, exceptées celles associant deux interfaces de « composants primitifs » contenus dans l'assemblage isolé ;

   (b) *encapsulation dans un composite*
   Puis, (2) le composite encapsulant les « composants primitifs » est créé. Les interfaces fournies sont celles fournies par les composants qu'il encapsule alors que les interfaces requises sont celles requises par les composants qu'il encapsule, exceptées celles qu'il fournit ;

   (c) *reconfiguration des connexions internes*
   Ensuite, (3) l'assemblage à encapsuler est connecté au composite généré : les interfaces fournies par le composite sont connectées aux interfaces fournies par les « composants primitifs » au travers de liaisons d'exportation. Les interfaces requises des « composants primitifs » et des sous-composites sont également connectées. Si une interface requise par un « composant primitif » est fournie par un autre « composant primitif » de l'assemblage à encapsuler alors la connexion est directe. Dans le cas contraire, il faut créer une liaison d'exportation entre cette interface requise et l'interface requise du composite généré correspondante ;

   (d) *reconfiguration des connexions externes*
   Enfin, le composite généré est connecté au composant adapté (connexion à la membrane du composite et aux autres sous-composants) : les interfaces fournies du composant composite

généré sont connectées à celles du composant adapté au travers de liaisons d'exportation ainsi qu'aux interfaces requises d'autres sous-composants (du composite adapté) qui étaient connectées directement aux interfaces fournies des « composants primitifs » encapsulés. Les interfaces fournies du composant composite généré sont quant à elles connectées aux interfaces fournies correspondantes ou bien avec une interface requise du composant composite. Dans le premier cas, l'interface est fournie par un autre sous-composite. Il faut donc établir une connexion directe entre ces deux interfaces. Dans le second cas, le service est fourni par un autre composant de l'application. Un lien d'exportation doit donc être mis en place entre l'interface requise du sous-composite et celle du composite correspondante.

La figure 4.15 montre un exemple de reconfiguration de la structure d'un composant C par encapsulation de ses sous-composants. Les « composants primitifs » C_1 et C_2 fournissant respectivement les services S_1 et S_2 sont encapsulés dans un composant C_5 qui pourra alors être redéployé sur un autre site. De la même manière, les composants C_3 et C_4 fournissant respectivement les services S_3 et S_4 sont encapsulés dans un composant C_6.

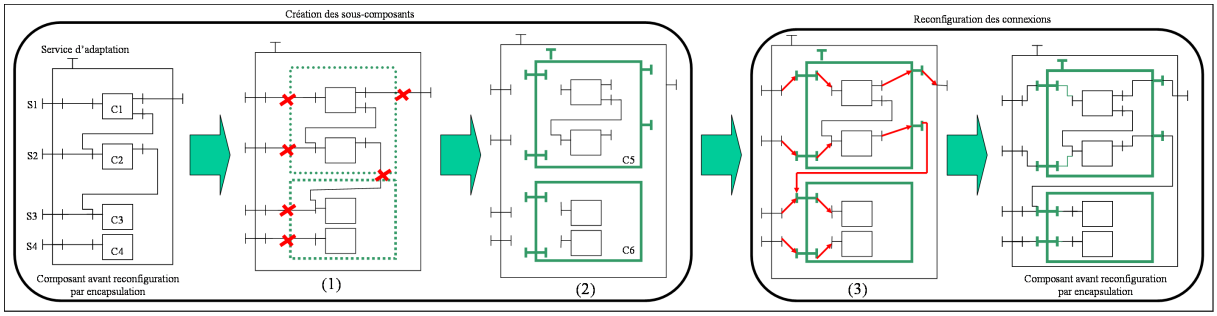

Figure 4.15 – Opération d'encapsulation de « composants primitifs »

2. *l'éclatement d'un composant composite généré par encapsulation*
   Cette opération consiste à éclater un composant composite généré lors de l'encapsulation de « composants primitifs ». Dans ce cas, le composite est remplacé par l'assemblage de « composants primitifs » qu'il contient. Cet assemblage fournit les mêmes services que le composite car le composite ne fournit pas de services qui lui sont propres ; de ce fait, l'intégrité du composant adapté est préservée. Cette opération de reconfiguration est réalisée de la manière suivante (voir Figure 4.16) :

   (a) *destruction des connexions*
   Tout d'abord, (1) le composant composite à éclater est déconnecté du composite le contenant (*i.e.* destruction des liaisons d'exportation relatives aux interfaces fournies et requises entre les deux composants) ainsi que des autres sous-composants qui lui sont liés (au travers une interface requise) ;

   (b) *destruction de la membrane*
   Ensuite, (2) la membrane du composant à éclater est détruite et l'assemblage de composants la contenant est extraite ;

(c) *reconfiguration des connexions*
Puis, (3) les composants issus de l'éclatement du composant sont assemblés avec les autres sous-composants ainsi qu'avec le composite le contenant : d'une part, les interfaces fournies par les sous-composants sont connectées avec les interfaces fournies du composite correspondantes (au travers de liaisons d'exportation) ; concernant les interfaces requises de l'assemblage de composants extraits, si une interface requise est fournie par un autre sous-composant alors les deux interfaces seront liées. Dans le cas contraire, l'interface requise de l'assemblage sera liée à l'interface requise du composite au travers une liaison d'exportation. Une fois que toutes les interfaces qui ont été déconnectées sont à nouveau liées à d'autres interfaces, l'opération d'éclatement se termine.

La figure 4.16 montre un exemple de reconfiguration de la structure d'un composant C par éclatement de sous-composites générés par encapsulation de « composants primitifs ». Dans cet exemple, les composants C_5 et C_6 sont éclatés de manière à obtenir la structure initiale du composant.

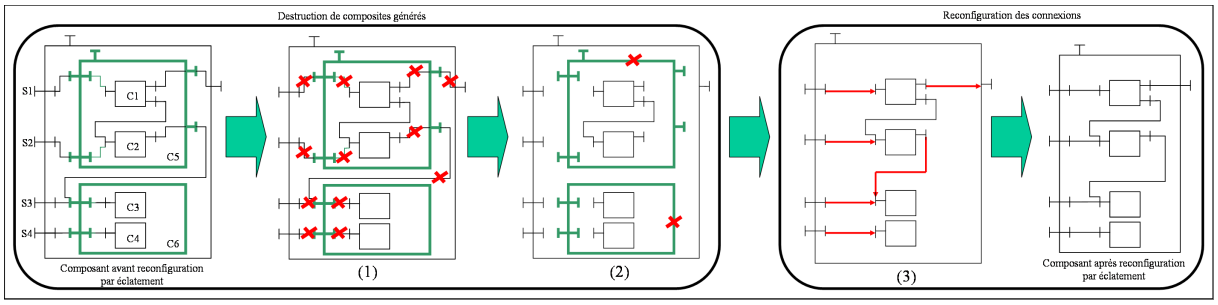

Figure 4.16 – Opération d'éclatement de « composants générés »

Ces deux opérations sont utilisées pour reconfigurer la structure d'un composant conforme au modèle canonique que nous avons défini précédemment, en fonction du la spécification de l'adaptation fournie par l'administrateur de l'application. Pour cela, plusieurs stratégies peuvent être envisagées :

- *stratégie de reconfiguration par la destruction*
  La première stratégie consiste à détruire tous les composites qui ont été générés. Puis, à reconstruire par encapsulation la nouvelle structure spécifiée. Cette stratégie n'est pas optimale en terme d'opérations de reconfiguration à appliquer sur le composant à adapter.

- *stratégie de reconfiguration par la comparaison*
  La deuxième stratégie consiste à comparer la structure actuelle du composant à celle souhaitée au travers de la spécification fournie et à déterminer le delta entre ces deux structures. L'algorithme de reconfiguration consiste alors à détruire un minimum de composites qui ont été créés lors d'adaptations précédentes en fonction de la nouvelle spécification (voir Figure 4.17). Les composants obtenus vont alors servir de briques de base à la création de la nouvelle structure. Ensuite, des opérations d'encapsulation sont réalisées afin de rendre la structure du composant adapté, conforme à la nouvelle spécification. Cette stratégie permet de cibler les opérations de reconfiguration à réaliser.

$S = C_{1..n}$ //sous-composants du composite avant son adaptation
$S' = C'_{1..m}$ //sous-composants à générer par reconfiguration
$i = 1$
$j = 1$
Tant que $|S'| \neq \oslash$ Faire
  Si structureExterne(C_i)= structureExterne(C'_j) alors
    $S' = S' - \left\{C'_j\right\}$
    $i = i + 1$
    $j = 0$
  Sinon
    Si structureExterne(C_i)$\bigcap$ structureExterne(C'_j)$\neq \oslash$ alors
      éclater(C_i)
      éclater(C_j)
      $j = 0$
    Sinon
      $j = j + 1$
retourner S

Figure 4.17 – Algorithme de génération des briques de base servant à la reconfiguration

4.5.5 Redéploiement dynamique du résultat de la reconfiguration

Le redéploiement dynamique consiste à déployer sur des sites distants les sous-composants générés lors de la reconfiguration du composant dynamiquement adaptable, tout en préservant leur consistance et leur intégrité.

4.5.5.1 Processus de redéploiement d'un composant

Le processus de redéploiement d'un composant est le suivant : dans un premier temps, une copie du composant à redéployer doit être envoyée sur le site distant sans arrêter le composant local. Puis, les invocations de services fournis par le composant à transférer sont interceptées par un composant de connexion préalablement introduit. Lorsque le composant à redéployer se trouve dans un état stable, celui-ci est arrêté et son état est transféré au composant distant. Lorsque le composant déployé sur le site distant est démarré, les messages mis en attente durant l'arrêt du composant y sont alors transmis (voir Figure 4.18). Le déploiement d'un composant sur un site distant nécessite la présence d'un composant spécifique appelé composant serveur, chargé de déployer automatiquement les composants qui lui sont transmis.

Plusieurs cas de figures peuvent se présenter lors du redéploiement d'un sous-composant généré par encapsulation ou par éclatement de sous-composants existants :

Cas 1 : le composite (*i.e.* composant que l'administrateur veut adapter) n'est pas présent sur le futur site de déploiement du nouveau composant généré.
Dans ce cas, une copie du composant sous format canonique va être déployée sur le site en question. Cette copie va alors être reconfigurée de manière à obtenir la structure du composant souhaitée (*i.e.* structure conforme à la spécification fournie par l'administrateur sans considération des sites de déploiement). Les composants virtuels correspondant aux composants primitifs contenus dans le nouveau composant à déployer sur le site seront alors remplacés par des composants locaux associés. Puis, les transferts d'état entre les composants locaux distants seront alors réalisés. Ensuite, les composants locaux transférés seront alors remplacés par des composants virtuels. Enfin, la nouvelle copie du composant adaptée est démarrée.

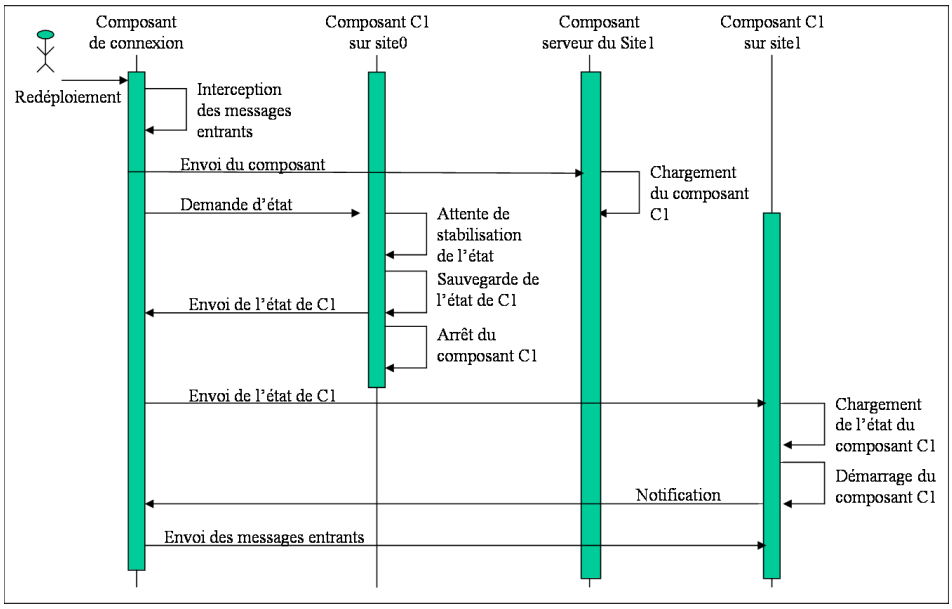

Figure 4.18 – Redéploiement dynamique de composants logiciels

Nous pouvons noter que les composants locaux peuvent provenir de différentes copies du composite (*i.e.* copies déployées sur des sites différents).

Cas 2 : le composite est déjà distribué.
Dans ce cas, les modifications de la structure du composant sous format canonique doivent être propagées sur chaque copie déployée sur des sites distants, de manière à prendre en compte l'adaptation.

Cas 3 : une copie du composite (voir Section 3.2.4.2) est déjà déployée le futur site de déploiement du nouveau composant généré.
Dans ce cas, la structure de la copie distante du composite est alors reconfigurée. Puis, les composants locaux correspondant aux composants primitifs des nouveaux composants générés sont alors transférés (*i.e.* remplacement des composants virtuels en question de la copie distante par des composants locaux) sur leur nouveau site de déploiement et remplacés par des composants virtuels.

Cas 4 : une partie du composant à transférer est déjà déployée sur le nouveau site.
Dans ce cas, seuls les composants locaux qui ne sont pas encore déployés sur le nouveau site doivent être transférés après reconfiguration de la structure.

Cas 5 : après transfert du composant généré, la copie du composite adapté ne contient plus de composants locaux.
Deux cas de figures se présentent alors : soit le site en question correspond au site de déploiement initial du composant. Dans ce cas, la copie du composite contenant uniquement des composants virtuels est conservée ; dans le cas contraire (*i.e.* le site en question ne correspond pas au site de

déploiement initial du composant), la copie du composite est détruite après le transfert de tous les composants locaux.

4.5.5.2 Problèmes à gérer liés au déploiement dynamique

Pour assurer ce redéploiement dynamique des sous-composants issus de l'adaptation, nous dotons les composants de mécanismes permettant d'une part, de stabiliser l'état de ses sous-composants et d'autre part, de réaliser des transferts d'état de sous-composants locaux vers des sous-composants distants identiques.

Mécanismes de stabilisation de l'état d'un composant Nous considérons qu'un composant est dans un état « stable » lorsqu'aucun des services qu'il fournit n'est en cours d'exécution et qu'aucune des ressources logicielles qu'il définit n'est en cours d'utilisation. De ce fait, pour qu'un composant soit dans un état stable, il suffit d'intercepter les éventuelles invocations de ses services ainsi que les demandes d'accès à une ressource logicielle qu'il définit.

Ces opérations d'interceptions de messages sont réalisées par l'intermédiaire de connecteurs ou de composants spéciaux de connexion entre les composants générés de manière à garantir l'intégrité du composite. Leur rôle est :

1. d'intercepter les messages entrant dans le composant que l'on veut redéployer (*i.e.* composant cible), durant la durée d'indisponibilité de ce composant,

2. de stocker temporairement ces messages jusqu'à ce que le composant cible soit à nouveau en activité,

3. de redéployer le composant cible sur le site distant,

4. de réaliser le transfert d'état entre les deux copies du composant,

5. de transférer les messages temporairement stockés vers le nouveau composant distant,

6. et enfin de modifier les connecteurs de communications entre les composants distants de manière à ce que les nouveaux messages entrants soient redirigés vers le composant distant tout en respectant l'ordre d'arrivée des messages (*i.e.* synchronisation entre les messages stockés temporairement dans les connecteurs ou les composants de connexion et les messages transmis après réactivation du composant distant).

Dès lors que tous les services en cours d'exécution et les accès à des ressources logicielles se terminent, nous considérons que le composant se trouve dans un état stable. Cette stratégie suppose que l'exécution d'un service et l'accès à une ressource logicielle se terminent en un temps fini. Ceci exclut donc les *threads* qui pourraient être traités en sauvegardant directement la pile d'exécution de l'application et en la rechargeant sur le site distant [116].

Mécanismes de transfert d'état de composants Pour réaliser le transfert d'état entre deux instances d'un même composant, nous utilisons les interfaces de sauvegarde et de chargement d'état d'un composant que nous avons définies dans notre modèle d'adaptation structurelle dynamique.

Une fois que l'état de l'instance du composant à déplacer sur une machine distante est stabilisé, il est sauvegardé (*i.e.* sauvegarde des états des ressources logicielles définies dans le composant) sous la forme d'un fichier XML contenant des couples ressource/valeur :

$< nom\_de\_la\_ressource > Valeur < /nom\_de\_la\_ressource >$

Le fichier contenant la sauvegarde de l'état du composant est alors envoyé sur le site contenant la nouvelle instance du composant à déployer. Ce fichier est alors chargé sur la nouvelle instance du composant déployé sur la machine distante par l'intermédiaire des interfaces de chargement d'état. Ainsi, l'état du composant déplacé sur une machine distante est transféré.

Nous pouvons noter que cette solution a été choisie pour gérer le transfert d'état de ressources relativement simples (attributs de type primitifs). Dans la littérature, beaucoup d'approches ont traité les problèmes inhérents aux transferts d'états entre plusieurs composants. La résolution de ces problèmes n'étant pas au centre de cette thèse, pour plus de détails sur les solutions qui ont été apportées dans les approches existantes, nous invitons le lecteur à consulter [126].

4.6 De l'auto-adaptation de composants vers l'auto-adaptation d'architectures à base de composants

Dans les sections précédentes, nous avons étudié l'adaptation au niveau des composants logiciels (niveau micro). Or, il parait évident que l'adaptation d'un composant peut avoir des impacts sur les autres composants de l'application qui le contient. De ce fait, afin de réaliser l'adaptation d'une application, il est indispensable de concevoir une stratégie de gestion de l'adaptation au niveau macro (*i.e.* niveau global à l'application).

Comme nous l'avons expliqué précédemment, notre approche permet d'adapter automatiquement un composant à son contexte d'exécution. Or, dans le domaine des composants logiciels, une application est conçue par l'assemblage de composants logiciels existants. De ce fait, adapter un composant n'est pas suffisant pour adapter une application conçue à base de composants. En effet, l'adaptation d'un composant d'une application peut avoir des impacts sur les autres composants qui la constituent. Par exemple, si deux composants d'une même application sont déployés sur une machine à ressources limitées et que lors de leur exécution, les ressources disponibles ne sont pas suffisantes pour assurer leur continuité de service, les deux composants vont déclencher automatiquement une phase d'adaptation structurelle. Si ces deux adaptations ne sont pas coordonnées par une entité logicielle centralisée, il peut se produire la situation suivante : la majorité des services fournis par chaque composant est distribuée sur l'infrastructure disponible alors que les ressources disponibles après l'adaptation des deux composants auraient pu permettre le déploiement de services (sur la machine de l'utilisateur) qui ont été distribués. Ainsi, pour adapter une application il est nécessaire d'introduire des mécanismes permettant de coordonner l'adaptation des différents composants qu'elle contient.

Pour assurer la coordination de l'adaptation des composants d'une application, nous déployons, sur chaque site, un composant non fonctionnel dédié appelé composant de coordination. Ce composant doit être chargé de déclencher une phase d'adaptation s'il détecte que les ressources disponibles sur son site de déploiement sont insuffisantes pour garantir la continuité de service des composants fonctionnels qui y sont déployés. Puis, il doit être en mesure de déclencher des adaptations sur des composants fonctionnels

déployés sur son site et d'en contrôler les impacts. De ce fait, un composant de coordination doit intégrer des mécanismes lui permettant d'acquérir des informations sur son contexte d'exécution ainsi que des mécanismes de prise de décisions lui permettant de déclencher des adaptations et de coordonner les adaptations qu'il aura déclenchées.

Ainsi, un composant de coordination est un composant composite qui contient deux sous-composants (voir Figure 4.19) :

1. *un composant de gestion de contexte*
   le premier sous-composant, appelé gestionnaire de contexte (CM) offre des mécanismes permettant d'acquérir, de modéliser et d'analyser le contexte d'exécution de la partie d'application le contenant. Chaque partie d'application fait référence à l'ensemble des composants d'une application déployés sur un seul site. L'union de ces parties constitue l'application ;

2. *un composant de prise de décision*
   Le second sous-composant est appelé composant décisionnel (DM). Il est chargé de réaliser le processus de coordination des adaptations des composants de la partie d'application le contenant. Son comportement est détaillé ci-après.

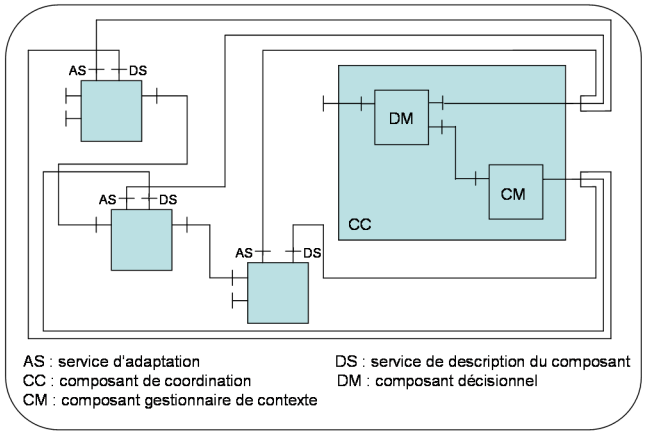

Figure 4.19 – Architecture d'une application auto-adaptable

Les composants de coordination ont deux rôles : leur premier rôle est de déclencher des phases d'adaptation de par l'analyse du contexte d'exécution de l'application. Ainsi, cette action qui était jusqu'à présent réalisée par les composants eux-mêmes est déléguée aux composants de coordination. Cette stratégie permet de centraliser et ainsi mieux contrôler les déclenchements de phases d'adaptation. Elle permet également de gagner en performances ; car un seul composant, sur chaque site d'implantation de l'application est chargé d'acquérir des informations sur le contexte pour déclencher l'adaptation. Le deuxième rôle de ces composants est de coordonner l'adaptation des composants présents dans l'application. Pour cela, deux stratégies de coordination peuvent être mises en place (voir Figure 4.20) : la première stratégie consiste à utiliser les composants de coordination pour déclencher, en fonction du contexte, des phases d'auto-adaptation au niveau des composants déployés sur une même machine (*i.e.* stratégie d'adaptation au niveau composant) ; la seconde stratégie consiste quant à elle à utiliser les composants de coordinations pour déclencher une phase d'adaptation et à déterminer automatiquement,

en fonction du contexte, les nouvelles structures des composants déployés sur une même machine (*i.e.* stratégie d'adaptation au niveau service).

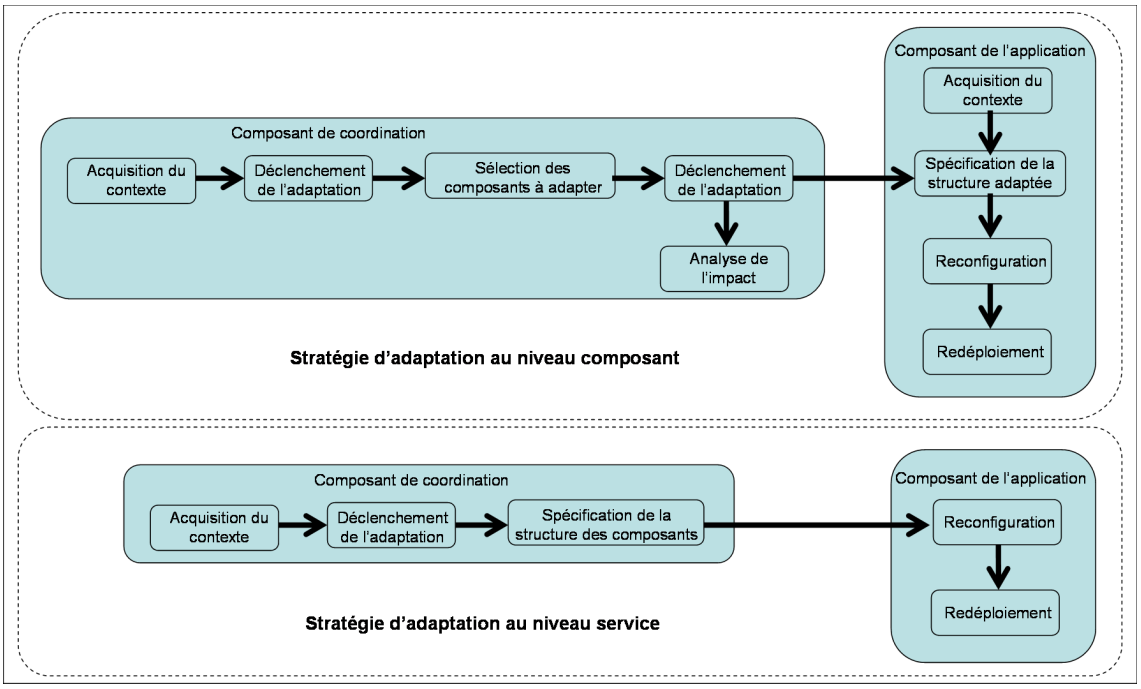

Figure 4.20 – Les différentes stratégies de coordination de l'adaptation au niveau macro

4.6.1 Stratégie d'adaptation au niveau composant

4.6.1.1 Présentation

La première stratégie (gros grain au niveau des composants mais impact local au niveau de l'application) consiste à sélectionner les composants devant être adaptés, à analyser l'impact de leur adaptation sur les autres composants et à déclencher d'autres adaptations si besoin est. Pour cela, il est nécessaire d'établir une classification des composants en fonction de leur intérêt à être adapté au contexte d'exécution. L'intérêt d'un composant à être adapté dépend de l'impact de son adaptation sur la continuité de service. Cependant, l'assurance de cette continuité de service ne doit pas être réalisée au détriment de la qualité de service. Ainsi, il est indispensable de prendre en compte le surcoût lié à l'adaptation. Par exemple, il doit être privilégié l'adaptation de composants peu utilisés, car leur distribution ne va pas entraîner de surcoûts visibles (liés aux communications distantes très coûteuses) du point de vue de l'utilisateur. Dans ce cas, les composants de coordination doivent être capables de capter et d'analyser le contexte d'exécution des composants sur leur site de déploiement et ainsi d'en extraire des informations sur le besoin d'adaptation de l'application et sur les impacts de chaque adaptation afin de déclencher des phases d'adaptation sur les composants les moins utilisés.

Afin de mettre en place cette stratégie d'auto-adaptation d'une application, les composants logiciels assemblés doivent être conformes au modèle de composants auto-adaptatifs défini précédemment. En effet, chaque composant doit disposer de mécanismes lui permettant d'acquérir son contexte d'exécution,

de déterminer une nouvelle structure adaptée au contexte courant et de reconfigurer dynamiquement sa structure. Cependant, le déclenchement d'un processus d'adaptation n'est pas réalisé par le composant lui même mais par le composant de coordination déployé sur la machine. Les mécanismes d'acquisition et de modélisation du contexte d'exécution sont partagés entre les composants fonctionnels et le composant de coordination ; chacun disposant d'une vue partielle sur le contexte. En fait, le composant de coordination dispose d'une vue sur le contexte de déploiement et chaque composant fonctionnel offre une vue sur son propre contexte d'exploitation et son contexte applicatif. Lors des étapes de prise de décisions ces vues doivent être agrégées afin d'obtenir une vue globale du contexte permettant de prendre en compte tous les éléments.

4.6.1.2 Processus de coordination

Le processus de coordination de l'adaptation au niveau macro comporte quatre étapes :

1. *détection des besoins d'adaptation par observation de l'environnement (contexte d'exécution) de l'application*
   Cette étape du processus nécessite l'acquisition et l'analyse du contexte. Dans ce cas, le contexte fait référence au contexte d'exécution des composants logiciels et plus particulièrement aux ressources évolutives du contexte de déploiement ; mais également au contexte d'utilisation des composants (*i.e.* ressources requises par les composants pour garantir une continuité de service) fournis par les interfaces de description de service.
   L'application doit être capable d'assurer la continuité des services qu'elle propose. Pour cela, des mécanismes de déclenchement automatique d'adaptation doivent être mis en place : lorsque les ressources évolutives deviennent critiques (*i.e.* inférieures à un seuil calculé en fonction des besoins de chaque service ou bien défini par le concepteur de l'application), l'adaptation structurelle doit être déclenchée.

2. *sélection des composants à adapter en fonction de leur taux d'utilisation*
   La deuxième étape consiste à déterminer en fonction du contexte extrait lors de la première étape, quels sont les composants de l'application qui doivent être adaptés pour garantir une continuité de service tout en préservant la qualité de service. Différentes stratégies peuvent être adoptées en fonction des spécificités de l'application et des besoins liés à son utilisation. Par exemple, une stratégie peut consister à adapter, en priorité, les composants les moins utilisés en les distribuant sur l'infrastructure disponible à condition que leur distribution soit susceptible de libérer des ressources manquantes pour assurer la continuité de service. Par exemple, si le composant ne consomme pas d'espace disque et que la ressource manquante au site de déploiement est l'espace disque, alors l'adaptation de ce composant sera ignorée. Ainsi, le choix des composants à adapter se fait en fonction du taux d'utilisation de leurs services fournis et de l'impact de leur adaptation.
   Dans un premier temps, il est nécessaire d'établir une classification (ordre total) des composants déployés sur un site, en fonction de leur taux d'utilisation. Cette classification va permettre de déterminer les composants à adapter en priorité. Trois solutions sont possibles pour déterminer le taux d'utilisation.

   (a) Soit le taux d'utilisation d'un composant C_k noté $T_{Use}(C_k)$ est calculé en fonction de la moyenne arithmétique d'appels de services à un composant, notée M_{Use}. Cette stratégie

permet d'adapter, en priorité, les services les plus utilisés. De plus, dans l'objectif de garantir une qualité de service, les composants dont la répartition d'utilisation des services est la plus déséquilibrée doivent être adaptés en premier.

Par exemple, si l'on considère deux composants C_1 fournissant trois services S_1, S_2 et S_3 ; et C_4 fournissant également trois services S_4, S_5 et S_6 : à l'instant t du déclenchement de l'adaptation, on observe les valeurs présentées dans le tableau 4.1. On note $\alpha(S_i)$ le nombre d'appels du service S_i. Nous pouvons remarquer que les deux composants ont des moyennes d'utilisation identiques ($M_{Use}(C_1) = M_{Use}(C_4) = 50$). Cependant, la répartition des charges au niveau des services n'est pas la même : les services S_2 et S_3 sont très peu utilisés alors que tous les services du composant C_4 sont utilisés avec une répartition des charges équilibrées. De ce fait, il parait judicieux de privilégier l'adaptation du composant C_1 par rapport au composant C_4 car la distribution des services S_2 et S_3 pourrait permettre de libérer des ressources sur le site sans avoir un impact notable pour l'utilisateur (*i.e.* pas de surcoût engendré par la distribution au niveau des services les plus utilisés).

Ainsi, la moyenne arithmétique $M_{Use}(C_k)$ doit être pondérée par le nombre de services dont le nombre d'appels dépasse un seuil fixé par l'administrateur de l'application. Par défaut, nous fixons ce seuil à la moyenne arithmétique d'appels de services au composant ($M_{Use}(C_k)$). De cette manière, dans le cas où deux composants auraient des moyennes arithmétiques proches, nous privilégions l'adaptation de composants ayant la répartition d'utilisation de ses services la moins équilibrée des deux. Cette solution accorde plus d'importance aux services les plus utilisés.

Le taux d'utilisation d'un composant est alors calculé de la manière suivante :

$$T_{Use}(C_k) = \left|\{S_j \in S_{C_k} \quad / \quad \alpha(S_j) \geq M_{Use}(C_k)\}\right| * M_{Use}(C_k)$$

Tel que

$$M_{Use}(C_k) = \frac{1}{|S_{C_k}|} \sum_{j=0}^{|S_{C_k}|} \alpha(S_j)$$

(b) Soit le taux d'utilisation d'un composant C_k noté $T'_{Use}(C_k)$ est calculé en fonction de la moyenne harmonique du nombre d'appels de ses services. Cette stratégie permet d'obtenir des moyennes très faibles lors de répartitions de charges non équilibrées entre les services fournis par un composant. Ainsi, les composants les moins utilisés ne sont pas forcément ceux qui auront des taux d'utilisation faibles mais plutôt ceux dont la répartition des charges est la plus déséquilibrée.

Dans ce cas, le taux d'utilisation d'un composant est donc calculé de la manière suivante :

$$T'_{Use}(C_k) = M'_{Use}(C_k)$$

Tel que

$$M'_{Use}(C_k) = \frac{|S_{C_k}|}{\sum_{j=0}^{|S_{C_k}|} \frac{1}{\alpha(S_j)+1}}$$

(c) La dernière solution consiste à associer les deux stratégies de calcul précédentes en évaluant le taux d'utilisation d'un composant C_k noté $T''_{Use}(C_k)$ à la moyenne arithmétique de la moyenne harmonique $M'_{Use}(C_k)$ et la moyenne arithmétique $M_{Use}(C_k)$. Cette stratégie permet à la fois de tenir compte du déséquilibre de la répartition des charges d'un composant et de son utilisation globale.

Dans ce cas, le taux d'utilisation d'un composant est donc calculé de la manière suivante :

$$T''_{Use}(C_k) = \frac{1}{2}\left(M_{Use}(C_k) + M'_{Use}(C_k)\right)$$

Tel que

$$M_{Use}(C_k) = \frac{1}{|S_{C_k}|}\sum_{j=0}^{|S_{C_k}|}(\alpha(S_j)+1) \quad et \quad M'_{Use}(C_k) = \frac{|S_{C_k}|}{\sum_{j=0}^{|S_{C_k}|}\frac{1}{\alpha(S_j)+1}}$$

En appliquant un algorithme de tri classique sur ces valeurs, nous obtenons une classification des composants potentiellement adaptables en fonction du taux d'utilisation choisi (voir Tableau 4.1). La stratégie d'adaptation consiste alors à déclencher l'adaptation structurelle sur les composants ayant le taux d'utilisation le plus bas. Le nombre de composant à adapter dépend de la consommation en ressources de chaque composant. De ce fait, il faut, après avoir classifié les composants par leur taux d'utilisation, sélectionner les composants de telle sorte que les ressources susceptibles d'être libérées par l'adaptation soient supérieures à celles requises pour garantir une continuité de service de l'application sur le site ;

Composant	$\alpha(S_k)$	$T_{Use}(C_k)$	Classement	$T'_{Use}(C_k)$	Classement	$T''_{Use}(C_k)$	Classement
C_1	140 - 5 - 5	50	2	7.3684	2	28.6842	3
C_2	12 - 35 - 45	61.3333	3	22.3668	5	26.5167	1
C_3	20 - 158 - 3 - 12	48.25	1	7.6989	3	27.9744	2
C_4	55 - 45 - 50	100	6	49.6655	6	49.8327	5
C_5	300 - 2 - 5 - 15 - 26	69.6	4	4.2654	1	36.9327	4
C_6	300 - 4 - 52 - 3	89.75	5	11.0065	4	71.2532	6
C_7	100 - 156 - 150 - 120	263	7	130	7	65	7

Table 4.1 – Exemple de classification de composants en fonction de leur taux d'utilisation

3. *déclenchement du processus d'adaptation structurelle automatique sur les composants sélectionnés précédemment*
   Une fois que les composants à adapter sont sélectionnés, le composant de coordination du site concerné va déclencher les processus d'auto-adaptation structurelle de ces derniers. Cette opération est réalisée par l'invocation du service d'adaptation des composants concernés ;

4. *analyse de l'impact de l'adaptation structurelle*
   Après l'exécution du processus d'adaptation structurelle sur les composants sélectionnés lors de la deuxième étape, deux cas de figures peuvent apparaître : soit la nouvelle structure engendre une

diminution des ressources consommées permettant de garantir la continuité de service de l'application. Dans ce cas, le processus repasse en mode de détection du besoin de l'adaptation structurelle (première étape) ; soit les ressources libérées par l'adaptation structurelle ne sont pas suffisantes pour assurer une continuité de service. Dans ce cas, le processus réitère les phases de sélection et d'adaptation en fonction des composants venant d'être adaptés (élargir le champ de sélection des composants) jusqu'à l'obtention d'une structure adaptée au contexte courant (*i.e.* garantissant la continuité de service).

De plus, chaque composant de coordination doit être capable de décider s'il peut ou non autoriser, en fonction des ressources disponibles, le déploiement de nouveaux composants issus de l'adaptation structurelle réalisée sur d'autres sites fournissant d'autres composants de l'application.

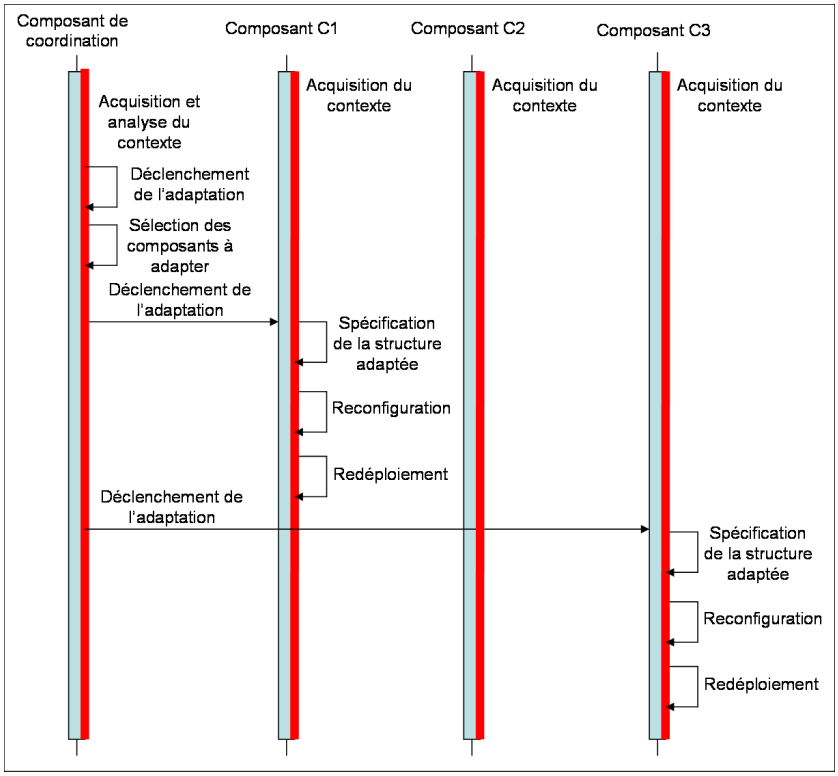

Figure 4.21 – Processus de coordination relatif à la stratégie d'adaptation au niveau composant

Un exemple de processus de coordination est fourni dans la figure 4.21. Il montre trois composants (C_1, C_2 et C_3) qui sont déployés sur un même site. A un instant donné, les ressources disponibles sur ce site ne sont plus suffisantes pour garantir la continuité de service de ces composants. Dès lors que le composant de coordination détecte ce problème, il va établir une classification des composants à adapter en fonction du taux d'utilisation de leur service : C_1 est le composant dont les services sont les moins utilisés, suivi de C_3 puis de C_2. Ainsi, le composant de coordination va déclencher l'auto-adaptation du composant C_1. Ce dernier va générer automatiquement la spécification d'une structure adapté à sa situation puis va reconfigurer sa structure courante afin de la rendre conforme à la spécification obtenue et

enfin, redéployer les sous-composants générés si nécessaire. Après analyse de l'impact de cette adaptation, le composant de coordination détecte que les ressources libérées ne sont pas suffisantes pour garantir la continuité de service des composants déployés sur le site. Ainsi, il va déclencher l'auto-adaptation du composant C_3. Une fois l'auto-adaptation réalisée, les ressources disponibles sont jugées suffisantes par le composant de coordination pour garantir la continuité de service.

4.6.1.3 Évaluation de la stratégie d'adaptation au niveau composant

Cette stratégie permet de cibler l'adaptation et ainsi minimiser le nombre de composants à adapter en déclenchant des phases d'adaptation uniquement sur les composants dont les services sont les moins utilisés par l'utilisateur. Ainsi, cette stratégie permet de minimiser le surcoût engendré par l'adaptation du point de vue de l'utilisateur.

Cependant, cette stratégie souffre des inconvénients suivants : tout d'abord, cette stratégie n'a qu'un impact local sur le fonctionnement de l'application car les adaptations sont réalisées sur des composants cibles. De ce fait, elle permet d'assurer la continuité de service de l'application mais pas d'améliorer la qualité de service au niveau global de l'application.

De plus, le choix des composants à adapter peut être sujet à discutions. Le choix de la stratégie de calcul du taux d'utilisation d'un composant reste à l'appréciation de l'administrateur de l'application.

Chaque composant doit être doté de mécanismes lui permettant d'acquérir des informations sur son contexte d'exécution et générer en fonction de ces informations une structure adaptée. Ainsi, les processus décisionnels relatifs à l'adaptation des composants sont situés au niveau des composants (*i.e.* processus décentralisé). Cette stratégie engendre donc des surcoûts importants pouvant provoquer des diminutions dans les performances de l'application. De ce fait, il ne peut être envisagé, dans certains cas, de doter tous les composants de l'application de tels mécanismes. Cependant, cette stratégie peut se révéler efficace si l'administrateur souhaite cibler les composants à adapter et ainsi augmenter la flexibilité uniquement pour certains composants judicieusement sélectionnés (les plus consommateurs en ressources, par exemple).

Par ailleurs, une itération peut ne pas suffire et, dans le pire des cas, tous les composants peuvent être adaptés les uns à la suite des autres. Dans ce cas, le temps d'adaptation de l'application peut se révéler important.

4.6.2 Stratégie d'adaptation au niveau service

4.6.2.1 Présentation

La deuxième stratégie (grains fins au niveau des composants mais impact global au niveau de l'application) consiste à appliquer les mécanismes de prise de décisions définis dans le cadre de l'auto-adaptation structurelle d'un composant logiciel en prenant pour point de départ l'ensemble des services déployés[3] sur un site.

Afin de mettre en place cette seconde stratégie d'auto-adaptation d'une application, les composants assemblés doivent être conformes au modèle de composants structurellement et dynamiquement adaptables défini précédemment. En effet, dans ce cas, les processus de prise de décisions sont centralisés au niveau des composants de coordination qui ont pour rôle d'acquérir des informations sur le contexte d'exécution courant et de déterminer pour chaque composant une structure adaptée.

[3]Uniquement les services déployés sur le site sont pris en considération ; les services accessibles par l'intermédiaire de composants distants sont exclus.

4.6.2.2 Processus de coordination

Cette stratégie consiste à considérer l'application déployée sur un site comme un composant composite encapsulant tous les composants déployés sur le site en question et fournissant les services fournis par ces composants. En appliquant notre stratégie de prise de décisions définie au niveau des services fournis par un composant sur l'ensemble des services fournis par l'application sur un site, nous obtenons une partition des services de l'application déployés sur ce site ; chaque élément de la partition étant associé à un site. Ensuite, à partir de la spécification obtenue, il suffit de générer une spécification pour chaque composant déployé sur la machine en séparant les services fournis par chaque composant. Chaque spécification sera alors transmise au composant correspondant au travers de l'interface d'adaptation structurelle dynamique. Une phase de reconfiguration de la structure interne du composant pourra alors être initiée et le composant en question sera redéployé si nécessaire.

Un exemple de processus de coordination est fourni dans la figure 4.22. Il montre trois composants (C_1, C_2 et C_3) qui sont déployés sur un même site. A un instant donné, les ressources disponibles sur ce site ne sont plus suffisantes pour garantir la continuité de service de ces composants. Dès lors que le composant de coordination détecte ce problème, il va générer une spécification de la structure de chaque composant et la transmettre au composant en question si celle-ci est différente de la structure courante (la structure du composant C_2 étant identique à l'actuelle, aucun message ne lui sera transmis). Les composants recevant un message contenant la spécification de leur nouvelle structure adaptée au contexte vont alors reconfigurer leur structure et redéployer leur sous-composant si nécessaire. Ainsi, la continuité de service sera garantie sur le site de déploiement des composants C_1, C_2 et C_3.

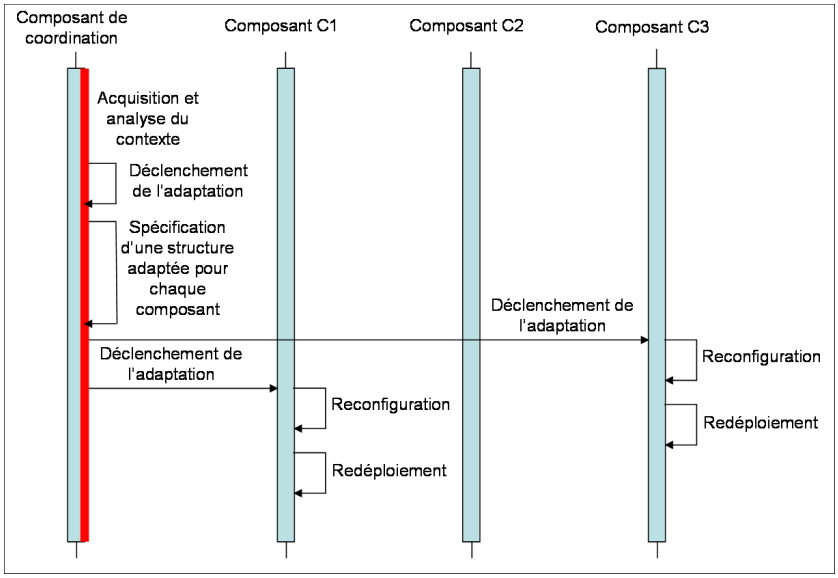

Figure 4.22 – Processus de coordination relatif à la stratégie d'adaptation au niveau service

4.6.2.3 Évaluation de la stratégie d'adaptation au niveau service

Cette stratégie à grains plus fins présente de nombreux avantages. Tout d'abord, elle permet d'obtenir une meilleure répartition des services d'un point de vue global. Dans ce cas, la répartition est plus précise et donc permet d'améliorer la qualité de service au niveau de l'application.

De plus, le surcoût engendrer par cette stratégie peut se révéler moins important que si l'on applique l'autre solution. En effet, dans ce cas, la gestion de la prise en compte du contexte est centralisée : un seul composant fournissant ce type de service (*i.e.* acquisition du contexte, déclenchement de l'adaptation et génération de la spécification correspondant à la nouvelle structure des composants déployés sur le site) est nécessaire par nœud de l'infrastructure distribuée.

Il convient également de noter qu'une seule adaptation suffit pour assurer la continuité de service de l'application sur un site de déploiement ; ce qui diminue le temps d'adaptation car les phases de reconfiguration des composants peuvent être réalisées en parallèle et non séquentiellement comme c'est le cas avec la première stratégie.

Cependant, cette stratégie présente des inconvénients. Tout d'abord, elle peut entraîner l'adaptation simultanée de tous les composants déployés sur une même machine, y compris ceux qui sont le plus utilisés. De ce fait, pendant la durée de l'adaptation les performances de l'application peuvent être dégradées de manière notable pour l'utilisateur contrairement à la première stratégie qui consiste à adapter en priorité les composants les moins utilisés.

Par ailleurs, cette stratégie ne permet pas de cibler l'adaptation et donc les composants les plus utilisés peuvent subir des adaptations. Ainsi, le surcoût lié à l'adaptation peut être perçu par l'utilisateur de l'application.

4.6.3 Bilan des deux stratégies

Dans cette section, nous avons proposé deux stratégies possibles pour gérer l'adaptation d'assemblages de composants : l'une agissant au niveau des composants et l'autre au niveau de l'assemblage. Comme nous avons pu le constater, ces deux solutions présentent chacune des avantages et des inconvénients ne permettant pas, de manière générique, de privilégier le choix de l'une ou de l'autre stratégie. Ainsi, l'administrateur de l'adaptation devra en fonction de l'application et de son contexte d'exécution, sélectionner la meilleure stratégie possible.

4.7 Ré-ingénierie de composants existants pour les rendre structurellement auto-adaptatifs

Afin de réaliser notre approche d'auto-adaptation structurelle sur des composants existants, nous fournissons un processus de ré-ingénierie permettant de transformer un composant existant en un composant structurellement auto-adaptatif. Ce processus de ré-ingénierie est basé sur la transformation structurelle que nous avons définie dans le chapitre précédent. Il comporte trois étapes :

- *spécification des « composants primitifs »*
  La première étape de notre processus de ré-ingénierie consiste à spécifier les composants qui serviront de brique de base lors de la phase dynamique. Ces sous-composants appelés « composants primitifs » fournissent chacun un élément de la partition des services fournis par le composant à

adapter. L'objectif est de regrouper les services les plus dépendants au sein de composants ne pouvant être fragmentés. Cependant, plus le nombre de « composants primitifs » est important, plus le composant est flexible car le nombre de combinaison d'assemblage possible est croissant. Par défaut, chaque interface fournie par le composant à adapter est réifiée en composant, de manière à assurer une grande flexibilité du composant.
Cette spécification peut être réalisée manuellement par un acteur externe de l'adaptation, ou bien automatiquement en prenant en compte les dépendances entre les interfaces. Nous détaillons ci-après la réalisation de cette étape de spécification ;

- *génération du composant sous format canonique*
  Une fois que les « composants primitifs » ont été spécifiés de manière automatique ou manuelle, le code source du composant doit être transformé de manière à le rendre conforme à la spécification générée précédemment. Cette transformation du composant sous format canonique est réalisée en deux sous-étapes :

  1. *génération et assemblage des « composants primitifs »*
     Afin de générer le composant composite contenant les composants primitifs (composant conforme au modèle de composants structurellement et dynamiquement adaptables), nous utilisons notre processus d'adaptation structurelle défini dans le chapitre précédent. Ce processus de transformation structurelle, réalisé de manière automatique, comporte trois étapes (voir Figure 4.23) : tout d'abord, le composant initial est fragmenté en un ensemble de composants (*i.e.* composants primitifs). Par défaut, chaque composant généré fournit un seul service du composant initial (*i.e.* composant interface). Puis, les composants primitifs sont assemblés, en tenant compte de leurs dépendances (*i.e.* assemblage horizontal). Enfin, l'assemblage obtenu est encapsulé dans un composant composite fournissant le même ensemble de services que le composant initial (*i.e.* assemblage vertical) ;

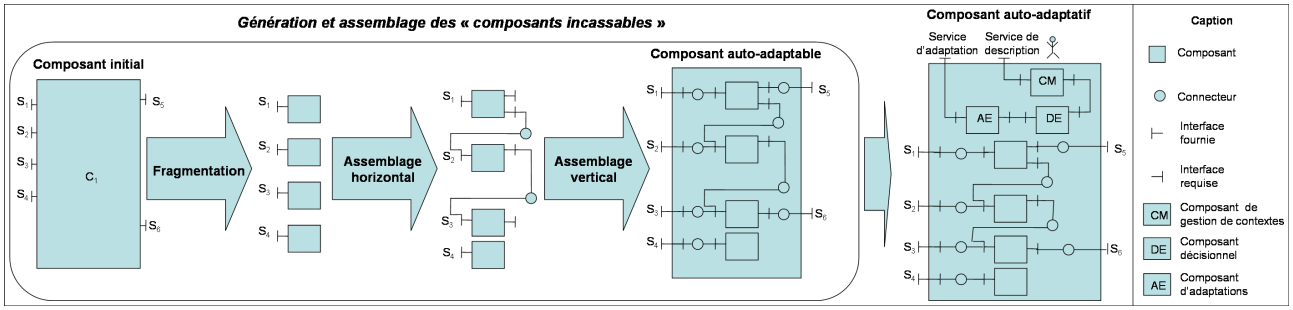

Figure 4.23 – Génération de composants dynamiquement et structurellement auto-adaptatifs

  2. *génération des composants dédiés à l'auto-adaptation*
     Une fois que le composant est conforme à notre modèle de composants structurellement et dynamiquement adaptables, il est nécessaire de lui introduire des mécanismes permettant de le rendre auto-adaptatif. Pour cela, nous devons générer les composants dédiés à l'automatisation de notre processus d'adaptation. La mise en œuvre de ces composants est détaillée dans les sections suivantes.

- *documentation du composant auto-adaptatif*
  Une fois que le composant est conforme au modèle de composants structurellement et dynamiquement auto-adaptatifs que nous avons défini précédemment, le composant doit être documenté. Cette opération consiste à décrire les ressources nécessaires au déploiement et à l'exécution de chaque service fourni par le composant concerné.
  La description peut être réalisée de manière automatique (*i.e.* détermination des ressources requises en fonction du code source du correspondant) ou bien manuellement par l'administrateur de l'adaptation. Cette description est intégrée au composant au moyen d'interfaces dédiées que nous avons décrites dans notre modèle de composants dynamiquement auto-adaptatifs. Ces données seront intégrées au gestionnaire de contexte afin d'être utilisées pour le déclenchement et la prise de décisions d'une phase d'adaptation dynamique de la structure du composant. Nous détaillerons la prise en compte de ces données dans la section suivante.

4.7.1 Spécification manuelle des « composants primitifs »

La spécification des « composants primitifs » peut être réalisée par un acteur externe de l'adaptation au travers d'une spécification (voir Figure 4.24). Cette spécification permet de définir les composants à générer et pour chaque composant, ses interfaces fournies. Un contrôle destiné à vérifier la validité de la spécification du résultat de la transformation doit être appliqué : les nouveaux composants doivent définir un ensemble d'interfaces comme étant chacun un sous-ensemble de l'ensemble des interfaces du composant initial. De plus, l'union de ces sous-ensembles doit être égale à l'ensemble des interfaces fournies par le composant initial. Également, aucun élément spécifié ne doit être en contradiction avec la description de l'architecture de l'application relative au composant initial.

Exemple de l'agenda-partagé
Pour illustrer cette étape, considérons l'exemple du composant *Agenda-partagé*. L'administrateur du composant souhaite créer six « composants primitifs » : *Agenda*, *MiseAjourAgenda*, *Reunion*, *MiseAjourReunion*, *Absence* et *MiseAjourAbsence* ; fournissant chacun l'interface de même nom. La spécification de l'adaptation permettant d'obtenir le composant *Agenda-partagé* sous format canonique est donnée dans la figure 4.25.

4.7.2 Spécification automatique des « composants primitifs » par l'évaluation des dépendances entre les services fournis

Cette étape de spécification des composants primitifs peut également être réalisée automatiquement. Son objectif est de regrouper les services fournis par le composant que l'on veut rendre conforme au modèle canonique en fonction de leurs dépendances : les services les plus dépendants doivent être fournis par un même composant primitif. Pour cela, il est nécessaire d'évaluer, de manière statique, les dépendances entre les différents services du composant adapté afin d'établir des regroupements qui vont correspondre aux services fournis par chaque composant primitif.

4.7.2.1 Évaluation des dépendances entre les services fournis

Les dépendances entre les services fournis par un composant sont de deux types : d'une part, les dépendances fonctionnelles liées aux appels entre les services (l'évaluation de ce type de dépendance

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!— An XML DTD for Scorpio : canonicalScript.dtd —>
<!— Canonical component generation —>

<!ELEMENT canonicalComponentStructure (u-component*)>
<!ELEMENT u-component (service+)>
<!ELEMENT service>

<!ATTLIST canonicalComponentStructure
 name CDATA #REQUIRED
>
<!ATTLIST u-component
 name CDATA #REQUIRED
>
<!ATTLIST service
 signature CDATA #REQUIRED
>
```

Figure 4.24 – Spécification nécessaire pour l'obtention d'un composant sous format canonique

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!DOCTYPE canonicalComponentStructure SYSTEM "/ScorpioData/xmldtds/canonicalScript.dtd">

<canonicalComponentStructure name="Agenda-partage">
  <u-component name="Agenda">
      <service signature="Agenda"/> </u-component>
  <u-component name="MiseAjourAgenda">
      <service signature="MiseAjourAgenda"/> </u-component>
  <u-component name="Reunion">
      <service signature="Reunion"> </u-component>
  <u-component name="MiseAjourReunion"/>
      <service signature="MiseAjourReunion"> </u-component>
  <u-component name="Absence">
      <service signature="Absence"/>
      <service signature="Droit"/> </u-component>
  <u-component name="MiseAjourAbsence">
      <service signature="MiseAjourAbsence"/>
      <service signature="MiseAjourDroit"/> </u-component>
</canonicalComponentStructure>
```

Figure 4.25 – Script d'obtention du composant *Agenda-partagé* sous format canonique

est appelée *couplage*) et, d'autre part, les dépendances liées aux ressources partagées entre ces services. L'évaluation de ce type de dépendance est quant à elle appelée *cohésion*.

Le couplage (4.8) est évalué au nombre de services appelés dans le code d'exécution d'un service. Une pondération pourra alors être établie en fonction de l'imbrication de ces invocations dans des blocs de conditionnelles. Plus l'invocation est située dans la profondeur notée α, plus la pondération doit être faible car la probabilité d'invocation du service est faible. Ainsi, l'évaluation du couplage, notée $C_{couplage}$, est réalisée de la manière suivante :

$$C_{couplage}(S_i, S_j) = \sum_{k=1}^{n} \frac{1}{\alpha(S_i, S_j, k)} + \sum_{k=1}^{m} \frac{1}{\alpha(S_j, S_i, k)} \quad (4.8)$$

$$Tel\ que \quad n = nombre\ d'appels\ dans\ S_i\ au\ service\ S_j \quad (4.9)$$

$$m = nombre\ d'appels\ dans\ S_j\ au\ service\ S_i \quad (4.10)$$

$$\alpha(x, y, i) = profondeur\ du\ i^{ème}\ appels\ au\ service\ y\ dans\ x \quad (4.11)$$

Concernant la cohésion (4.12), celle-ci dépend des ressources partagées durant l'exécution des différents services. Elle est évaluée au nombre de ressources partagées entre les différents services, pondéré par leur taille car plus la taille est importante, plus le coût lié au maintien de la cohérence sera important. Ainsi, l'évaluation de la cohésion, notée $C_{cohesion}$, est réalisée de la manière suivante :

$$C_{cohesion}(S_i, S_j) = \sum_{k=1}^{m} \beta(k) \tag{4.12}$$

$$Tel\ que\quad m = \quad nombre\ de\ ressources\ partagées\ entre\ S_i\ et\ S_j \tag{4.13}$$

$$\beta(k) = \quad taille\ de\ la\ \mathrm{k^{ème}}\ ressource\ partagée \tag{4.14}$$

L'évaluation de ces deux dépendances (*i.e.* couplage et cohésion) est alors utilisée pour établir des relations de proximités entre les différents services proposés par le composant à transformer sous format canonique. La proximité entre deux services varie de zéro, quand les deux services ne sont liés par aucune dépendance, à un. La valeur « un » signifie que les deux services sont identiques (par convention). Plus la valeur de la proximité est proche de un, plus les deux services sont dépendants. α et β sont des facteurs d'impact (*i.e.* pondération) respectivement du couplage et de la cohésion. L'administrateur de l'adaptation peut, en fonction des besoins, faire varier ces facteurs d'impact, en donnant plus de poids au couplage ($\alpha > \beta$) ou bien à la cohésion ($\alpha < \beta$). Par défaut, on suppose que ces deux dépendances ont des impacts identiques ($\alpha = 1$ et $\beta = 1$). Ainsi, la proximité entre deux services S_i et S_j, notée $Pr(S_i, S_j)$, est calculée de la manière suivante :

$$Pr(S_i, S_j) = \begin{cases} 1 & \text{si } S_i = S_j \\ \frac{1}{\alpha+\beta}(\alpha * C'_{couplage}(S_i, S_j) + \beta * C'_{cohesion}(S_i, S_j)) & \text{sinon} \end{cases}$$

Tel que

$$C'_{couplage}(S_i, S_j) = \begin{cases} 0 & \text{si } C_{couplage\_max} = 0 \\ \frac{C_{couplage}(S_i, S_j)}{C_{couplage\_max}} & \text{sinon} \end{cases}$$

$$C'_{cohesion}(S_i, S_j) = \begin{cases} 0 & \text{si } C_{cohesion\_max} = 0 \\ \frac{C_{cohesion}(S_i, S_j)}{C_{cohesion\_max}} & \text{sinon} \end{cases}$$

$$C_{couplage\_max} = max(\{C_{couplage}(x, y), \quad \forall x, y \in S \quad / \quad x \neq y\})$$

$$C_{cohesion\_max} = max(\{C_{cohesion}(x, y), \quad \forall x, y \in S \quad / \quad x \neq y\})$$

4.7.2.2 Algorithme de regroupement de services

Ensuite, l'application d'un algorithme de regroupement (en anglais « *clustering* ») (voir Figure 4.26) sur les évaluations des proximités entre les services fournis par le composant va permettre d'obtenir des regroupements correspondant aux composants primitifs à générer.

Le déroulement de cet algorithme est le suivant : tout d'abord, les services fournis par le composant sont répartis en n *clusters* (un service par *cluster*). Puis, l'algorithme recherche les deux *clusters* les plus proches (ceux dont la proximité est maximale) et les agrège au sein d'un même *cluster*. Cette opération est réitérée jusqu'à ce qu'il ne reste qu'un seul *cluster*. Enfin, le résultat est représenté sous forme d'un dendrogramme.

L'algorithme se termine en un nombre fini d'itérations (n) correspondant au nombre de services fournis par le composant initial. En effet, à chaque itération, le nombre de *clusters* est réduit de un du fait de la fusion des *clusters* les plus proches, jusqu'à ce que le nombre de *cluster* soit égal à un.

Le dendrogramme ainsi obtenu est alors découpé afin d'obtenir les *clusters* correspondant aux services des composants primitifs à générer. Avant de procéder au découpage, l'administrateur de l'application doit fixer un seuil de proximité appelé α_{Pr} au delà duquel les services ne peuvent pas être regroupés.

$Cl \leftarrow Ensemble \quad des \quad services \quad fournis$
Tant que $|Cl| \neq 1$ Faire
$\forall Cl_i, Cl_j \in Cl, T[Cl_i, Cl_j] \longleftarrow Pr(Cl_i, Cl_j)$
Déterminer Cl_{maxi} et Cl_{maxj} tel que
$\forall Cl_i, Cl_j \in Cl, Cl_i \neg Cl_j$
$T[Cl_{maxi}, Cl_{maxj}] \geq T[Cl_i, Cl_j]$
$Cl = Cl \cup \{(Cl_{maxi}, Cl_{maxj})\}$
$Cl = Cl - \{Cl_{maxi}, Cl_{maxj}\}$

Figure 4.26 – Algorithme de regroupement des services en clusters

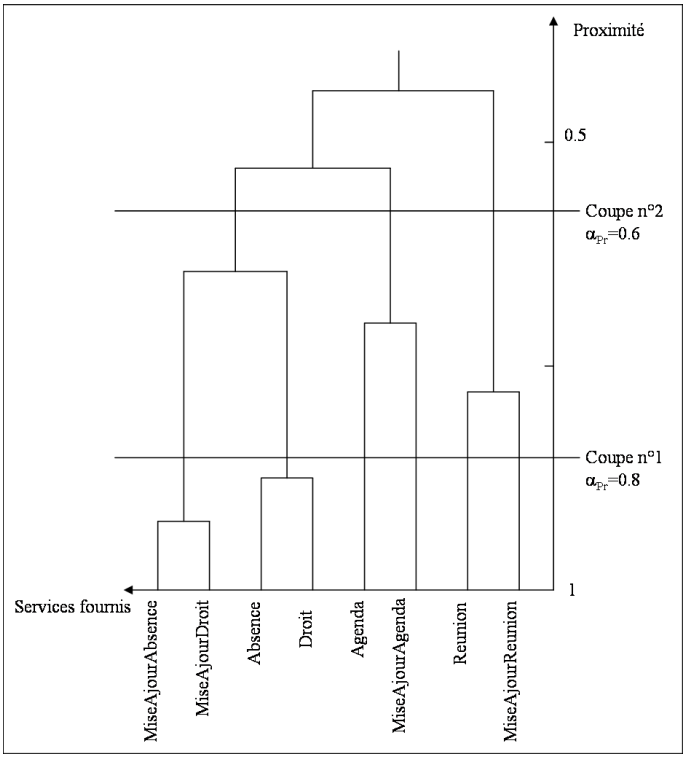

Figure 4.27 – Découpe du dendrogramme du composant Agenda-partagé

La valeur de ce seuil doit être comprise entre 0 et 1. Ensuite, le dendrogramme est découpé en fonction de ce seuil. Les *clusters* créés correspondent aux intersections entre la droite de proximité α_{Pr} et le dendrogramme. Ils constitueront par la suite les services fournis des différents sous-composants à créer.

Ainsi, si $\alpha_{Pr} = 1$, chaque service fourni par le composant adapté est contenu dans un *cluster* et donc va être fourni par un composant primitif (réification des services en composant). Plus la valeur de α_{Pr} sera proche de zéro, plus le nombre d'intersections sera faible et donc peu de « composants primitifs » seront créés ; ce qui engendrera une faible flexibilité du composant après sa transformation dans le format canonique.

Exemple de l'agenda-partagé
Par exemple, considérons le dendrogramme du composant *Agenda-partagé* présenté dans la figure 4.27. Si l'administrateur de l'application fixe la valeur du seuil de proximité à 0.8 (coupe numéro 1), nous obtenons six composants primitifs dont la spécification généré correspond à la figure 4.25. Dans le cas où $\alpha_{Pr} = 0.6$, nous obtenons trois composants (coupe numéro 2) : le premier composant fournit les services *MiseAjourAbsence*, *MiseAjourDroit*, *Absence* et *Droit* ; le second composant fournit les services *MiseAjourAgenda* et *Agenda* ; Enfin, le troisième composant fournit les services *Réunion* et *MiseAjour-Réunion*.

4.8 Étude et analyse des performances de l'approche d'auto-adaptation structurelle dynamique de composants

Contrairement au surcoût lié à l'adaptation structurelle par la ré-ingénierie qui est inhérent à la mise en œuvre de la distribution du composant, le surcoût lié à l'approche d'auto-adaptation structurelle dynamique est permanent. En fait, il fait référence :

- *au surcoût engendré par le format canonique du composant lors de son exécution*
  La génération du composant sous format canonique impose la mise en œuvre des mécanismes de communication et de synchronisation entre les composants primitifs même s'ils ne sont pas distribués. Ce qui implique que, contrairement à l'adaptation statique où le surcoût nécessaire pour garantir la cohérence du comportement du composant lié à sa distribution est indispensable, dans le cas dynamique, ce surcoût doit être considéré comme permanent. Cependant, il est nécessaire afin d'assurer une grande flexibilité du composant. En fait, plus le composant auto-adaptatif contient de composants primitifs, plus le nombre de combinaisons possibles pour la reconfiguration de sa structure sera important. Donc, l'augmentation de la flexibilité de la structure d'un composant entraîne une dégradation des performances de ce dernier qui peut être réduite par la parallélisation des services quand les composants primitifs sont distribués.
  Le calcul de ce surcoût est présenté dans le chapitre précédent ;

- *au surcoût engendré par l'acquisition et le traitement du contexte d'exécution pour déclencher une phase d'adaptation et calculer une structure du composant adaptée*
  L'acquisition et le traitement du contexte d'exécution d'un composant entraîne un surcoût en terme d'espace mémoire occupée et de performance. Concernant l'espace mémoire occupée, celle-ci dépend du nombre d'éléments de contexte pris en compte et de leur taille mémoire. De plus, afin d'optimiser le comportement de l'adaptation, nous avons vu qu'il est nécessaire de conserver un historique ; ce qui peut se révéler très consommateur en terme d'espace mémoire. Il doit donc être

envisagé d'adapter le nombre d'éléments pris en compte en fonction des ressources disponibles. Par ailleurs, le contexte doit être acquis et analysé en permanence afin de pouvoir garantir la continuité de service du composant en déclenchant une phase d'adaptation dès lors que les ressources disponibles sont jugées insuffisantes. Cette activité d'acquisition du contexte induit donc également un surcoût en termes de performance ;

- *au surcoût engendré par la prise de décision*
  La prise de décisions réalisée au cours d'un processus d'auto-adaptation introduit également un surcoût en terme d'espace mémoire occupée et de performance. Ce surcoût est étroitement lié aux mécanismes mis en œuvre pour déclencher une phase d'adaptation et générer automatiquement une structure adaptée à la situation. Ces deux tâches sont réalisées par l'intermédiaire de règles que nous avons définies dans les sections précédentes. Aussi, le surcoût engendré par ces deux tâches dépend du nombre de règles utilisées et de leur complexité.
  Concernant les règles de déclenchement de l'adaptation, ces règles sont de type évènement/action. De ce fait, leur complexité est négligeable. Concernant les règles de génération automatique de la spécification de l'adaptation, leur complexité dépend de leur tâche à accomplir. Les règles relatives à la classification des services en fonction de leur priorité ont une complexité de $O(\log n)$, n correspondant au nombre de services fournis par le composant. Les règles d'association des services à leur site de déploiement sont quant à elle linéaire ; les algorithmes de regroupement hiérarchique étant linéaires [72].
  Cependant, afin de diminuer le surcoût lié aux prises de décisions au moment de l'adaptation, il est possible d'agir sur les règles de génération de la spécification de la nouvelle structure. En effet, certaines règles notamment liées à l'optimisation du résultat de l'adaptation telles que prise en compte du contexte applicatif pourraient être ignorées de manière à réduire la complexité de la prise de décisions et ainsi diminuer le surcoût de l'adaptation ; notre objectif premier étant d'assurer la continuité de service et non optimiser les performances de l'application. Le choix de la prise en compte de ces règles d'optimisation doit donc être fait en fonction des ressources disponibles ;

- *au surcoût lié à la réalisation de l'adaptation du composant*
  Ce surcoût fait référence au surcoût engendré par la réalisation de la reconfiguration de la structure interne du composant en fonction de la spécification générée lors de la prise de décisions et au redéploiement des composants créés lors de la reconfiguration.
  Concernant le surcoût de la reconfiguration du composant, il est étroitement lié au modèle de composants utilisé.

$$C_{reconfiguration} = \lambda * C_{encapsulation} + \mu * C_{eclatement} \tag{4.15}$$

$$C_{encapsulation} = (\alpha + \beta) * C_{BD} + C_{CC} + (2 * \alpha + \beta + \gamma) * C_{BC} \tag{4.16}$$

$$C_{eclatement} = (2 * \alpha + \beta + \gamma) * C_{BD} + C_{CD} + (\alpha + \beta) * C_{BC} \tag{4.17}$$

λ : nombre d'opérations d'encapsulations de composants primitifs nécessaires pour l'obtention de la nouvelle structure

μ : nombre d'opérations d'éclatements de composites nécessaires pour l'obtention de la nouvelle structure

α : nombre de services encapsulés

β : nombre de fois où un services requis par l'un des composants encapsulés n'est pas fourni par le composite

γ : nombre de services requis par le composite

C_{BD} : coût de destruction d'une liaison

C_{BC} : coût de création d'une liaison

C_{CC} : coût de création d'un composite par encapsulation

C_{CD} : coût de destruction d'un composite par encapsulation

Le surcoût lié au redéploiement des sous-composants générés lors de la reconfiguration de la structure du composant dépend quant à lui des paramètres suivants :

1. des performances de l'infrastructure distribuée utilisée (bande passante, taux d'occupation du réseau, etc.),
2. de temps de chargement du composant transféré sur la machine distante,
3. du coût du transfert d'état du composant initial vers le nouveau composant déployé sur la machine distante.

Bilan Le surcoût lié à notre modèle de composants dynamiquement auto-adaptatifs étant relativement important comme nous avons pu le constater, il parait relativement peu envisageable de doter tous les composants d'une même application de tels mécanismes. Ainsi, une stratégie de gestion de l'adaptation au niveau de l'application s'impose. En effet, une centralisation de l'acquisition et du stockage du contexte d'exécution de l'application ainsi que des mécanismes de prises de décisions, sur un site permet de réduire fortement le coût de l'adaptation en terme d'espace mémoire occupé et de performances. Une telle affirmation renforce donc la nécessité de gérer l'adaptation au niveau macro comme nous l'avons proposé dans la section précédente.

4.9 Conclusion

Nous avons présenté dans ce chapitre une approche permettant à un composant de logiciel d'adapter automatiquement sa structure en fonction de son contexte d'exécution. Pour posséder une structure auto-adaptative, un composant doit être conforme à un modèle canonique particulier. Ce modèle est basé sur la réification d'interfaces fournies par le composant et l'intégration de mécanismes lui permettant de réaliser lui même son adaptation. Ces mécanismes sont chargés d'une part d'acquérir et d'analyser son contexte d'exécution afin de déclencher des phases d'adaptation et de déterminer une structure adaptée, lui garantissant une continuité et une qualité de service ; et d'autre part de modifier automatiquement sa structure afin de la rendre conforme à la spécification obtenue. Nous avons appliqué notre stratégie d'adaptation sur des composants destinés à être exécutés dans des environnements ubiquitaires.

Par ailleurs, nous avons constaté que l'adaptation structurelle d'un composant logiciel dans une application pouvait avoir des impacts sur les autres composants de l'application. De ce fait, nous avons proposé une approche permettant de coordonner les adaptations de composants contenus dans une application et ainsi généraliser notre approche pour l'adaptation d'architectures logicielles. Pour cela, nous avons proposé un modèle et deux stratégies permettant de coordonner l'adaptation de composants logiciels à différents niveaux. Le choix de la meilleure stratégie dépend fortement de l'application concernée. Ainsi, ce choix est laissé à l'appréciation de l'administrateur de l'adaptation.

Afin de mettre en œuvre notre approche sur des composants existants, nous avons proposé un processus de ré-ingénierie permettant d'obtenir un composant conforme à notre modèle d'auto-adaptation à partir d'un composant existant. Ce processus est basé sur notre approche d'adaptation structurelle par la ré-ingénierie, que nous avons présentée dans le chapitre précédent.

Nous avons évalué notre approche d'auto-adaptation structurelle dynamique. Cette évaluation nous a permis de constater que notre approche engendre un permanent surcoût lors de l'exécution centralisée du composant, lié à notre modèle de composants auto-adaptatifs. Cependant, ce surcoût est indispensable pour garantir une grande flexibilité du composant. Concernant le surcoût engendré par la prise en compte du contexte, il n'est pas négligeable ; cependant, il peut être réduit par le choix d'une stratégie d'adaptation au niveau de l'application, adaptée à la situation (*i.e.* partage de gestionnaire de contexte, centralisation des mécanismes de prise de décision, etc.).

Dans le chapitre suivant, nous présentons comment nous avons mis en œuvre toutes nos propositions dans le cadre d'un prototype et un scénario ubiquitaire.

CHAPITRE 5

Implémentation et expérimentation de l'adaptation structurelle dans des environnements ubiquitaires

5.1 Introduction

Dans les chapitres précédents, nous avons présenté notre proposition d'adaptation structurelle de composants logiciels réalisée de manière statique, dynamique et enfin automatique. Dans ce chapitre, nous présentons d'une part l'implémentation support de notre approche et d'autre part le scénario ubiquitaire que nous avons développé pour l'expérimenter.

Dans un premier temps, nous détaillons la réalisation de l'outil Static-Scorpio-Tool permettant de mettre en œuvre la ré-ingénierie structurelle d'un composant existant. Nous montrons notamment les verrous techniques liés à sa réalisation. Ensuite, nous détaillons notre outil Auto-Scorpio-Tool permettant de réaliser l'adaptation structurelle dynamique et automatique. En fait, il permet de transformer un composant existant en un composant structurellement et dynamiquement auto-adaptatif. Ces deux outils agissent sur des composants implémentés dans la plate-forme Julia qui est une implémentation Java du modèle de composants Fractal.

Par la suite, nous présentons un scénario mettant en jeu une application ubiquitaire et nécessitant la mise en œuvre de notre approche d'adaptation structurelle comme action d'adaptation. Il se déroule dans le cadre d'un chantier de construction en bâtiment et contient une application capable de s'adapter automatiquement à son contexte d'exécution en assurant la continuité des services relatifs à un utilisateur.

5.2 Fractal comme modèle de composants support et Julia comme plate-forme d'implémentation

Afin d'implémenter notre approche d'adaptation structurelle de composants logiciels, nous avons établi un certain nombre de choix. Ces choix sont les suivants :

1. *modèle de composants cible : Fractal*
   Nous avons constaté que, dans la littérature, aucune approche ne propose une implémentation pleinement conforme à la spécification présentée dans la section 2.5.2. Notre choix pour expérimenter notre approche s'est alors porté sur des composants conformes au modèle Fractal [39] dont la spécification est relativement proche de celle précédemment citée (voir figure 5.2). Fractal est un modèle de composants académique doté de caractéristiques faisant de lui l'un des modèles privilégiés dans le cadre de travaux de recherche. Les caractéristiques de Fractal qui nous ont poussées à développer notre approche sur des composants conformes à ce modèle sont les suivantes :

   - *la récursivité du modèle*
     Fractal est un modèle de composants hiérarchiques. En fait, un composant peut être de deux types : primitif ou composite. Un composant composite peut contenir par encapsulation d'autres composants primitifs ou non. Chaque composant dispose d'une membrane destinée à contrôler son comportement et sa structure. Dans le cas de composants composites, la membrane est utilisée pour contrôler l'encapsulation de ses sous-composants (voir Figure 5.1). La présence d'une telle membrane va donc faciliter la configuration du composite généré au cours de l'adaptation structurelle ;

   - *la réflexion*
     Les composants Fractal offrent des mécanismes d'introspection et d'intercession. En fait, un composant Fractal contient deux parties : une partie « contenu » et une partie « contrôleur » (voir Figure 5.1). La partie « contenu » fait référence aux services fonctionnels fournis par le composant. La partie « contrôleur » est quant à elle chargée de représenter le composant (*i.e.* introspection), de le configurer (*i.e.* gestion des attributs, etc.) et de l'administrer (*i.e.* gestion de son contenu, gestion des connexions, gestion du cycle de vie, etc.). Ces mécanismes sont intégrés au composant sous la forme d'interfaces non fonctionnelles dites « de contrôles ». Ces interfaces vont donc permettre de mettre en œuvre facilement l'auto-adaptation structurelle dynamique ;

   - *l'ouverture du modèle*
     Le modèle de composants Fractal est un modèle ouvert. Cette propriété signifie que de nouveaux services non fonctionnels peuvent être associés à un composant au travers des interfaces de contrôle. De ce fait, les services non fonctionnels générés au cours de l'adaptation structurelle par sa ré-ingénierie peuvent être introduits sous la forme d'interfaces de contrôle ; ce qui permet de garantir la transparence de l'adaptation. De même, l'adaptation structurelle peut être intégrée comme un service non fonctionnel proposé par le composant ;

   - *le partage de composant*
     Un composant peut être inclu (*i.e.* partagé) dans un ou plusieurs composants. Cette propriété peut être utilisée dans le cadre de l'adaptation afin d'éviter la duplication de certains composants non fonctionnels qui pourraient engendrer un surcoût trop important de notre approche. Par exemple, nous avons vu que dans le cadre de l'une des deux stratégies d'auto-adaptation au niveau macro (celle relative à la stratégie d'adaptation au niveau composant), il est préférable que pour des questions de coût, le composant de gestion de contexte soit partagé par tous les composants déployés sur un même site ;

- *l'utilisation d'un langage architectural*
  Fractal fournit un langage architectural appelé FractalADL permettant de spécifier l'architecture d'applications à base de composants. Ce langage est ouvert et extensible. Il permet de faciliter la vision de la structure d'un composant ou d'une application. De ce fait, il peut être utilisé pour mettre en œuvre les phases de spécifications de l'adaptation structurelle (où un acteur externe de l'adaptation doit spécifier la structure du composant qu'il souhaite obtenir) ;

- *l'indépendance par rapport à une implémentation spécifique*
  Le modèle de composants Fractal n'est pas défini par rapport à une implémentation spécifique. Il existe différentes implémentations du modèle de composants Fractal : à titre d'exemple, nous pouvons citer Julia [38] qui est une implémentation Java et Fractalk [30], une implémentation Smalltalk. Nous allons détailler ci-après notre choix.

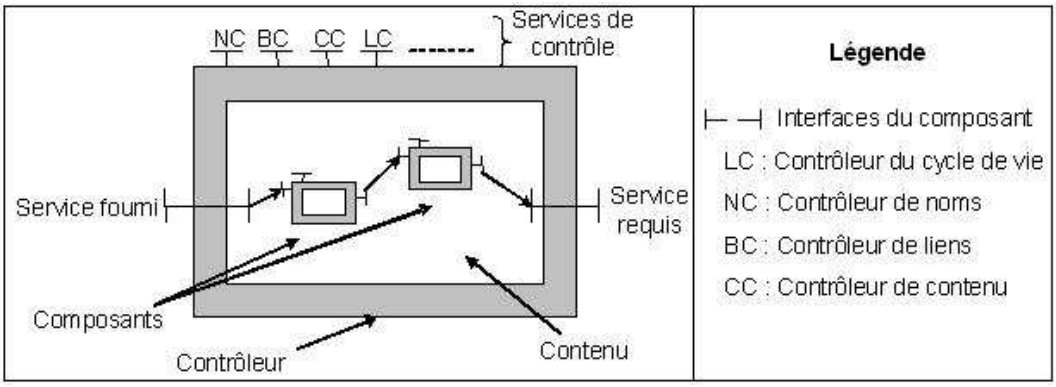

Figure 5.1 – Architecture d'un composant Fractal

2. *infrastructure logicielle de déploiement : Julia*
   Comme nous l'avons évoqué précédemment, le modèle de composants Fractal est dissocié d'une implémentation. De ce fait, nous devons sélectionner une implémentation parmi celles existantes. Notre choix s'est alors porté sur la plate-forme Julia [38] qui est la première implémentation du modèle de composants Fractal en Java. Julia est une des implémentations entièrement conformes au modèle de composants Fractal des plus aboutie. De plus, elle a été conçue pour être légère et efficace, ce qui est essentiel étant donné que nous avons pour objectif d'appliquer notre approche aux environnements ubiquitaires qui se caractérisent notamment par des ressources limitées. Par ailleurs, l'utilisation du langage de programmation Java, nous procure de nombreux avantages tels que sa portabilité (*i.e.* des machines virtuelles spécifiques aux environnements à ressources limitées sont disponibles) ou ses propriétés liées aux modèles objets.

   De plus, il fournit des mécanismes permettant de distribuer facilement un composant grâce à un intergiciel appelé FractalRMI [37]. Ce dernier, réalisé à base de composants Julia, permet d'établir des communications synchrones distantes entre des composants Julia.

   Conformément au modèle Fractal, un composant Julia peut être de deux types : primitif, il s'agit alors d'un objet Java standard ou composite, il s'agit alors d'un composant pouvant contenir d'autres composants primitifs ou non. Trois étapes sont nécessaires à la création de composants

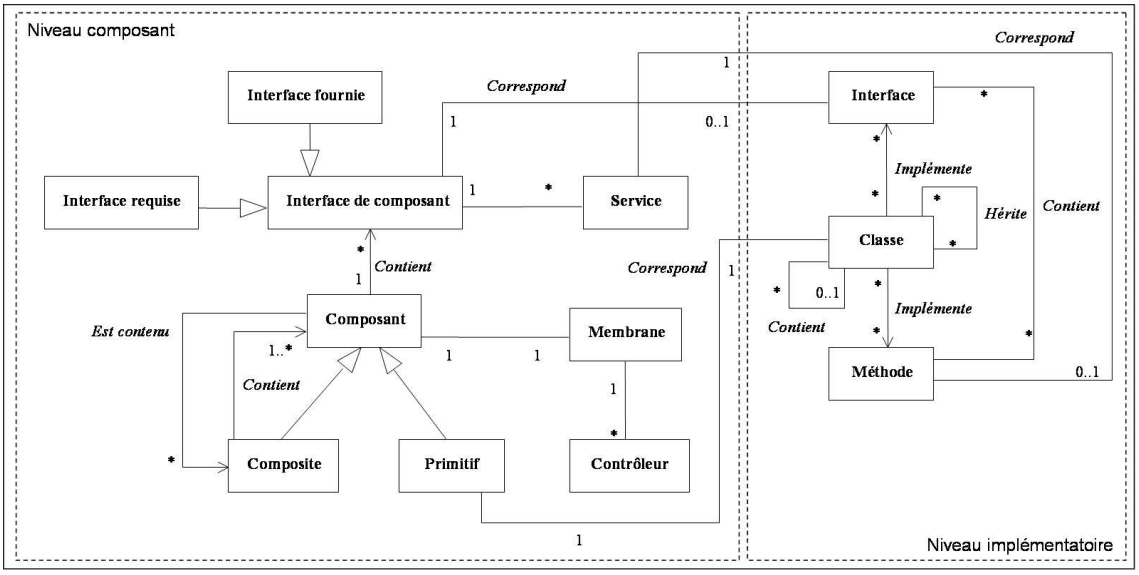

Figure 5.2 – Modèle d'un composant Fractal en Julia

Julia : tout d'abord, il faut définir les interfaces relatives aux services fournis et requis par le composant. Ensuite, il faut implémenter les composants primitifs et enfin décrire la structure et le comportement du composant (*i.e.* établir les connexions entre les interfaces clients et serveurs). La construction de l'architecture peut être réalisée de deux manières différentes :

- soit en utilisant un langage de description d'architecture appelé FractalADL. Dans ce cas, une application Fractal/Julia est constituée d'un ensemble de classes Java et d'un ou plusieurs fichiers contenant la description architecturale de l'application. Cette description formulée dans un langage de type XML dont la DTD est fournie dans la figure 5.3 est alors analysée par des composants du noyau Julia afin de créer les instances des composants spécifiés ;

- soit en utilisant du code Fractal spécifique fourni par une API Java. Dans ce cas, l'exécution de ce code Julia va entraîner la création d'instances de composants.

Dans le cadre de notre expérimentation, nous avons supposé que les composants Julia sur lesquels nous réalisons l'adaptation structurelle sont implémentés en pur Java (*i.e.* toutes les méthodes sont implémentées en Java).

5.3 Static-Scorpio-Tool : un outil pour l'adaptation structurelle par la ré-ingénierie

Le premier outil que nous avons développé dans le cadre de Scorpio-Tool a pour objet d'adapter la structure de composants logiciels monolithiques en fonction d'une spécification réalisée par un acteur externe de l'adaptation. Cet outil met en œuvre le processus de ré-ingénierie que nous avons proposé dans le chapitre 3. Il permet également de réaliser un déploiement distribué automatique du composant issu de l'adaptation.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!ELEMENT definition (comment*,interface*,component*,binding*,content?,attributes?,controller?,
template-controller?,logger?,virtual-node?,coordinates*) >
<!ATTLIST definition
  name CDATA #REQUIRED
  arguments CDATA #IMPLIED
  extends CDATA #IMPLIED >

<!ELEMENT component (comment*,interface*,component*,binding*,content?,attributes?,controller?,
template-controller?,logger?,virtual-node?,coordinates*) >
<!ATTLIST component
  name CDATA #REQUIRED
  definition CDATA #IMPLIED >

<!ELEMENT interface (comment*) >
<!ATTLIST interface
  name CDATA #REQUIRED
  role (client | server) #IMPLIED
  signature CDATA #IMPLIED
  contingency (mandatory | optional) #IMPLIED
  cardinality (singleton | collection) #IMPLIED >

<!ELEMENT binding (comment*) >
<!ATTLIST binding
  client CDATA #REQUIRED
  server CDATA #REQUIRED >

<!ELEMENT attributes (comment*,attribute*) >
<!ATTLIST attributes
  signature CDATA #IMPLIED >

<!ELEMENT attribute (comment*) >
<!ATTLIST attribute
  name CDATA #REQUIRED
  value CDATA #REQUIRED >

<!ELEMENT controller (comment*) >
<!ATTLIST controller
  desc CDATA #REQUIRED >

<!ELEMENT template-controller (comment*) >
<!ATTLIST template-controller
  desc CDATA #REQUIRED >

<!ELEMENT content (comment*) >
<!ATTLIST content
  class CDATA #REQUIRED >

<!ELEMENT logger EMPTY >
<!ATTLIST logger
  name CDATA #REQUIRED >

<!ELEMENT virtual-node EMPTY >
<!ATTLIST virtual-node
  name CDATA #REQUIRED >

<!ELEMENT coordinates (coordinates*) >
<!ATTLIST coordinates
  name  CDATA #REQUIRED
  x0    CDATA #REQUIRED
  x1    CDATA #REQUIRED
  y0    CDATA #REQUIRED
  y1    CDATA #REQUIRED
  color CDATA #IMPLIED >

<!ELEMENT comment EMPTY >
<!ATTLIST comment
  language CDATA #IMPLIED
  text CDATA #IMPLIED >
```

Figure 5.3 – Schéma DTD de FractalADL

5.3.1 Architecture de Static-Scorpio-Tool

5.3.1.1 Vue globale de l'outil

L'outil permettant de mettre en œuvre l'adaptation structurelle par la ré-ingénierie comprend trois composants (voir Figure 5.4) :

- *un composant dédié à découverte de composants et de services :*
  Ce composant permet de contrôler le résultat de l'adaptation structurelle de par la vérification de la disponibilité des services qui ont été redéployés sur des sites distants. Ainsi, ce composant offre des services permettant à un acteur externe de l'adaptation de rechercher des services fournis par des composants fonctionnels pouvant être déployés sur les nœuds d'une infrastructure distribuée. Deux modes de recherche sont disponibles à travers l'interface de l'outil. Le premier mode consiste à spécifier une adresse IP, ensuite l'outil va rechercher les composants mis à disposition par le site et afficher la liste des services qu'ils proposent. Le deuxième mode permet de parcourir une infrastructure distribuée spécifiée par l'administrateur, en recherchant les composants et les services qu'ils proposent. Cette fonctionnalité de découverte de composants peut être utilisée pour la maintenance et l'évolution d'une application ou bien pour la création de nouvelles applications par l'assemblage de composants existants, déjà distribués sur l'infrastructure disponible ;

- *un composant serveur de composants logiciels :*
  Grâce aux services fournis par ce composant, un site pourra réceptionner des composants fonctionnels provenant de sites distants et les exécuter automatiquement. Un composant serveur doit être préalablement déployé sur chaque nœud de l'infrastructure distribuée afin d'exécuter la procédure de déploiement automatique ;

- *un composant de réalisation de l'adaptation :*
  Ce composant offre les services permettant de réaliser l'adaptation structurelle interne de composants logiciels existants. Tout d'abord, l'utilisateur doit spécifier le nom du composant à adapter, le répertoire dans lequel se trouve le code source de l'application qui le contient ainsi que la description d'architecture de l'application. Ensuite, l'utilisateur pourra procéder à l'étape de spécification de la structure du résultat à obtenir. Cette étape est guidée par l'outil qui propose à l'utilisateur la liste des interfaces disponibles destinées à construire les nouveaux sous-composants ainsi que la liste des sites susceptibles de déployer automatiquement un sous-composant généré. L'administrateur doit alors désigner le site d'implantation de chaque sous-composant[1]. Une fois la nouvelle structure spécifiée, le code source du composant et la description de l'architecture de l'application sont automatiquement mis à jour (*i.e.* analyse, refactorisation et instrumentation de code). Au cours de cette étape, des mécanismes de distribution sont intégrés automatiquement au composant composite. Dans le cadre de notre prototype, nous avons utilisé les protocoles fournis par FractalRMI [37] pour assurer la distribution du composant adapté. Les fichiers contenant le code source généré ainsi que la description de la nouvelle architecture du composant et de l'application sont alors automatiquement compilés. Ensuite, les fichiers binaires ainsi obtenus sont envoyés par l'intermédiaire de l'infrastructure distribuée, sur leur site de déploiement correspondant[2]. Enfin, ces fichiers sont exécutés automatiquement.

[1] Par défaut, le composant est déployé sur le site local.

[2] Chaque site de déploiement fournit une infrastructure capable de recevoir et d'exécuter des fichiers binaires (*i.e.* serveur de composants logiciels).

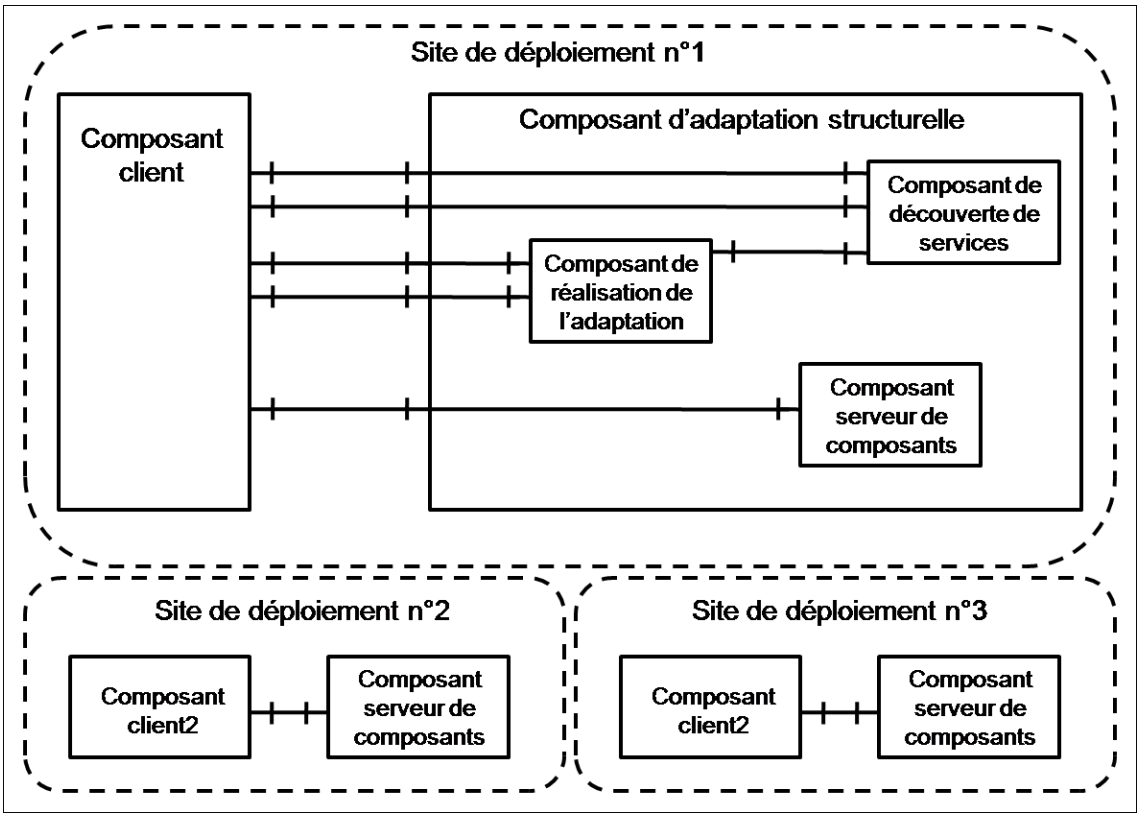

Figure 5.4 – Architecture de l'outil d'adaptation structurelle par la ré-ingénierie de composants existants

5.3.1.2 Architecture détaillée du composant de réalisation de l'adaptation

Le principal composant de Static-Scorpio-Tool est le composant de réalisation de l'adaptation. Il s'agit d'un composant composite constitué de six sous-composants. Son architecture est présentée dans la figure 5.5. Les différents sous-composants présents dans l'architecture du composant de réalisation de l'adaptation sont les suivants :

- *un composant de gestion d'architecture*
  Ce composant est chargé d'analyser la description de l'architecture de l'application initiale et de générer une nouvelle description correspondant à la nouvelle structure du composant à adapter ;

- *un composant de gestion de code*
  Ce composant est consacré à la manipulation du code source du composant à adapter. Il contient deux sous-composants permettant d'analyser le code source du composant initial (composant d'analyse de code) et de générer le code correspondant aux nouveaux composants créés (composant de génération de code).
  Le composant d'analyse de code se charge de générer automatiquement les graphes SBDG servant de support à la fragmentation du composant. En fait, la génération des graphes SBDG correspondant aux différents composants créés par l'adaptation se base sur l'extraction du modèle Famix

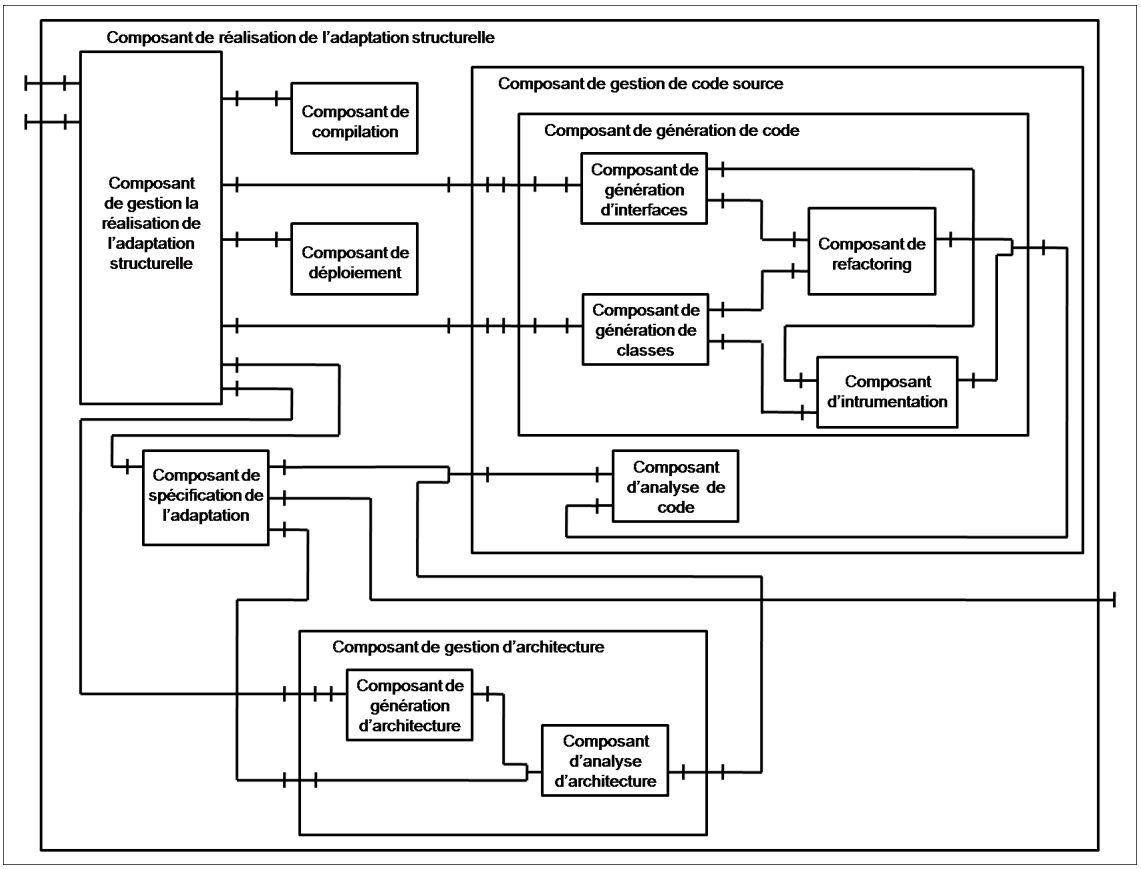

Figure 5.5 – Architecture du composant de réalisation de l'adaptation structurelle

[48] de l'application. Ce modèle est développé dans le cadre du projet FAMOOS [51] pour représenter des systèmes à base d'objets. Famix est un méta-modèle qui fournit une représentation indépendante du langage, de codes sources orientés objet (voir Figure 5.6 [51]). En fait, le modèle Famix est utilisé afin de déterminer les entités structurelles (classes, méthodes, attributs) ainsi que les relations entre ces entités (appels de méthodes, accès à un attribut) au niveau de l'implémentation du composant (niveau objet). Le format d'échange de donnée utilisé pour le transfert de modèles entre les différents outils est le standard industriel appelé CDIF. Le composant d'analyse de code utilise alors les fichiers CDIF générés par l'outil d'extraction du modèle Famix pour générer les graphes SBDG correspondant aux composants créés. De nouvelles entités structurelles liées au paradigme composant sont introduites dans le modèle extrait (*i.e.* composant, interfaces fournies, méthodes internes, services) : les composants à générer et les interfaces fournies sont déterminés par l'analyse de la spécification de l'adaptation ; chaque composant fournit un ensemble d'interfaces (création d'une dépendance structurelle entre le composant et les interfaces qu'il fournit). Chaque interface fournie est associée à une interface implémentée par le composant. Cette association est déterminée par l'analyse de la description de l'architecture de l'application. Les autres dépendances structurelles liées à l'implémentation au niveau objet (dépendances structurelles en interfaces et méthodes, entre interfaces et classes d'implémentation, entre méthodes internes et classes d'implémentation, etc.) sont extraites par l'analyse du fichier CDIF. Concernant les dépendances fonctionnelles (*i.e.* appels de méthodes), celles-ci sont également déterminées par l'analyse du fichier CDIF ;

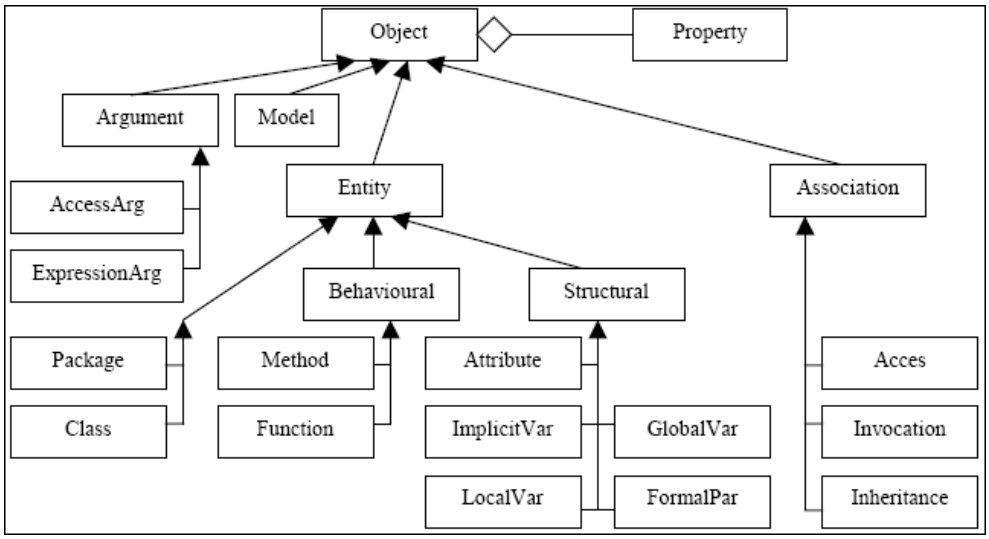

Figure 5.6 – Méta-modèle Famix

La génération de code est chargée de deux tâches : construire les classes qui vont être associées aux composants générés (composant de génération de classe) et introduire les interfaces qui vont permettre de réaliser l'assemblage de ces composants (composant de génération d'interfaces). La réalisation de ces deux tâches nécessite l'utilisation de mécanismes de « refactorisation » et d'instrumentation de code respectivement fournis par le composant de *refactoring* et le composant d'instrumentation.

- *un composant de compilation*
  Un composant de compilation est également présent dans l'architecture de Static-Scorpio-Tool. Il est chargé de générer automatiquement le code binaire du composant adapté en fonction du code généré par les composants de gestion d'architecture et de gestion de code ;

- *un composant de gestion de déploiement*
  Afin de réaliser un déploiement automatique du composant résultat de l'adaptation, un composant de gestion de déploiement est chargé d'encapsuler les codes binaires générés par le composant de compilation puis d'envoyer les différents paquets obtenus (un par site de déploiement du composant) aux sites de déploiement correspondants. En fait, chaque paquet est envoyé au composant serveur déployé sur chaque site distant. Ces composants serveurs vont alors « dépaqueter » les codes binaires correspondants aux composants devant être déployés sur le site. Enfin, ces codes sont exécutés automatiquement. Une fois que tous les paquets ont été transmis et sont déployés sur leur site correspondant, le composant issu de l'adaptation devient opérationnel ;

- *un composant dédié à l'interfaçage graphique de la spécification*
  Un sous-composant du composant de réalisation de l'adaptation est chargé de l'interfaçage de la spécification de l'adaptation. Cette opération consiste à spécifier pour chaque composant généré, ses interfaces fournies et son site de déploiement. Elle est guidée par l'outil grâce aux mécanismes d'introspection fournis par la plate-forme Julia et à des systèmes de contrôle. En fait, les mécanismes d'introspection vont permettre de déterminer les interfaces fournies par le composant à adapter qui vont servir de base à la spécification. Les mécanismes de contrôle vont quant à eux être chargés de vérifier la cohérence de la spécification (une même interface ne peut être fournie par plusieurs sous-composants, etc.). Les sites de l'infrastructure susceptibles de déployer des sous-composants sont sélectionnés en utilisant un service fourni par le composant de recherche de composants. En fait, seuls les sites sur lesquels sont déployés un composant serveur sont susceptibles de déployer les composants issus de l'adaptation structurelle ;

- *un composant de gestion de la réalisation de l'adaptation structurelle*
  Le dernier sous-composant est chargé d'orchestrer l'adaptation structurelle. De ce fait, il requiert tous les services des composants cités précédemment. Il offre deux services : le premier permet de réaliser l'adaptation à partir d'une spécification fournie en paramètre et le deuxième utilise les services fournis par le composant d'interfaçage graphique pour générer la spécification.

5.3.2 Problèmes liés à l'analyse et à la génération de code source de composants Julia

La mise en œuvre de notre processus d'adaptation engendre l'apparition de nombreux problèmes liés à l'analyse et l'instrumentalisation du code due à la spécificité Java. Les problèmes se posent sur cinq niveaux : les attributs, les constructeurs et autres initialiseurs, les méthodes, les classes et enfin les interfaces.

5.3.2.1 Gestion des attributs partagés

En Fractal, on distingue deux types d'attributs : les attributs du composant et les attributs de contenu. Les attributs du composant sont uniquement de type primitif ou « *String* » donc par conséquent, il n'existe pas de problèmes liés au alias ou au passage par référence. L'utilisation d'accesseurs implémentant les

méthodes de l'interface « *attribute-controller* » est obligatoire pour tout accès à la ressource par un autre composant. Dans le cadre du maintien de la cohérence pour ce type d'attribut, il faut identifier à tout moment les points d'accès à cet attribut de manière à déterminer les sections d'exclusion mutuelle et à propager les modifications dans le cas d'une solution avec duplication.

Si on opte pour une stratégie de centralisation des ressources, deux solutions sont possibles :

- *passage par des accesseurs partout*
  Si l'attribut est appelé sans utiliser des accesseurs, il faut instrumentaliser le code de manière à ce que le composant accède à ses attributs au moyen d'accesseurs. Par la suite, tout accès à un attribut va être effectué par l'intermédiaire d'une méthode (*i.e.* accesseur). L'avantage de cette technique est qu'elle facilite la maintenance des données (*i.e.* gestion de la cohérence et de la synchronisation au niveau des méthodes et non au niveau des instructions) ;

- *passage par accesseurs si besoin*
  Cette stratégie consiste à modifier le code de chaque méthode de manière à ce que celle-ci utilise des accesseurs pour accéder à un attribut uniquement lorsqu'il n'est pas contenu dans le composant lui-même. L'inconvénient de cette méthode est l'obligation de gérer la maintenance des données au niveau des instructions (*i.e.* à chaque accès à la ressource dans le code).

Si l'on opte pour une duplication des ressources, l'utilisation d'accesseurs n'est pas obligatoire dans l'implémentation du composant. On peut alors appliquer les mêmes stratégies que dans le cas précédent.

Un des problèmes liés à la gestion des attributs réside dans leur portée. Lors de l'étape de mise à jour de la structure, il faut déterminer les méthodes qui font appel à un attribut. Pour chaque attribut, il faut balayer le corps de la méthode en recherchant toutes les occurrences de chaque nom d'attribut. Si le nom d'un attribut apparaît au moins une fois dans le corps de la méthode alors l'attribut et la méthode sont fonctionnellement liés. Or, un attribut interne d'une méthode peut avoir le même nom que l'un des attributs de la classe. Il faut donc tenir compte de la portée des variables. Si lors du balayage de la méthode on observe la déclaration d'une variable portant le même nom qu'un attribut, il faut extraire le bloc de code dans lequel se trouve la déclaration et ne considérer que les appels à l'attribut lorsque ce dernier est appelé par l'intermédiaire du mot clé « *this* » correspondant à l'instance de l'objet (*i.e.* du composant).

Le cas des attributs de contenu est différent. Ces derniers sont des attributs de la classe d'implémentation du composant mais ne sont pas des attributs du composant. Ils peuvent être de type primitif ou de type classe. Leur gestion est donc différente des attributs de composant. Donc aux problèmes liés à la gestion des attributs de composants s'ajoutent de nouveaux problèmes liés à l'utilisation d'objets. L'utilisation d'accesseurs n'est pas obligatoire. S'ils existent, ils ne doivent pas être définis dans une interface de type « *attribute-controller* ». Les problèmes rencontrés lors de l'analyse du code sont essentiellement liés à la gestion d'alias si l'attribut est une classe. Il faut donc vérifier toutes les affectations pour connaître les variables pour lesquelles la cohérence devra être maintenue. La possibilité de faire appel à une méthode en passant par référence ses paramètres constitue également une possibilité pour la création d'alias. Une méthode ne peut pas modifier un paramètre d'un type primitif, ni modifier un paramètre objet pour qu'il fasse référence à un autre objet. Par contre, une méthode peut modifier les valeurs d'un paramètre objet. Donc si l'un des paramètres est un attribut de la classe passé par référence, il faudra alors maintenir la cohérence de ce paramètre chaque fois qu'il sera consulté ou modifié. Une solution possible à ce problème est de créer une nouvelle méthode en supprimant le paramètre correspondant à la ressource et ajouter des accesseurs.

```
Methode(type a) {... a=b ; ...} '        Methode() {... seta(b) ; ...}
```

5.3.2.2 Gestion des initialiseurs et les constructeurs

Les constructeurs de la classe d'implémentation du composant doivent subir un traitement particulier. Deux solutions sont possibles pour gérer les constructeurs.

- *Stratégie de duplication des initialiseurs et des constructeurs*
  La première solution consiste à dupliquer le contenu du constructeur dans chaque classe d'implémentation de chaque nouveau composant spécialisé. Dans ce cas, il faut également dupliquer toutes les déclarations des attributs de la classe d'implémentation de départ. Ce cas va poser des problèmes d'optimisation car dans la classe, vont être déclarés des attributs qui ne seront utilisés que dans le constructeur. Aucune maintenance ne va être appliquée sur ces derniers. On peut les considérer comme des attributs « fantômes ».

- *Stratégie de fragmentation des initialiseurs et des constructeurs*
  Si un constructeur est présent dans la classe d'implémentation du composant, il faut le découper de manière à le répartir suivant les besoins (*i.e.* dépendances aux attributs). Il faut donc déterminer quelle instruction agit sur quel attribut afin de répartir les instructions dans les différentes implémentations.

Un initialiseur est un bloc d'instructions encadré par des accolades, destiné à être exécuté juste avant le constructeur d'une classe. Leur traitement est identique à celui du constructeur.

Par ailleurs, une classe non imbriquée peut être munie de un ou plusieurs blocs « *static* » qui sont exécutés au chargement de la classe. Lors d'une instanciation d'une classe, les initialiseurs de cette dernière sont exécutés immédiatement avant le constructeur (*i.e.* après que les données statiques soient allouées et avant le constructeur). Un initialiseur statique est un bloc d'instructions encadré par des accolades possédant un modificateur « *static* ». Les initialiseurs statiques permettent de lancer des traitements uniquement pour la première instance d'une classe, car ils ne sont exécutés qu'une seule fois. Un bloc statique peut servir à l'initialisation des attributs « *static* », ou au chargement de librairies. Le traitement à appliquer est identique au précédent.

5.3.2.3 Gestion des méthodes partagées

En Julia, les services sont des méthodes de la classe d'implémentation d'un composant mais toutes les méthodes ne sont pas des services. Seules les méthodes appartenant à une interface sont des services. Il faut donc faire la distinction entre les méthodes internes à la classe d'implémentation du composant et les services fournis par le composant.

Dans le cas des méthodes internes à un composant (*i.e.* méthodes non accessibles de l'extérieur : ce n'est pas un service fourni par le composant), deux stratégies de gestion sont possibles :

- *une stratégie de centralisation des méthodes internes*
  Si la méthode n'utilise aucune ressource et ne fait appel à aucune méthode d'un composant de l'application, celle-ci peut être dupliquée dans le composant « base de données ».
  Si ces méthodes affectent une ou plusieurs ressources sans avoir de liens sémantiques avec les

services proposés par le composant alors il faut copier cette méthode dans l'implémentation du composant « bases de données » contenant les ressources. Par exemple, si on a une ressource qui est un tableau d'entier et une méthode de tri qui agit sur ce tableau, la méthode va être copiée dans le même composant qui contient la ressource c'est-à-dire le composant « base de données ».
Si ces méthodes ont un lien sémantique avec un service proposé par le composant. Alors, le code de ces méthodes va être dupliqué dans chaque implémentation des nouveaux composants issus de l'adaptation dont une méthode au moins fait appel à celle-ci.
Si la méthode est fortement liée à un composant spécifié lors de la première étape, alors il faut copier son code dans l'implémentation du composant puis l'associer à une interface. De cette manière, les autres composants créés vont pouvoir accéder à ce service par l'intermédiaire de l'interface. Ce nouveau service ne doit pas être accessible par les autres composants de l'application (*i.e.* authentification des composants) ;

- *une stratégie de duplication des méthodes internes*
  Dans ce cas il faut copier ces méthodes dans chaque composant issu de l'adaptation qui contient une méthode faisant appel à celle-ci.

Pour établir le graphe des dépendances entre les composants ainsi que pour mettre en œuvre les sections critiques, il est indispensable d'analyser le code source du composant à adapter. Il faut déterminer pour chaque service (*i.e.* méthode), l'ensemble des attributs et des méthodes qui sont utilisés par celui-ci. L'analyseur doit être capable de retrouver les méthodes qui sont appelées ainsi que le type de ses paramètres. En effet, deux méthodes peuvent avoir le même nom mais des paramètres de type différent. Les paramètres d'une méthode peuvent être une variable, une instruction ou bien l'appel à une méthode. Si le paramètre est une variable (attribut de la classe, variable de la méthode, variable globale, un attribut d'une classe interne, etc.), il faut déterminer son type. L'analyseur est donc confronté au problème du polymorphisme. Si le paramètre est une instruction, l'analyseur devra être capable de déterminer le type de la valeur correspondant à l'instruction. Et enfin, si le paramètre est l'appel à une méthode (*i.e.* service du ou d'un autre composant, méthode interne au composant, méthode d'une classe interne au composant, etc.), il faut déterminer ce type de retour. Le polymorphisme dynamique constitue un des problèmes majeurs dans l'analyse de codes Java.

```
1  Class A { public void methode(int a) {...} }
2  Class B extands A { public void methode(int a) {...} }
3
4  ...
5  A a;
6  If (...)
7          {a=new A();}
8  else
9          {a=new B();}
10 a.methode(1);
11 ...
```

5.3.2.4 Gestion des classes

La classe d'implémentation d'un composant est spécifiée au moment de l'assemblage. Celle-ci peut correspondre à un nœud d'une hiérarchie de classe (*i.e.* héritage des classes d'implémentation des composants). La seule contrainte est que la classe spécifiée pour l'implémentation du composant doit être concrète (*i.e.* toutes les méthodes la contenant doivent être implémentées, aucune ne doit être abstraite).

Si dans la hiérarchie, aucune méthode n'est redéfinie, l'analyseur doit être capable de parcourir la hiérarchie de classe afin de récupérer le code source de toutes les méthodes qui sont implémentées dans la classe spécifiée. Deux solutions sont possibles : soit on crée une seule classe d'implémentation pour chaque composant qui va fournir l'ensemble des services de la hiérarchie (cette solution n'est valable que si aucune des méthodes d'une super classe n'est redéfinie dans une sous-classe), soit on reconstitue la hiérarchie de classe. Pour cela, l'analyseur doit être capable de parcourir la hiérarchie de classe pour retrouver les interfaces disponibles ainsi que les classes contenant l'implémentation de chaque méthode afin de reconstituer la hiérarchie. Il peut être possible de fusionner deux classes mère et fille si aucune méthode de la classe mère n'est redéfinie dans la classe fille.

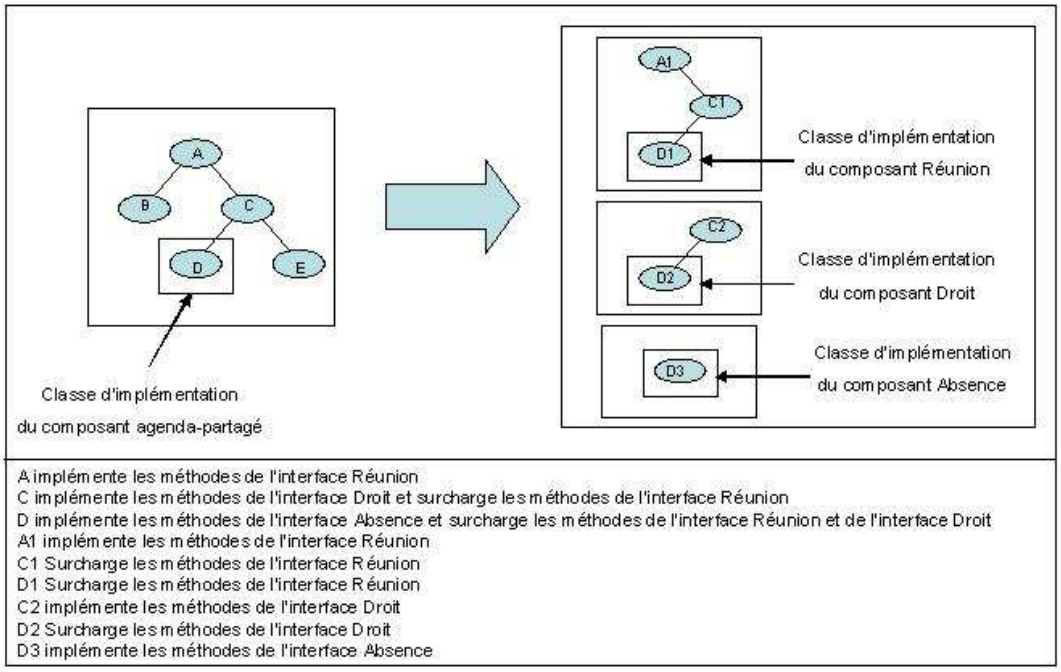

Figure 5.7 – Graphe d'héritage des classes d'implémentation des composants

Le traitement des classes internes est identique à celui des méthodes internes. Un autre problème qui se pose dans l'analyse du code Java est dû à la possibilité de concevoir des classes génériques (*i.e.* paramétrage du type des attributs). La classe d'implémentation du composant peut être l'implémentation d'une classe générique. Deux solutions sont possibles pour résoudre le problème lié aux classes génériques : soit on fait la fusion de la classe générique et de la classe d'implémentation (*i.e.* on remplace les types génériques par les types spécifiés lors de l'implémentation), soit on crée la classe générique contenant uniquement les méthodes souhaitées et la classe spécifiant les types des variables.

5.3.2.5 Gestion des interfaces

Pour réaliser l'étape de spécification, il faut déterminer l'ensemble des interfaces du composant à adapter. Si la classe d'implémentation est un nœud d'une hiérarchie de classe alors il faut repérer toutes les interfaces qui sont implémentées en partant du nœud et en remontant dans la hiérarchie de classe jusqu'à la racine. S'il existe une hiérarchie d'héritage au niveau des interfaces, il faut appliquer la même

stratégie que pour les classes de manière à obtenir l'ensemble des signatures des méthodes que doit fournir la classe qui implémente l'interface. Par ailleurs, la génération des nouveaux composants issus de l'adaptation peut entraîner des conflits de noms dûs à l'héritage multiple des interfaces. En effet, si deux méthodes appartenant à des interfaces différentes ont les mêmes noms et que la spécification prévoit la création d'un seul composant contenant ces deux interfaces alors nous pouvons être confrontés à des conflits. Trois cas sont possibles :

- soit les deux méthodes ont des paramètres différents,
  Dans ce cas, il suffit de transférer l'implémentation de ces deux méthodes dans la classe correspondante.

- soit elles ont exactement la même signature,
  Dans ce cas, si le type de retour est « *void* », une solution pourrait être de concaténer le code source de ces deux méthodes afin de les exécuter séquentiellement. Par exemple, considérons le cas de la méthode *afficher* dont la signature est la suivante : *public void afficher()*. Cette méthode est à la fois contenue dans l'interface *Réunion* et dans l'interface *Absence*. Cependant, l'implémentation de cette méthode est différente selon qu'elle fasse référence à l'interface *Réunion* ou à l'interface *Absence*. Un problème se pose alors si la spécification de l'adaptation est la suivante :

```
1 <component name="Agenda">
2         <service name="Réunion"/>
3         <service name="Absence"/>
4  ...
5 </component>
```

  Une solution pourrait être la concaténation du code de la méthode *afficher* relative à l'interface *Réunion* avec lui de la méthode *afficher* relative à l'interface *Absence*. Le résultat de la nouvelle méthode créé va donc afficher les réunions et les absences d'un individu.

  Si le type de retour est différent de « *void* », il se pose des problèmes de type sémantique. Il est impossible de connaître le bon élément de retour. Une solution possible serait de renommer les méthodes de façon à les distinguer.

- soit les deux méthodes ont les mêmes paramètres et des types de retour différents.
  Dans ce cas, les deux interfaces sont incompatibles. Une première solution consisterait à modifier le nom des méthodes de manière à les distinguer. Il faut alors répercuter (*i.e.* propager) ces modifications dans toutes les méthodes faisant appel à celles-ci ainsi que dans la hiérarchie de classe. Le problème de cette stratégie est que l'on modifie l'accès à un service.

5.3.3 Interfaçage graphique de Static-Scorpio-Tool

Afin de faciliter la spécification de l'adaptation structurelle par la ré-ingénierie, nous avons développé une interface graphique. Cette interface permet à un acteur externe de l'adaptation de spécifier la nouvelle structure interne du composant qu'il souhaite obtenir et d'administrer son déploiement automatique.

Cette interface graphique est constitué de trois onglets ; chacun proposant les services d'un composant de notre outil :

- *un serveur de composants*
  Le premier onglet permet de démarrer un serveur de composants sur le site où l'outil est déployé et de visualiser les opérations réalisées par ce serveur (récupération de composants, chargement de composants, etc.) ;

- *une interface graphique d'adaptation structurelle manuelle*
  Cet onglet permet de réaliser la spécification du résultat de l'adaptation. Pour cela, l'administrateur doit fournir le chemin d'accès au code source du composant à adapter ainsi que le nom du composant. Ce dernier est alors chargé dans l'outil afin de guider l'administrateur qui peut dès lors créer de nouveaux sous-composants en indiquant leur nom, leur site de déploiement et les interfaces qu'ils fournissent. Le site de déploiement doit être sélectionné parmi ceux disposant d'un serveur de composants (la liste des sites est générée automatiquement). Concernant les interfaces fournies, l'administrateur peut visualiser les interfaces disponibles et les services qu'ils proposent et les sélectionner afin de composer la nouvelle structure du composant (voir Figure 5.8).

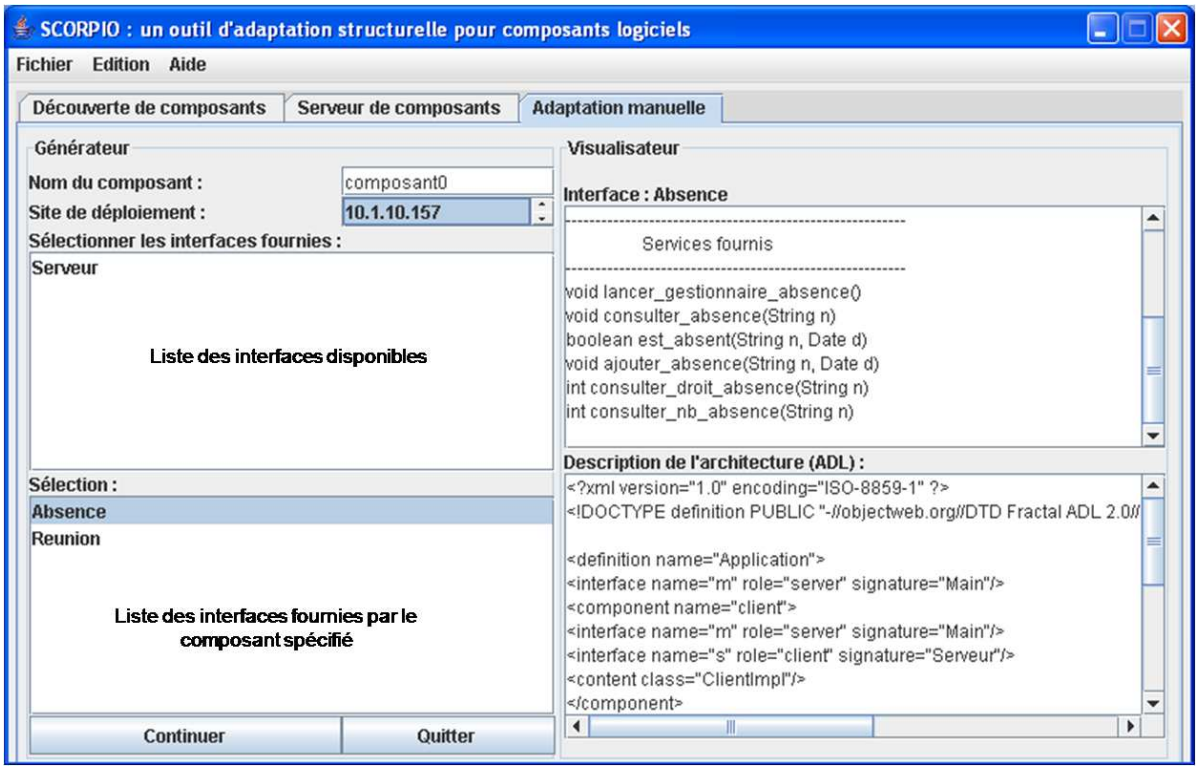

Figure 5.8 – Illustration d'utilisation de Static-Scorpio-Tool pour la spécification du résultat de l'adaptation structurelle

  Une fois la spécification terminée, le code source correspondant est alors généré automatiquement et le résultat est affiché à l'administrateur afin qu'il puisse le contrôler et le valider (voir Figure 5.9). Il peut ensuite procéder à un déploiement automatique du composant adapté ;

- *une interface graphique de découverte de composants*
  Ce dernier onglet permet de visualiser le résultat de la réalisation du déploiement automatique

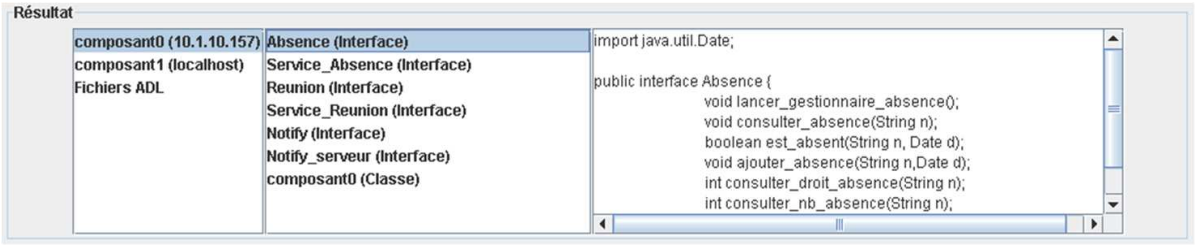

Figure 5.9 – Illustration d'utilisation de Static-Scorpio-Tool pour la visualisation du résultat de l'adaptation structurelle

du composant adapté. Il permet de vérifier que les composants ont bien été déployés sur leur site distant correspondant. Ainsi, il fournit une interface graphique de découverte de composants et de services déployés sur un site. Deux stratégies de recherche sont disponibles : la première stratégie consiste à fournir l'adresse IP du site que l'administrateur souhaite contrôler. L'outil va alors présenter les composants et les services qui y sont déployés ; la seconde stratégie consiste à détecter les sites disponibles d'une infrastructure distribuée (tous les sites sont recensés, y compris ceux qui ne disposent pas de serveur de composants) et à proposer soit d'y déployer un serveur de composants (si un point d'accès au site est disponible) soit d'analyser les composants et les services qui y sont déployés (voir Figure 5.10).

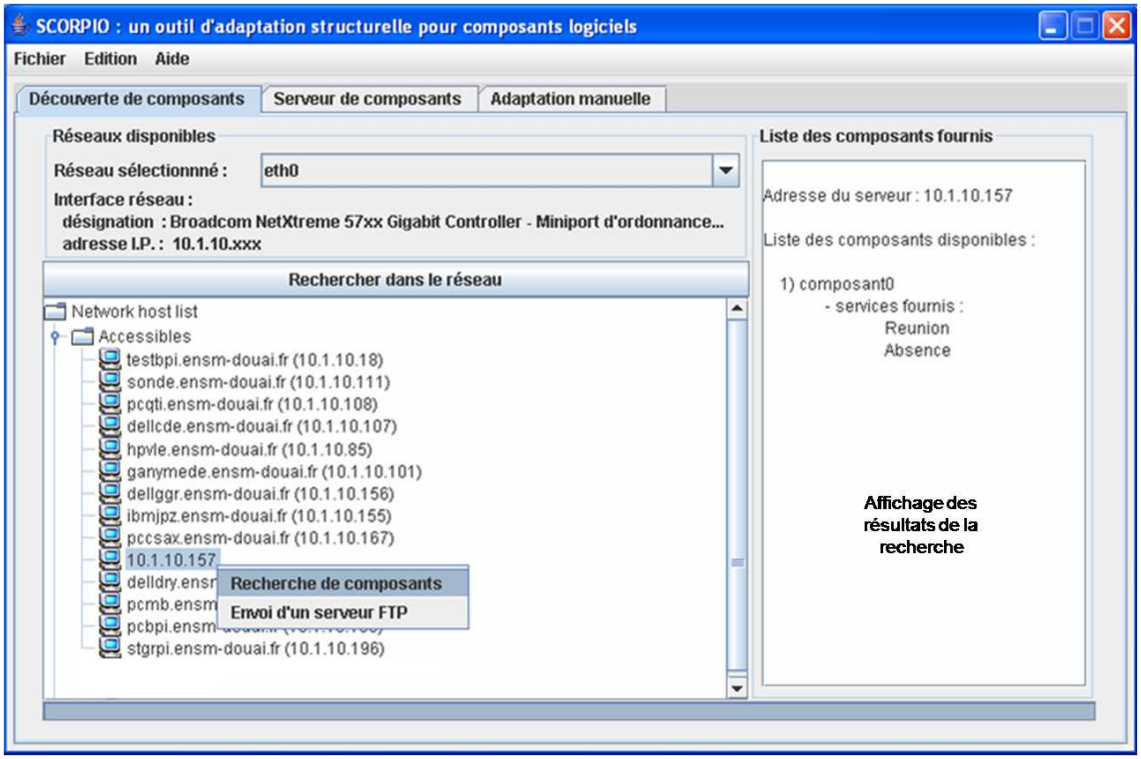

Figure 5.10 – Illustration d'utilisation de Static-Scorpio-Tool pour la recherche de composants logiciels dans une infrastructure distribuée

5.4 Auto-Scorpio-Tool : un outil pour la génération de composants structurellement auto-adaptatifs

Nous avons également implémenté notre approche d'auto-adaptation structurelle dynamique au travers de la réalisation d'un outil permettant de transformer un composant existant en un composant auto-adaptatif, appelé Auto-Scorpio-Tool.

5.4.1 Architecture d'Auto-Scorpio-Tool

Notre prototype permettant de mettre en œuvre l'auto-adaptation structurelle dynamique a pour fonctionnalité de générer à partir du code source d'un composant existant, un composant structurellement auto-adaptatif. Il est basé sur notre prototype d'adaptation structurelle pour la ré-ingénierie de composants existants. En fait, l'architecture de cet outil est quasiment identique à l'outil permettant de réaliser l'adaptation structurelle par la ré-ingénierie à la différence qu'elle inclut un composant destiné à intégrer au composant les mécanismes d'auto-adaptation.

Ainsi, cet outil permet de générer les composants incassables qui vont servir de brique de base à la reconfiguration dynamique du composant en fonction du contexte. De plus, il permet d'introduire dans le composite généré, des mécanismes liés à l'auto-adaptation à savoir : le déclenchement automatique d'une phase d'adaptation, la génération automatique d'une spécification de la structure du composant adaptée au contexte courant ainsi que la réalisation automatique de la reconfiguration et du redéploiement dynamique.

Le composant d'intégration des mécanismes d'auto-adaptation est composé de trois sous-composants ; chacun ayant pour rôle d'intégrer au composant adapté un des sous-composants non fonctionnels que nous avons défini dans notre modèle de composants structurellement auto-adaptatifs (voir Figure 5.11) : un composant de gestion de contexte qui alimente le composant chargé de déclencher une phase d'adaptation et de déterminer une nouvelle structure adaptée au contexte courant ainsi qu'un composant d'adaptation qui va réaliser la transformation de l'architecture afin de la rendre conforme à la nouvelle structure.

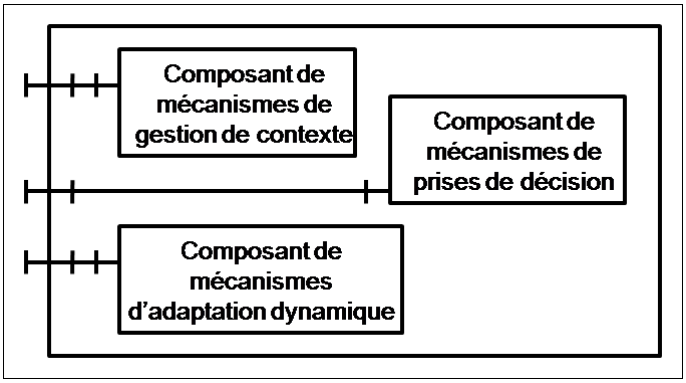

Figure 5.11 – Architecture du composant d'intégration des mécanismes d'auto-adaptation

5.4.1.1 Composant d'intégration des mécanismes de gestion du contexte

Ce composant est chargé d'intégrer au composant adapté des mécanismes de gestion du contexte. Dans notre implémentation, nous avons choisi Context-Toolkit car il présente de nombreux avantages

tels que la portabilité, la facilité d'utilisation, etc. En effet, il permet de gérer simplement des contextes distribués en perpétuelle évolution. Chacun des objets qui le constitue est indépendant des autres, c'est à dire qu'il possède son propre *thread* d'exécution et peut être utilisé localement ou accédé via un réseau (communications TCP/IP). Par ailleurs, il offre des outils performants pour l'interprétation et l'agrégation de données contextuelles.

« *Context-Toolkit* » [110] fournit un environnement permettant de développer des applications sensibles au contexte de par l'utilisation de « *widgets* » qui encapsulent les capteurs. Ces éléments peuvent être organisés comme une architecture hiérarchique qui supporte des processus de base comme l'agrégation des informations. En fait, il existe des interfaces qui permettent de cacher à l'application les opérations d'acquisition d'informations contextuelles.

Ainsi, une application sensible au contexte contient trois types d'objets (voir Figure 5.12 [110]) :

- *des « widgets »*
  Un « *widget* » est un composant qui fournit à une application des informations sur le contexte relatif à leur environnement fonctionnel. Il est défini par ses attributs (*i.e.* morceaux de contexte) et ses services. Ces données sont disponibles pour les autres composants. Il contient un historique. Ces composants cachent les mécanismes de récupération de données liés à l'utilisation de capteurs. Ils sont capables de faire des abstractions de données sur le contexte. Par exemple, si une action de l'utilisateur ou un changement dans l'environnement d'exécution n'est pas significatif alors l'application ne va pas être informée. Ils fournissent les modules réutilisables et paramétrables de capture du contexte. Par exemple, la localisation de l'utilisateur est un module utilisé dans une grande partie des applications sensibles au contexte. Les *widgets* encapsulent l'information liée au contexte et fournissent des méthodes pour y accéder.
  Les serveurs de contexte sont utilisés pour rassembler le contexte entier au sujet d'une entité particulière. Il permet de créer des vues sur les « *widgets* ». Il contient des attributs, des services et un historique ;

- *des interpréteurs*
  Ils sont chargés d'interpréter le contexte. Les interprétations sont destinées à élever le niveau de l'abstraction d'un morceau de contexte ;

- *des agrégateurs*
  Ils sont chargés de rassembler les morceaux multiples d'information sur le contexte qui sont logiquement liés. Le besoin d'agrégation vient en partie de la nature distribuée d'information de contexte.

Un « *widget* » de contexte acquiert toutes les informations requises pour chaque attribut. Pour chaque entité de contexte, un agrégateur collecte les informations nécessaires à partir des « *widgets* ». Les applications peuvent alors consulter les entités de contexte ainsi créées. Des interpréteurs peuvent être utilisés de manière à faciliter, à l'application, la prise en compte de son contexte. Ils sont chargés d'interpréter les informations recueillies sur le contexte. Par exemple, si la bande passante du réseau est de 56 Kbits/s, il doit signaler à l'application que la bande passante est faible. Ils sont également chargés d'assurer les conversions.

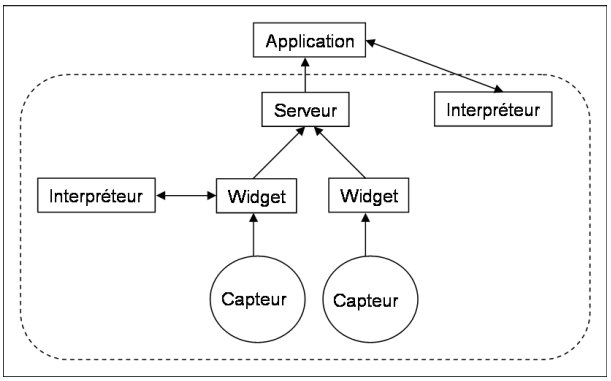

Figure 5.12 – Architecture de *Context-Toolkit*

5.4.1.2 Composant d'intégration de mécanismes de prises de décisions

Pour traiter les informations sur le contexte des diverses entités accessibles au travers de l'infrastructure distribuée, un composant de prises de décisions est intégré au composant adapté. Dans le cadre de notre implémentation, nous avons utilisé un moteur d'inférence de première génération appelé Jess (The Java Expert System Shell) pour réaliser la prise de décisions. Jess fonctionne essentiellement en chaînage avant. Il implémente l'algorithme « *Rete* » pour mettre en œuvre le patron de conception « *matching* ». Rete est un algorithme qui exploite l'espace mémoire disponible pour réduire le temps de calcul. De ce fait, l'espace mémoire utilisé par Jess ne peut être négligé (environs 16Mb). Cependant, Jess contient des fonctionnalités lui permettant de réduire l'espace mémoire utilisé. Mais, cette réduction entraîne une baisse des performances du moteur. De plus, étant donné que Jess utilise une quantité de mémoire non négligeable, ses performances sont étroitement liées à celles de la machine virtuelle Java utilisée et plus particulièrement aux performances de son *garbage collector*.

Jess permet d'établir une séparation de la partie analyse et décision, de la partie action. Ainsi, il donne la possibilité à l'administrateur d'ajouter facilement et dynamiquement des règles sur le contexte sans modifier le code de l'application.

La base de faits contient les informations contextuelles relatives aux conditions d'utilisation de chaque service proposé par le composant à adapter (*i.e.* description des services) ainsi que les données relatives aux nœuds de l'infrastructure distribuée, susceptibles de déployer un fragment du composant initial.

Les règles correspondant aux tâches décrites précédemment (*i.e.* tâches établies dans la stratégie de spécification d'une structure adaptée au contexte), permettent d'inférer sur les faits afin d'associer à chaque interface fournie par le composant adapté, un site de déploiement. Un exemple de règles relatives à la sélection des services en fonction des ressources requises (mémoire disponible, vitesse du processeur et système d'exploitation) est donné dans la figure 5.13. Les règles se composent en deux parties une partie prémisse/condition (partie de droite) et une partie action/conséquence (partie gauche). Elles s'apparentent à un *IF <conditions> THEN <actions>*. De la même manière que nous l'avons évoqué précédemment, nous obtenons une spécification d'une structure du composant adaptée au contexte.

```
1  (defrule site-attribution
2  (service (name ?n) (memory ?rfm) (cpu ?rcp) (os ?ros))
3  (site (adress ?adr) (freememory ?fm) (cpu ?cp) (os ?os))
4  ?fact <- (etat-site (site ?adr) (usedmemory ?mem))
5  (test (<= (+ ?rfm ?mem) ?fm))
6  (test (<= ?rcp ?cp))
7  (test (eq ?ros ?os))
8  (not (service-site (service ?n)))
9  =>
10 (assert (service-site (service ?n) (site ?adr)))
11 (retract ?fact)
12 (assert (etat-site (site ?adr) (usedmemory (+ ?rfm ?mem))))
13 )
```

Figure 5.13 – Exemple de règles définies en utilisant le moteur de décisions Jess

5.4.1.3 Composant d'intégration de mécanismes d'adaptation dynamique

Le composant d'intégration de mécanismes de reconfiguration est chargé de générer le composant d'adaptation que nous avons défini dans le cadre de notre modèle de composants structurellement auto-adaptatifs. Ce composant est constitué de deux sous-composants : un composant de reconfiguration et un composant de redéploiement.

Le composant de reconfiguration permet de restructurer le composant adapté en fonction de la spécification générée par le composant décisionnel. Ce composant est constitué de trois sous-composants :

- *un composant de fusion*
  Ce composant permet de fusionner au sein d'un nouveau sous-composant un ensemble de services qu'il aura préalablement défini. Les ensembles de services sélectionnables correspondent aux services fournis par les composants incassables. Les nouveaux composants seront alors automatiquement générés par encapsulation des composants incassables dans des composites ;

- *un composant d'éclatement*
  Ce composant a été conçu afin de permettre l'éclatement de composants composites qui auront été générés par encapsulation de composants incassables ;

- *un composant de prise de décisions*
  Ce composant utilise les services fournis par les deux sous-composants cités précédemment afin d'obtenir un composant dont la spécification est conforme à la celle fournie.

Le composant d'adaptation contient également un composant de redéploiement chargé de transférer les composants générés lors de la reconfiguration, sur leur nouveau site d'exécution.

5.4.2 Interfaçage graphique d'Auto-Scorpio-Tool

L'interface de notre outil graphique pour générer des composants auto-adaptatifs est similaire à celle présentée dans le cadre de l'adaptation structurelle par la ré-ingénierie de composants existants.

L'administrateur doit spécifier le composant qu'il souhaite rendre auto-adaptatif. Ensuite, il doit sélectionner son site de déploiement primaire (site sur lequel le composant doit être déployé initialement) et fournir les règles d'auto-adaptation dans un langage interprétable par le moteur d'inférence utilisé (ici nous avons utilisé Jess). Dès lors, l'administrateur peut déclencher l'auto-adaptation. Le composant est

rendu conforme à notre modèle de composants structurellement adaptables puis est déployé automatiquement en fonction du contexte.

La figure 5.14 montre un composant appelé *ServeurImpl* que l'administrateur souhaite déployer sur le site dont l'adresse IP est : 10.1.10.159. Ainsi, après avoir chargé le composant *ServeurImpl* dans l'outil, spécifié son site de déploiement et décrit les règles d'auto-adaptation, l'administrateur déclenche le déploiement automatique du composant. Dans le cas montré dans la figure 5.14, le composant ne dispose pas de ressources nécessaires à son déploiement sur le site souhaité : la mémoire disponible est trop faible (26Mo). De ce fait, une adaptation structurelle du composant *ServeurImpl* est donc indispensable. L'outil va alors calculer, en fonction des règles d'auto-adaptation et des données relatives aux sites de déploiement potentiels, une spécification de la structure du composant adaptée à la situation. La spécification ainsi obtenue est montrée dans la figure 5.15. La structure du composant va alors être adaptée de manière à la rendre conforme à cette spécification. Le composant sera ensuite automatiquement déployé.

Par ailleurs, nous avons doté l'application contenant le composant adapté de mécanismes permettant à l'administrateur de visualiser dynamiquement la structure des composants qu'elle contient et de la modifier. L'affichage de la structure des composants est mis en œuvre sous forme d'un arbre dynamique : pour chaque sous-composant, sont indiqués leurs propres sous-composants ainsi que leurs interfaces fournies. L'administrateur peut alors agir sur la structure du composant en fusionnant des sous-composants ou en éclatant des composites qu'il aura créé. L'interface graphique offrant ces services est présentée dans la figure 5.16.

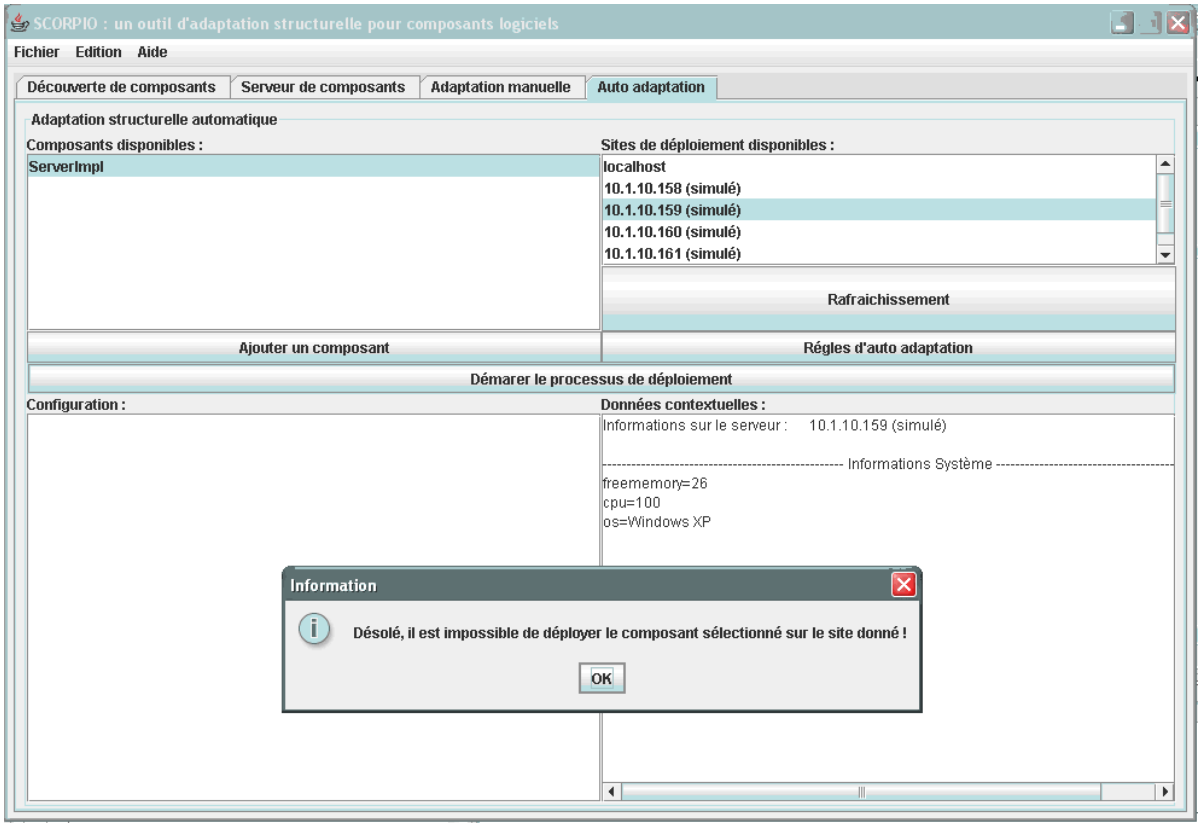

Figure 5.14 – Illustration d'utilisation d'Auto-Scorpio-Tool (1)

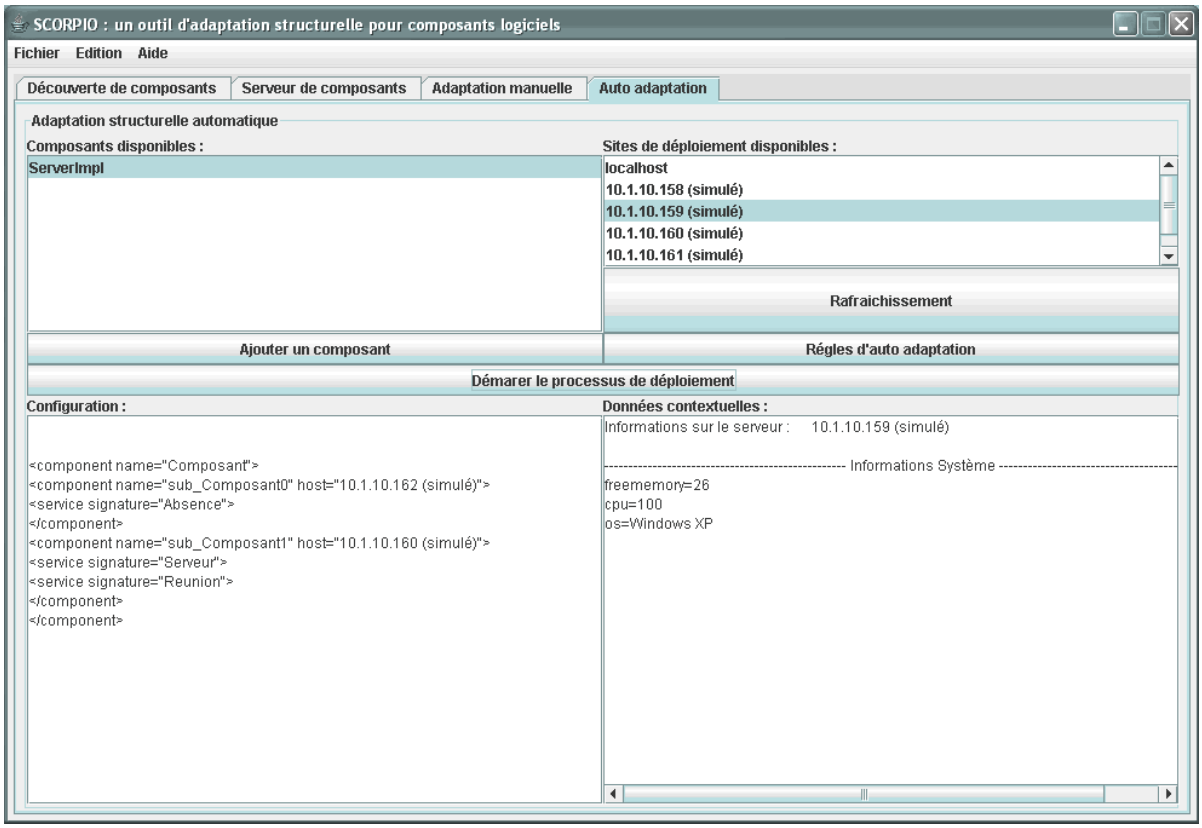

Figure 5.15 – Illustration d'utilisation d'Auto-Scorpio-Tool (2)

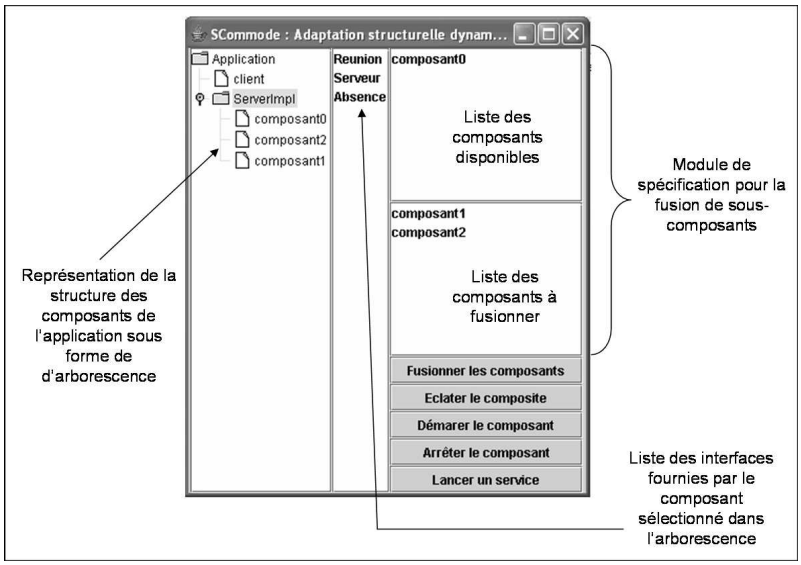

Figure 5.16 – Illustration d'utilisation d'Auto-Scorpio-Tool pour l'adaptation dynamique manuelle

5.5 Expérimentation : Ubibuilding site

Dans le cadre de nos expérimentations, nous avons proposé un scénario mettant en jeu notre approche d'adaptation structurelle de composants logiciels. Ce scénario appelé *Ubibuilding site* repose sur la conception d'une application de « Chantier Ubiquitaire » capable de s'adapter automatiquement à son contexte d'exécution.

5.5.1 Présentation de Ubibuilding Site

5.5.1.1 Vue générale

Notre scénario se déroule dans le cadre d'un chantier de construction où tous les acteurs sont équipés d'entités matérielles mobiles capables de stocker et d'exécuter des composants logiciels, mais dotées de ressources variables (voir Figure 5.17). Ces entités mobiles sont connectées les unes aux autres via une infrastructure distribuée dotée de caractéristiques variables (*i.e.* différents moyens de communication sont mis en jeu pour connecter les différentes machines : réseau filaire, réseau sans fil présentant des caractéristiques variables tels que la portée, la bande passante, etc.). Cependant, elles peuvent être connectées et déconnectées à tout moment. En effet, l'utilisateur peut sortir à tout moment de la zone de couverture du chantier ou bien le matériel peut rencontrer des défaillances (batterie insuffisante, perte de connexion, etc.).

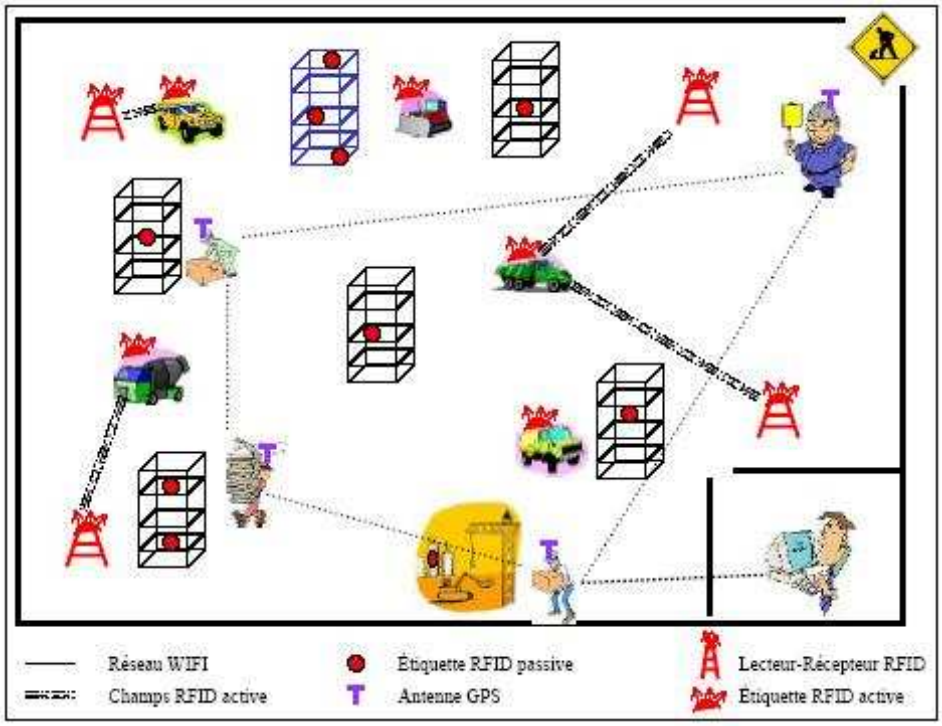

Figure 5.17 – Vue générale du chantier ubiquitaire

Comme nous l'avons évoqué précédemment, il existe une grande diversité d'intervenants, dans le cadre d'un chantier de construction. En effet, les métiers mis en jeu dans le cadre de ce type d'activité

sont très diversifiés. Ils peuvent être classés suivant leur secteur d'intervention. Plus généralement, ils sont répartis en quatre grandes catégories de métiers :

- la première réside dans les phases de préparation et de suivi des travaux (architecte, chef de bureau d'études, dessinateur, géomètre, ingénieur d'études en maîtrise d'ouvrage, maître d'œuvre, etc.),
- la deuxième catégorie est centrée sur le gros œuvre (chef de chantier gros œuvre, charpentier, maçon, soudeur, tailleur de pierre, etc.),
- la troisième catégorie regroupe tous les métiers du second œuvre (carreleur, chauffagiste, couvreur, électricien, menuisier, plâtrier, etc.),
- et enfin la dernière catégorie concerne les finitions et la maintenance du bâtiment (domoticien, miroitier, peintre, moquettiste, ascensoriste, etc.).

On observe également des hiérarchies au sein même de chaque métier. Par exemple, pour les métiers de préparations de travaux, nous pouvons constater la hiérarchie suivante : responsables, assistants principaux, assistants, etc. Concernant les métiers relatifs au gros œuvre, au second œuvre ainsi qu'à la finition et la maintenance, on parle plutôt d'une hiérarchie patron, chef de groupe, ouvriers, etc.

L'objectif de l'application que l'on souhaite mettre en œuvre consiste à fournir à chacun des acteurs l'ensemble des services qui peuvent leur être utile lors de la réalisation de travaux au sein du chantier. Les services que l'application se propose de fournir aux différents acteurs peuvent être divisés en deux catégories :

1. *un ensemble de services indépendants des métiers*
   Ces services sont indépendants des métiers des utilisateurs cibles (par exemple, rechercher un lieu, un objet ou un responsable dans le chantier, appeler un responsable dans le chantier, afficher sa position dans le chantier, consulter la date et l'heure, etc.). Ainsi, ces services devront être accessibles par tous les corps de métiers à tout moment ;

2. *un ensemble de services spécifiques aux corps de métier*
   Ces services correspondent à une cible particulière d'utilisateur. Ainsi, ils ne pourront être accessibles qu'aux utilisateurs exerçant la profession concernée. Par exemple, pour l'architecte, l'application doit lui fournir l'ensemble des services qu'il a besoin (gestion budgétaire, gestion des contacts, gestion du cahier des charges, gestion du planning des travaux, gestion de plans, etc.).

Par ailleurs, nous disposons d'une application fournissant les services susceptibles d'être utilisés par les différents acteurs mis en jeu dans le cadre d'un chantier de construction. Cependant, étant données les propriétés des différents nœuds de l'infrastructure distribuée (*i.e.* ressources limitées, système ouvert, etc.), et de l'application (*i.e.* ressources nécessaires importantes), il ne peut être envisagé de déployer tous les services de l'application sur chaque machine. De ce fait, notre objectif va être de fournir à chaque utilisateur les services qu'il a besoin dans la situation dans laquelle il se trouve tout en garantissant leur continuité de service.

5.5.1.2 Architecture matérielle de Ubibuilding-site

L'architecture matérielle destinée à supporter notre application ubiquitaire est constituée de trois types d'éléments : les objets communicants (*i.e.* entités fixes), les points d'entrée mobiles (*i.e.* entités mobiles) et enfin, les objets passifs.

1. Les objets communicants (*i.e.* entités fixes) : ces machines servent de relais d'informations (*i.e.* nœuds) et de station de stockage de données.

   - Les serveurs : ces machines sont utilisées comme des supports à application. Ils interagissent avec les différentes entités mobiles associées aux agents via un réseau sans fil (*i.e.* généralement *Wifi*). Ils permettent de stocker les données (plans du chantier, liste des employés, matériaux disponibles, agendas, etc.) et sont également destinés à recevoir des sauvegardes. Ils permettent ainsi de conserver un historique du chantier (*i.e.* travaux réalisés, conditions, etc.). Les serveurs ne sont jamais déconnectés.

   - Les bornes *Wifi* : ces machines servent de routeurs entre différentes entités du réseau. Elles sont disposées de manière à couvrir la totalité du chantier et peuvent être utilisées pour la localisation des utilisateurs (*i.e. Wifi* géolocalisé).

2. Les objets passifs : ces éléments regroupent l'ensemble des outils destinés à acquérir et à analyser des éléments du contexte.
   Par exemple, une station météo équipée de divers capteurs permettant l'acquisition de données en temps réel sur les conditions atmosphériques (thermomètre, anémomètre, pluviomètre, baromètre, etc.) est intégrée au réseau. Elle est positionnée à un endroit stratégique du chantier. Celle-ci doit être reliée à un serveur capable d'interpréter ces données afin de fournir aux autres entités du réseau un ensemble de services liés à la météorologie. Par ailleurs, il doit conserver un historique des données ainsi récupérées.

3. Les points d'entrée mobiles : ils sont généralement utilisés comme des outils (*i.e.* passerelles) permettant d'accéder à un réseau généralement sans fil (GSM, WAP, GPRS, *Wifi*, etc.) afin d'accroître la mobilité de l'utilisateur. Ce dernier peut les connecter et les déconnecter du réseau à tout moment. Parmi ces objets, nous pouvons citer les étiquettes RFID, les téléphones portables et autres smartphones, les PDA, les tabletPC, les ordinateurs portables, etc. Par ailleurs, les entités mobiles sont dotées de caractéristiques techniques plus limitées que les serveurs fixes. Par exemple, la capacité de stockage et la vitesse du processeur sont plus faibles. Ainsi, toute application déployée sur ce type de machine doit prendre en compte ses caractéristiques. De plus, ils sont autonomes c'est-à-dire qu'ils fournissent eux-mêmes l'énergie nécessaire à leur fonctionnement. Or, cette énergie n'étant pas illimitée, il est nécessaire d'en tenir compte pour y déployer des applications.

Les entités fixes sont connectées au moyen de réseau filaire et sont disposées au sein du chantier en fonction des besoins contrairement aux entités mobiles qui n'ont pas de position fixe et qui communiquent uniquement par réseau sans-fil (*i.e. Wifi*, GSM/GPRS, *Bluetooth*, infrarouge,etc.). Ainsi, une entité mobile doit être capable de communiquer avec les entités fixes mais également avec les autres entités mobiles présentes sur le chantier. Chaque agent du chantier est équipé d'une entité mobile en

fonction de son métier et de son niveau dans la hiérarchie. Par exemple, un architecte va être équipé d'un TabletPC ou d'un portable car la taille de l'écran figure parmi les plus grandes dans la catégorie des éléments mobiles (*i.e.* afin de faciliter la visualisation de cartes). Chaque objet (bâtiment, matériau, etc.) est équipé d'une puce RFID permettant de l'identifier et de le repérer dans le chantier.

5.5.1.3 Architecture logicielle de Ubibuilding-Site

Pour construire notre application, nous utilisons la technologie des composants logiciels car elle nous permet de manipuler des unités logicielles de haut niveau et ainsi de gagner en réutilisabilité. Par ailleurs, nous disposons de composants existants fournissant des services nécessaires au fonctionnement de l'application. De plus, toutes les étapes du cycle de vie d'une application (*i.e.* conception, déploiement, exécution, etc.) sont prises en compte par le modèle ce qui permet de faciliter son adaptation. Notre application de gestion de chantier ubiquitaire est donc un assemblage de composants pouvant être distribués sur l'ensemble du réseau. Chaque composant fournit un ensemble de services et peut nécessiter pour fonctionner des services fournis par d'autres composants. Les composants fonctionnels (*i.e.* métier) de notre application sont les suivants :

- *un composant de services généraux*
  Ce composant fournit les services indépendants du métier. Il devra être chargé sur toutes les entités mobiles connectées au réseau ;

- *des composants spécifiques à chaque métier*
  A chaque métier est associé un composant logiciel qui va fournir l'ensemble des services correspondant à celui-ci. Ces composants pourront être chargés sur les entités mobiles en fonction du profil de leur utilisateur ;

- *un composant de gestion de travail collaboratif*
  Ce composant fournit un support permettant aux différents acteurs du chantier de communiquer. Il offre différents services dont le but est de permettre la communication audio/vidéo et le partage de supports d'affichages et d'outils au sein du chantier. Ce composant correspond au support de communication fourni dans le cadre de l'application *Com-In-Project* que nous avons présentée dans les chapitres précédents ;

- *un composant gestion de données météorologiques*
  Ce composant fournit des services relatifs à la météorologie (prédictions climatiques, informations sur les conditions météorologiques courantes, etc.) ;

- *des composants de sauvegarde et de gestion d'historiques*
  Ces composants sont chargés de conserver une historique des évènements intervenus dans le cadre du chantier ubiquitaire (par exemple, un historique des conditions météorologiques rencontrées, etc.).

5.5.2 L'adaptation structurelle dans Ubibuilding-site

5.5.2.1 Cas général

A cause de la grande diversité des métiers mis en jeu dans le domaine de la construction en bâtiment, il parait impossible de déployer sur chaque entité mobile l'ensemble des composants de l'application avec tous les services qu'ils proposent car les caractéristiques techniques des différentes entités mobiles sur lesquelles est déployée l'application sont souvent relativement limitées. En effet, les entités mobiles sur lesquelles est exécutée l'application, présentent des caractéristiques très variables. Par exemple, leur capacité de calcul et de stockage peuvent être très réduites. En conséquent, une sélection, en fonction du contexte, des composants ainsi que des services qu'ils proposent devient indispensable afin de garantir le bon fonctionnement de l'application.

La stratégie pour laquelle nous avons opté est la suivante : tous les composants logiciels de l'application sont stockés sur des serveurs fixes en permanence reliés au réseau via des connexions sans-fil couvrant la zone de chantier. Dès lors qu'une entité mobile pénètre dans la zone de couverture du chantier, elle va être automatiquement détectée. Puis, l'application va se déployer automatiquement sur l'unité mobile en fonction du contexte. Étant donné que l'application ne peut pas être déployée dans son intégralité, ses composants doivent être adaptés. Afin de répondre aux attentes des utilisateurs, nous avons choisi une stratégie consistant à partitionner les composants de l'application et ne charger sur l'unité mobile de l'utilisateur que les services nécessaires en fonction du contexte. Les autres services pourront être répartis sur l'infrastructure distribuée disponible.

Par ailleurs, nous avons écarté l'hypothèse de stocker uniquement les composants sur des serveurs dédiés accessibles par l'intermédiaire d'un réseau sans fil couvrant le chantier car cette stratégie implique une faible autonomie des entités mobiles. En effet, si l'utilisateur se trouve dans une zone qui n'est pas couverte par le réseau, ou bien s'il se produit une panne matérielle du réseau ou de l'un des serveurs, aucun service ne sera disponible. De plus, les réseaux sans-fil ont des bandes passantes relativement faibles. Une autre raison qui nous a poussé à ne pas développer cette stratégie réside dans la sécurité d'utilisation des services. En effet, certains utilisateurs peuvent imposer que les informations qu'ils détiennent ne soient pas transmises sur des unités distantes, accessibles par d'autres utilisateurs. Dans ce cas, les traitements de ces données doivent être réalisés sur leur propre machine et donc les services y référant doivent y être déployés.

Pour réaliser le partitionnement des différents composants de l'application, nous utilisons le processus d'adaptation structurelle décrit dans les chapitres précédents, qui va nous permettre d'extraire, à partir des composants de l'application globale, de nouveaux composants correspondant aux besoins de chaque utilisateur qui seront déployés sur les entités mobiles.

5.5.2.2 Un exemple d'adaptation

Considérons les machines appartenant au chef de projet du chantier ubiquitaire. Celui-ci dispose d'un ordinateur de bureau présentant les caractéristiques suivantes : processeur Intel Core 2 Duo, 1 Go de Ram, 120 Go de disque dur, connexion réseau filaire (via le port Ethernet) ; et d'un PDA doté de ressources limitées : PocketPc Toshiba e800 Intel® PXA263 (1,3V) 400 MHz, 128 Mo de SDRAM, connexion réseau Wifi. Le chef de chantier souhaite disposer des services spécifiques à son métier sur son ordinateur de bureau et d'outils de travail collaboratif sur son PDA. Ces derniers vont lui permettre d'organiser son travail (gestion de réunion, etc.) tout en gardant sa mobilité. Ainsi, le composant d'agenda-partagé que nous avons étudié dans le cadre de nos expérimentations précédentes doit être déployé sur le PDA du chef de projet.

Les services spécifiques au métier de chef de projet sont déployés sur sa machine de bureau sans aucun problème car les ressources disponibles le permettent. Cependant, le déploiement des services de travail collaboratif sur son PDA pose problème. En fait, le composant *Agenda-partagé*, ne peut pas être déployé dans son intégralité sur cette machine car sa capacité mémoire se révèle insuffisante (30 Mo sont disponibles alors que 60 Mo sont nécessaires pour garantir la continuité des services de ce composant). Une première solution consisterait à déployer ce composant sur la machine fixe du chef de chantier et à accéder aux services qu'il fournit au travers de protocoles réseaux. Cependant, à cause de la fréquence des déconnections pouvant intervenir dans le cadre du réseau sans fil, une telle solution ne peut être envisagée car le chef de projet souhaite accéder au moins aux services de consultation de son agenda personnel n'importe où et n'importe quand. Une autre solution pourrait être la distribution des services en fonction de ses besoins et des caractéristiques de l'infrastructure de déploiement. Pour cela, il est nécessaire d'adapter la structure du composant car celui-ci est doté d'architecture monolithique ne lui permettant pas de répondre à de telles attentes. Grâce à notre outil Scorpio-Tool permettant de réaliser l'adaptation structurelle, nous avons créé un composant composite à partir du composant monolithique initial. Cette nouvelle structure rend alors possible la distribution des services du composant *Agenda-partagé* en fonction des attentes du chef de projet. Ainsi, le composant composite généré au moyen de notre outil a pu être déployé sur le PDA et sur la machine fixe.

5.6 Conclusion

Dans ce chapitre, nous avons présenté d'une part l'implémentation support de notre approche et d'autre part le scénario ubiquitaire que nous avons développé pour l'expérimenter.

Dans un premier temps, nous avons présenté Static-Scorpio-Tool qui est un prototype permettant de mettre en œuvre la ré-ingénierie structurelle d'un composant existant. Cet outil permet de réaliser le déploiement flexible de composants logiciels existants. Ensuite, nous avons présenté Auto-Scorpio-Tool qui est un prototype permettant de réaliser l'adaptation structurelle dynamique et automatique. En fait, il permet de transformer un composant existant en un composant structurellement et dynamiquement auto-adaptatif. Ces deux outils agissent sur des composants implémentés dans la plate-forme Julia qui est une implémentation Java du modèle de composants Fractal.

Enfin, nous avons présenté un scénario ubiquitaire de construction en bâtiment dans lequel l'utilisation de notre outil d'adaptation structurelle se révèle indispensable pour garantir une continuité des services proposés à l'utilisateur.

Conclusion et perspectives

Résumé des contributions de la thèse

Dans le cadre de cette thèse, nous nous sommes intéressés à la problématique de l'adaptation structurelle de composants logiciels. Dans ce cadre, nous avons étudié différentes facettes de cette problématique :

- **un état de l'art sur l'adaptation logicielle à base de composants**
  Dans un premier temps, une étude de la littérature nous a permis de positionner l'adaptation par rapport à la problématique de la réutilisation et de présenter les concepts et principes généraux de l'adaptation logicielle et plus particulièrement de l'adaptation d'applications conçues à base de composants logiciels. Nous avons alors proposé une classification des travaux existants d'adaptation logicielle suivant différents critères en relation avec la problématique de l'adaptation structurelle. Cette étude nous a permis de montrer les avantages et les inconvénients des approches existantes. Nous avons pu alors constater leurs limitations, notamment en ce qui concerne la non prise en charge de l'adaptation de la structure de composants bien que cette adaptation puisse se révéler indispensable dans de nombreux cas. Par exemple, la création d'applications destinées à être exécutées dans certains types d'environnements peut nécessiter cette adaptation comme nous avons pu le montrer lors de notre étude des environnements ubiquitaires. L'état de l'art que nous avons réalisé a constitué l'état de l'art présenté dans les deux livrables [52, 84] du projet Mosaïques.

- **l'adaptation structurelle par la ré-ingénierie de composants existants**
  Après avoir analysé les travaux existants, nous avons étudié l'adaptation structurelle par la ré-ingénierie de composants logiciels existants pour répondre à certains besoins tels que la réalisation de l'adéquation entre l'architecture logicielle et l'architecture matérielle de par le déploiement flexible de composants logiciels en fonction du contexte. Nous avons alors proposé un processus de transformation structurelle permettant de générer un composant structurellement adapté à partir d'un composant existant. La mise en œuvre de ce processus requiert la définition d'un modèle garantissant la cohérence du résultat de l'adaptation que nous avons présenté. Puis, nous avons enrichi ce modèle en donnant la possibilité aux acteurs de l'adaptation de configurer certaines propriétés liées à la relation de composition et d'introduire dans le composant, des mécanismes de distribution. Ainsi, dans ce cadre, nous avons proposé :

  1. *un processus de transformation structurelle permettant d'obtenir à partir d'un composant existant, un composant structurellement adapté*
     L'adaptation structurelle par la ré-ingénierie de composants existants est réalisée par l'intermédiaire d'un processus de transformation structurelle semi-automatique [15, 20] constitué

de deux étapes nécessitant le code source du composant à adapter. La première étape réside dans la décomposition du composant à adapter : tout d'abord, la nouvelle structure souhaitée doit être spécifiée par un acteur externe de l'adaptation. Le composant est ensuite automatiquement fragmenté en fonction de la spécification fournie. La deuxième étape consiste à recomposer automatiquement le composant tout en tenant compte des dépendances existantes entre les différents fragments générés lors de l'étape précédente : tout d'abord, les composants générés par la fragmentation du composant initial sont assemblés, tout en garantissant la cohérence du comportement du nouvel assemblage. Enfin, cet assemblage est intégré au reste de l'application lors de la dernière étape, par encapsulation dans un composant composite dont le comportement et la structure externe sont identiques à ceux du composant initial ;

2. *un modèle de composants support de l'adaptation structurelle*
   La mise en œuvre de l'adaptation structurelle par la ré-ingénierie nécessite la définition d'un modèle de composants logiciels auquel tout composant structurellement adapté en utilisant notre approche doit être conforme [20] ; ceci afin de garantir le comportement du résultat de notre processus de transformation. Dans ce modèle, nous introduisons notamment les interfaces permettant de garantir la cohérence et l'intégrité des nouveaux composants générés par la fragmentation du composant initial ;

3. *l'introduction de mécanismes de distribution dans un composant structurellement adapté centralisé*
   Nous avons pu constater que la majorité des applications de l'adaptation structurelle nécessite une distribution du composant après la réorganisation de sa structure. Or, le composant obtenu après l'exécution de notre processus de transformation d'un composant existant en un composant structurellement adapté est encore centralisé. Il est donc nécessaire de le rendre distribué. Ainsi, nous avons proposé une approche permettant d'intégrer des mécanismes de distribution à un composant logiciel structurellement adapté [18, 117]. Pour cela, nous avons proposé un modèle de composants structurellement adaptés distribués et un processus permettant d'obtenir un composant conforme à ce modèle à partir d'un composant issu de notre processus de transformation structurelle. Ces propositions sont présentées comme des extensions du modèle de composants structurellement adaptés et du processus de transformation structurelle ;

4. *la configuration de propriétés liées à la relation de composition dans un composant structurellement adapté*
   Afin d'intégrer au composant composite issu de l'adaptation structurelle de nouveaux points de variabilité, en termes de flexibilité et en vue de fournir des facilités pour une éventuelle future adaptation fonctionnelle (*i.e.* adaptation comportementale), nous avons proposé un modèle de composants composites donnant la possibilité à un acteur externe de l'adaptation de configurer certaines propriétés du composite, liées à la relation de composition (encapsulation, cycle de vie, etc.) [14, 17].

- **l'auto-adaptation structurelle dynamique**
  Dans les travaux précédents, la transformation est réalisée de manière statique ; ce qui impose l'arrêt du composant pour réaliser l'adaptation. Néanmoins, il est clair que, dans certains cas, où

la continuité de service est primordiale, l'approche statique que nous avons proposée ne peut répondre à ce type d'attente. En se basant sur ces considérations, nous avons proposé une approche d'adaptation dynamique de composants logiciels.
Cependant, notre processus d'adaptation dynamique requiert en entrée une spécification de la nouvelle structure du composant correspondant aux besoins liés à son utilisation, généralement fournie par l'administrateur de l'application. De plus, le déclenchement d'une phase d'adaptation structurelle est également réalisé manuellement par l'administrateur. Or, dans de nombreux environnements où le contexte d'exécution est en perpétuelle évolution, l'intervention humaine ne peut être envisagée. De ce fait, nous avons proposé une approche d'auto-adaptation de composant logiciel.
Par ailleurs, nous avons pu constater que l'adaptation structurelle d'un composant logiciel pouvait avoir des impacts sur les autres composants d'une architecture logicielle. De ce fait, nous avons proposé une approche permettant de réaliser l'adaptation structurelle au niveau d'architectures logicielles afin de tenir compte des adaptations de chaque composant qu'elle contient. Ainsi, dans ce cadre, nous avons proposé :

1. *l'adaptation structurelle dynamique*
   L'adaptation structurelle dynamique d'un composant logiciel par sa fragmentation en composants élémentaires est basée sur l'introduction d'une nouvelle propriété sur les composants logiciels permettant de les rendre structurellement et dynamiquement adaptables [16, 19]. Dans ce cas, l'adaptation est accomplie en deux phases : la première phase doit être réalisée avant le déploiement du composant. Elle a pour objectif de générer un composant structurellement et dynamiquement adaptable, à partir du composant initial. En fait, un composant répondant à cette propriété est un composant conforme à un format canonique de type composite où, par défaut, chaque interface fournie est réifiée en un sous-composant. Ce résultat est obtenu en se basant sur notre processus de ré-ingénierie de composants existants. La seconde phase, réalisée dynamiquement, réside dans la reconfiguration du composant adaptable, en se basant sur une spécification fournie par l'administrateur de l'application. Cette opération consiste à créer dynamiquement les nouveaux composants spécifiés par l'administrateur et à les redéployer, si nécessaire ;

2. *l'auto-adaptation structurelle dynamique*
   L'auto-adaptation consiste à automatiser la phase de spécification ainsi que la phase de déclenchement d'un processus d'adaptation structurelle [21, 22]. Pour réaliser l'auto-adaptation, nous avons défini une architecture de composants structurellement auto-adaptatifs et un processus d'auto-adaptation géré au niveau des composants. Ainsi, nous introduisons dans les composants des mécanismes chargés d'une part d'acquérir et d'analyser son contexte d'exécution afin de déclencher des phases d'adaptation et de déterminer une structure adaptée, lui garantissant une continuité et une qualité de service ; et d'autre part de modifier automatiquement sa structure afin de la rendre conforme à la spécification obtenue. Nous avons appliqué notre stratégie d'adaptation sur des composants destinés à être exécutés dans des environnements ubiquitaires ;

3. *L'adaptation structurelle d'architectures logicielles*
   L'adaptation structurelle d'une architecture logicielle consiste à coordonner les adaptations structurelles des composants qu'elle contient afin de tenir compte de l'impact de l'adaptation

d'un composant sur les autres composants. Pour répondre à ce besoin, nous avons proposé un modèle et deux stratégies permettant de coordonner l'adaptation de composants logiciels à différents niveaux.

Afin de situer nos travaux par rapport à ceux existants, nous avons appliqué les critères de comparaison que nous avons définis dans le chapitre 1 sur les différentes contributions de cette thèse (voir Tableau 5.1).

Travaux	Type	Environnement	Raisons	Automatisation (acteurs)	Moment (dynamicité)	Facette cible (entités cibles)	Techniques
Adaptation structurelle par la ré-ingénierie	Processus générique	Distribué	Réutilisabilité Flexibilité Adaptativité Performance	Semi-automatique (Outil, administrateur)	Avant l'exécution (Statique)	Structure (Composants)	Transformation
Adaptation structurelle dynamique	Modèle abstrait Processus générique	Distribué	Réutilisabilité Flexibilité Adaptativité Performance	Semi-automatique (Composants spécifiques, administrateur)	Pendant l'exécution (Dynamique)	Structure (Composants)	Reconfiguration
Auto-adaptation structurelle dynamique	Modèle abstrait Processus générique	Ubiquitaire	Réutilisabilité Flexibilité Adaptativité Performance	Automatique (Composants spécifiques)	Pendant l'exécution (Dynamique)	Structure (Composants)	Reconfiguration

Table 5.1 – Bilan des contributions

Perspectives

Trois perspectives majeures se dégagent de notre travail.

Extension vers les composants orientés services

Dans nos choix de travail, nous avons défini les interfaces du composant comme les unités ne pouvant être fragmentées et servant de briques de base à la fragmentation et à la reconfiguration du composant. De ce fait, une perspective de travail pourrait consister à proposer une approche permettant de transformer un composant existant en un composant orienté service.

Dans les modèles de composants orientés services [41], les interfaces fournies par un composant sont généralement assimilées à des services. En fait, un service tel qu'il est défini dans les systèmes orientés services (SOA) est une fonctionnalité réutilisable dont le comportement est défini de façon contractuelle dans un descripteur. Chaque descripteur contient des informations qui décrivent le comportement du service et qui le caractérisent. Ces informations sont utilisées pour réaliser une composition de services. Ainsi, notre modèle de description de services pourrait être étendu afin de le rendre conforme à ce type de descripteur.

La composition de services peut être vue : comme étant une chorégraphie *i.e.* interaction pair à pair entre les services constituant le service composé ; ou bien comme étant une orchestration *i.e.* un client/application-utilisateur coordonne l'exécution des constituants du service composé.

Ainsi, une application construite à partir de services utilise un ensemble de services pouvant être différents entre deux exécutions. En fait, les fournisseurs de ces services ne sont pas figés dans l'application. Le choix du fournisseur d'un service est déterminé à l'exécution ; l'assemblage des services étant réalisé dynamiquement à partir de ces descripteurs. En conséquence, le système mis en jeu doit supporter la découverte et l'assemblage de services pendant l'exécution. Pour obtenir un composant orienté services

à partir d'un composant « standard » , il est nécessaire de lui intégrer des mécanismes de disponibilité dynamique [42].

Une approche dirigée par les modèles pour la restructuration de composants logiciels

Nous avons proposé, dans le cadre de notre approche, un processus d'adaptation structurelle avec l'ambition de le rendre le plus indépendant possible d'un modèle de composants et d'une infrastructure logicielle de déploiement spécifiques. Ainsi, notre approche a été conçue pour permettre l'adaptation de tout type de composant répondant aux caractéristiques que nous avons fixées dans nos hypothèses de travail.

Cependant, nous avons mis l'accent sur la transformation de code en illustrant notre approche avec des composants conçus en utilisant le modèle Fractal et son implémentation Java appelée Julia. Aussi, une des perspectives de notre approche est d'utiliser la transformation de modèles comme base à toutes les transformations réalisées dans le cadre de notre approche.

La transformation de modèle est une méthodologie qui a été proposée dans le cadre de l'ingénierie dirigée par les modèles (IDM, en anglais MDE : *Model-Driven Engineering*). L'IDM est une approche de développement de logiciel qui met la notion de modèle (plutôt que le code) au centre du cycle de développement [10]. Cette approche s'appuie principalement sur l'initiative MDA (*Model-Driven Architecture*), menée par l'OMG (*Object Management Group*). L'initiative MDA vise à organiser le développement dirigé par les modèles en couches, allant des modèles indépendants de toute plate-forme appelés « PIM » (*Platform Independent Models*) aux modèles spécifiques à une plate-forme appelés « PSM » (*Platform-Specific Models*).

Ainsi, pour formaliser nos travaux d'adaptation structurelle suivant une approche IDM, nous devons identifier les modèles mis en jeu. A titre d'exemple, nous pouvons citer comme modèle de composants indépendant de toute plate-forme, notre modèle de composants à restructurer et notre modèle de composants support à la restructuration. Les modèles de composants spécifiques à une plate-forme sont quant à eux, ceux proposés dans la littérature (par exemple, Fractal, EJB, COM). Puis, nous devons proposer des opérations de transformation de modèle permettant de mettre en œuvre nos processus d'adaptation structurelle. Par exemple, nous devons proposer les opérations permettant de transformer le modèle de composants à restructurer vers le modèle de composants structurellement adaptés, puis réaliser les projections vers des modèles de composants spécifiques tels que Fractal.

Formalisation de notre approche par la transformation de graphes

Dans le cadre de notre approche d'adaptation structurelle, nous avons pu constater que la structure d'un composant logiciel pouvait être assimilée à un graphe orienté dont les nœuds représentent les entités structurelles contenues dans le composant et les arcs représentent les relations structurelles entre ces entités. Le code source du composant peut également être assimilé à un graphe [89].

De ce fait, la transformation structurelle d'un composant peut être considérée comme une transformation du graphe représentant sa structure : le graphe initial correspond à la structure du composant avant son adaptation et le graphe résultat correspond à la structure du composant souhaitée. Les opérations de transformation de graphe à appliquer sur le graphe représentant la structure du composant peuvent être extraites de notre processus. Par exemple, lors de notre processus de transformation structurelle de composants existants, l'étape de fragmentation du composant à adapter va consister à partitionner le graphe représentant sa structure en différents sous-graphes ; chacun représentant la structure d'un composant généré. Ensuite, les étapes d'assemblages vont modifier le graphe afin de préserver les invariants fixés

par notre processus d'adaptation structurelle. Ces transformations doivent être répercutées sur le graphe représentant le code afin de préserver le comportement du composant.

Ainsi, la transformation de graphes [91] peut être proposée comme formalisme pour expliciter les opérations de restructuration au niveau de la structure du composant ainsi qu'au niveau de son code source et montrer que ces opérations préservent le comportement du composant.

Annexes

ANNEXE A

Glossaire

A

Adaptation : action qui consiste à rendre un dispositif, des mesures, etc., apte à assurer ses fonctions dans des conditions particulières ou nouvelles.

Adaptation dynamique de composants : adaptation de composants pendant leur exécution.

Adaptation structurelle de composants : modification de la structure d'un composant tout en préservant ses services et son comportement.

Adaptation statique de composants : adaptation de composants qui ne sont pas en cours d'exécution.

ADL (*Architecture Description Language*) : langage fournissant une syntaxe et une sémantique formelle pour modéliser l'architecture d'un système.

Application sensible au contexte : application capable d'acquérir et de traiter des informations sur son contexte d'utilisation.

Application ubiquitaire : application capable d'être exécutée dans un environnement ubiquitaire.

Architecture logicielle : description de haut niveau de la structure ou du comportement d'un système ; elle inclut la description des éléments à partir desquels les systèmes sont construits, les interactions entre ces éléments, les patrons qui guident leur composition et les contraintes sur ses patrons.

Auto-adaptation de composants : adaptation de composants réalisée automatiquement par les composants eux-mêmes.

C

Composant adaptable : composant qui possède la capacité d'être adapté soit par l'intervention d'une entité extérieure à celui-ci (par exemple, une personne physique, un composant, etc.) soit en modifiant lui-même son comportement en fonction des éléments qu'il connaît.

Composant auto-adaptatif : composant capable d'adapter automatiquement sa structure et son comportement en réponse à des variations de son environnement d'exécution.

Composant architectural : élément architectural qui encapsule une fonctionnalité métier.

Composant composite : composant ayant la capacité de contenir, par encapsulation, d'autres composants appelés sous-composants.

Composant non-fonctionnel : composant fournissant des services transversaux à toute application tels que la gestion du contexte ou l'adaptation.

Composant logiciel : unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites sur son contexte. Un composant peut être déployé indépendamment et il est sujet à compositions par des parties tierces.

Composant logiciel COTS : composant logiciel disponible sur étagère.

Composant métier : composant logiciel fournissant les services fonctionnels de l'application le contenant.

Composant monolithique : composant construit comme un seul bloc, ne contenant pas de sous-composants.

Configuration architecturale (topologie) : description de la structure complète d'un système représentée généralement sous forme de graphe connexe regroupant des composants et des connecteurs.

Connecteur : élément d'architecture modélisant de manière explicite les interactions entre un ou plusieurs composants architecturaux en définissant les règles qui gouvernent ces interactions.

Contexte : toutes les informations qui peuvent être utilisées pour caractériser la situation d'une entité. Une entité peut être une personne, un lieu, ou un objet qui est considéré comme ayant un lien avec l'interaction entre un utilisateur et une application incluant l'utilisateur et l'application.

Context-awareness : capacité d'un système à acquérir des informations sur son contexte d'exécution et à les présenter à l'utilisateur (*i.e. context-aware* passif) ou à les prendre en compte pour déclencher automatiquement ou semi-automatiquement des services (*i.e. context-aware* actif).

Continuité de service : propriété d'un service garantissant à l'utilisateur sa disponibilité quel que soit l'évolution de son contexte d'exécution.

D

Déploiement : ensemble d'activités qui suivent l'achèvement du processus de développement d'une application et qui visent à la mettre à la disposition des utilisateurs. Le déploiement couvre toutes les étapes depuis la validation de l'application par le producteur jusqu'à son installation puis sa désinstallation en passant par sa maintenance sur la machine cible.

E

Entité structurelle : tout élément identifiable de manière unique faisant partie de la structure du composant (*i.e.* faisant partie de son espace de noms). Les entités structurelles peuvent être composées d'autres entités structurelles (par exemple, les ports sont composés d'interfaces et les interfaces sont composées de services) ou bien hériter des propriétés d'autre entité structurelle de même type.

Extensibilité : capacité d'un système à être étendu c'est-à-dire à se voir ajouter de nouvelles caractéristiques ou de nouvelles fonctionnalités, sans perte des existantes et dans le meilleur des cas sans modification de code.

I

Intercession : capacité d'un système à agir sur lui-même.

Interopérabilité : propriété d'un système lui permettant de communiquer sans ambiguïté et opérer avec d'autres systèmes, qu'ils soient identiques ou radicalement différents.

Interface : ensemble de points d'entrée d'un système.

Introspection : capacité d'un système à s'observer et à raisonner sur son propre état.

Q

Qualité de service : aptitude d'un service à répondre adéquatement à des exigences, exprimées ou implicites, qui visent à satisfaire ses usagers. Ces exigences peuvent être liées à plusieurs aspects d'un service : son accessibilité, sa continuabilité, sa disponibilité, sa fiabilité, sa maintenabilité, etc.

M

Métamodèle : modèle d'un langage de modélisation. Un métamodèle sert ainsi à exprimer les concepts communs à l'ensemble des modèles d'un même domaine.

Mobilité : propriété d'un système caractérisé par le fait que l'utilisateur puisse continuer d'accéder à l'information fournie par une infrastructure distribuée sans tenir compte de son emplacement.

Modèle : spécification d'un système qui en donne une vue simplifiée pour répondre à des objectifs précis.

Modèle de composants : spécification de composants par la définition de l'ensemble de leurs caractéristiques, la manière dont ils peuvent être assemblés ainsi que leur support à leur exécution.

Modèle de composants hiérarchiques : modèle permettant de définir des composants composites.

P

Point de variabilité : partie d'un système supportant plusieurs mises en œuvre.

Port : ensemble de points d'entrée d'un composant (*i.e* ensemble d'interfaces).

Proximité de service : évaluation des dépendances entre plusieurs services.

R

Ressource logicielle : entité structurelle identifiable de manière unique, possédant un état et dont la persistance est supérieure à celle du service dans lequel elle est utilisée.

Ré-ingénierie : processus qui consiste à redécouvrir la conception des logiciels existants afin d'utiliser cette information pour reconstruire le système existant dans le but de l'améliorer.

Réutilisation logicielle : approche de développement d'applications selon laquelle il est possible de construire une application à partir d'entités logicielles existantes ayant été produites à l'occasion de précédents développements et par des personnes généralement différentes de celles qui conçoivent la nouvelle application.

Réutilisabilité logicielle : capacité d'un système à être utilisé plusieurs fois dans différents contextes.

S

Service : fonctionnalité fournie par une application ou un composant.

Structure de composant : graphe dont les nœuds représentent les entités structurelles contenues dans un composant et les arêtes représentent les relations structurelles entre ces entités.

U

Ubiquité : propriété d'un réseau se caractérisant par la présence d'entités mobiles communicantes, parfois de très petite taille et possédant des caractéristiques techniques très différentes les unes des autres allant du téléphone portable aux capacités réduites (*i.e.* puissance de calcul faible, batterie à faible autonomie, taille de l'écran limitée, capacité de stockage faible, etc.) à l'ordinateur de bureau.

ANNEXE B

Travaux représentatifs de l'adaptation de composants et d'architectures logiciels

Cette annexe recense et présente l'ensemble des travaux que nous avons étudiés en vue de la réalisation de notre état de l'art sur l'adaptation de composant ou d'architectures logicielles. Comme nous l'avons évoqué précédemment, l'adaptation peut agir sur son comportement ou sur sa structure. Pour faciliter sa lisibilité, nous avons classifié, dans cet annexe, ces travaux en trois catégories : les travaux relatifs à l'adaptation du comportement de composants ou d'architectures logiciels, les travaux relatifs à l'adaptation de la structure de composants ou d'architectures logiciels et enfin, les travaux relatifs à l'adaptation à la fois du comportement et de la structure de composants ou d'architectures logiciels.

B.1 Travaux relatifs à l'adaptation du comportement des composants

Dans cette section, nous avons répertorié un échantillon représentatif des approches permettant d'adapter le comportement de composants logiciels ou d'applications conçues à base de composants logiciels.

B.1.1 ACEEL

Les travaux menés dans le cadre du projet ACEEL (*self-Adaptive ComponEnts modEL*) [44, 45] ont pour objectif d'étudier le comportement de composants logiciels dans des environnements où les variations des niveaux de ressources disponibles sont importantes et de proposer des solutions pour adapter de manière optimale ces composants. Ce projet se place dans le cadre d'environnements ubiquitaires et mobiles. Dans ce type d'environnement, les variations de la disponibilité des ressources sont dues essentiellement aux connexions et aux déconnexions des machines et autres fluctuations de la bande passante d'un réseau. L'objectif d'ACEEL est de rendre efficace l'exploitation d'applications usuelles dans des environnements différents. Pour cela, ACEEL se propose de fournir des mécanismes appropriés pour supporter le développement de logiciels adaptatifs pour les environnements mobiles.

Pour permettre l'adaptation des composants logiciels dans ce type d'environnement, ACEEL proposent un nouveau modèle de composants auto-adaptatifs dotés d'un système de détection/notification des variations survenues dans l'environnement ainsi qu'un *framework* Java permettant de créer des composants adaptatifs. La technique d'adaptation consiste à faire s'accommoder l'application aux conditions

de l'environnement (*i.e.* adaptations réactives). Il doit être également possible d'étendre les fonctionnalités existantes d'un composant (*i.e.* adaptations évolutives). L'adaptation est donc réalisée par le logiciel (plus précisément par les composants qu'il contient) pendant l'exécution. Le comportement de l'application va donc être modifié dynamiquement. L'objectif est d'apporter de la flexibilité et d'augmenter les performances de l'application.

Pour mettre en place cette approche, ACEEL propose un nouveau modèle de composants auto-adaptatifs contenant un niveau méta destiné à gérer le contrôle de l'adaptation. Le terme de composant désigne une entité logicielle qui fournit un service particulier via une interface séparée de l'implémentation mettant en œuvre ce service. Les services proposés ne doivent pas être implantés sous forme de boîte noire car l'adaptation ne doit pas être transparente.

Le modèle ACEEL se base sur le patron de conception « *Strategy* » [56] qui consiste à disposer de plusieurs implémentations pour une même interface. Cette technique permet de modifier facilement le comportement d'un composant sans en changer son architecture.

Le modèle proposé par ACEEL est sur deux niveaux : le niveau de base créé en accord avec le patron de conception « *Strategy* » et le niveau méta qui est chargé de réaliser l'auto-adaptation. Ce dernier contient les outils nécessaires pour analyser le contexte et réagir en fonction de ces données pour modifier le comportement de l'application. Les règles pour réaliser l'adaptation sont décrites dans ce niveau. Ainsi, ce méta-niveau s'appuie sur des méthodes de détection et de notification des changements de l'environnement afin de modifier le comportement de l'objet au travers de politiques d'adaptation. Ces politiques sont spécifiés par l'intermédiaire de scripts XML séparés du code métier et pouvant être chargés dynamiquement par des composants dédiés. Ces scripts contiennent deux parties : d'une part des informations sur le cycle de vie de chaque instance et d'autre part, des règles d'adaptation de type $< Evenement > \Rightarrow < action >$. Deux types d'actions sont possibles : soit le paramétrage du composant (*i.e.* modification des valeurs de ses attributs), soit le changement de son comportement (*i.e.* changement d'implémentations parmi celles disponibles).

Le processus d'adaptation est réalisé en cinq étapes : (1) lorsqu'une phase d'adaptation est déclenchée par le gestionnaire d'évènements, (2) les interfaces du composant sont temporairement gelées (*i.e.* stockage des invocations de services du composant). Puis, (3) la politique d'adaptation est interprétée (*i.e.* réalisation de l'adaptation). Ensuite, (4) les interfaces du composant sont dégelées (*i.e.* déstockage des invocations de services du composant) et (5) il peut alors reprendre son activité .

Le modèle de composants ACEEL offre donc un support permettant d'adapter le comportement d'un composant à son environnement. Cependant, il impose un certain nombre de contrainte sur les composants. Tout d'abord, les objets implémentant les comportements possibles d'un composant ne doivent pas avoir d'état spécifique et leur impact sur l'état général du composant doit être le même. Ceci permet d'éviter les problèmes de transfert d'état. Pour s'adapter, les composants doivent être conformes au modèle proposé dès leur conception. Aucun processus ne permet à partir d'un composant existant d'obtenir un composant répondant au modèle ACEEL. Ainsi, une phase de re-développement des composants doit être envisagée. Par ailleurs, le composant proposé ne supporte pas la composition hiérarchique. De ce fait, une adaptation de la structure du composant ne peut être envisageable. Par ailleurs, le concepteur doit définir tous les comportements possibles ainsi que les règles pour passer d'une stratégie à l'autre. L'ajout de règles peut être réalisé dynamiquement ; ce qui n'est pas le cas des comportements. Le rôle des développeurs est donc prépondérant afin de créer des composants auto-adaptables.

B.1.2 Adaptative Component

Boinot at al. [28] définissent un « *adaptative component* » comme étant un composant capable d'adapter son comportement en fonction de ses différents contextes d'exécution. Le modèle proposé dans le cadre de ces travaux a pour objectif d'adapter les composants logiciels de manière automatique pendant l'exécution de l'application. Les composants sont implémentés dans un langage étant une extension de Java. Ainsi, le système fournit un compilateur permettant de générer du code binaire Java standard. Cette extension de Java prévoit la création de nouvelles classes appelées « *adaptation classes* » permettant pour chaque interface de fournir plusieurs implémentations possibles. Chacune de ces implémentations correspond à un certain nombre de conditions d'exécutions qui peuvent être réifiées par d'autres composants. En fait, toutes les informations correspondantes à la définition des différentes stratégies d'adaptation possibles doivent être explicitement spécifiées de manière statique dans les différentes classes d'implémentation des composants. Les différents scénarios d'adaptation doivent donc être définis avant l'exécution de l'application par le développeur.

Les composants fournissent toujours les mêmes services (*i.e.* même interfaces fournies), cependant, le comportement de ces services peut changer en fonction du contexte d'exécution. Le changement d'une implémentation de service par une autre de manière dynamique nécessite un transfert d'état. Pour cela, toutes les implémentations d'un composant donné doivent partager la même représentation des données.

La stratégie d'adaptation mise en œuvre pour la création de composants auto-adaptatifs est similaire à celle utilisée dans le cadre d'ACEEL. Les inconvénients de cette approche sont les mêmes. Seul le comportement des composants peut être adapté et la place du programmeur est prépondérante.

B.1.3 MolèNE

MolèNE (*MObiLE Networking Environement*) [94, 114] est un *framework* dédié à l'auto-adaptation dynamique de composants logiciels dans des environnements mobiles. En fait, l'architecture de Molène comporte deux niveaux : le premier niveau, appelé *MolènNE toolkit* fournit des services indépendants du contexte (*i.e.* application, architecture matérielle et environnement) ; le second niveau, appelé *MolèNE framework* offre des services dépendants du contexte.

Molène propose un système d'observation du contexte d'exécution ainsi que des mécanismes de reconfiguration basés sur les techniques de réflexion et d'introspection de langages orientés objets. L'implémentation de la stratégie d'adaptation ainsi que l'observation du contexte sont réalisées au niveau méta par l'intermédiaire d'un *framework* appelé DNF (*Detection and Notification Framework*). La gestion du contexte est réalisée par des moniteurs de différents niveaux : les BM (*Base Monitor*) sont chargés d'acquérir des données brutes et les HLM (*High Level Monitors*) permettent d'agréger et d'interpréter des données fournies par des BM ou d'autres HLM.

Chaque composant Molène est alors doté d'un contrôleur qui est chargé de traiter les informations transmisent par le gestionnaire de contexte (DNF) et d'appliquer les politiques d'adaptation. Ces politiques sont spécifiées par l'administrateur de l'application. Les opérations de reconfiguration disponibles sont le paramétrage des attributs des composants et le remplacement dynamique de l'implantation d'un composant. La mise en œuvre de l'adaptation est classique : tout d'abord, le composant est désactivé, puis la reconfiguration est alors réalisée, ensuite, l'état du composant avant son adaptation est transféré au nouveau composant et enfin, le composant est réactivé.

B.1.4 Dyva

Dyva [74, 75] est une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels. Elle traite plus précisément de l'adaptation de leur architecture tout en se détachant de tout modèle spécifique. Les raisons qui poussent à l'adaptation sont les suivantes : la correction (*i.e.* si le composant contient une erreur), la performance (*i.e.* si le composant est jugé inefficace), l'adaptativité (si le composant n'est pas adapté à l'environnement) et l'évolution (*i.e.* si le composant ne contient pas certaines fonctionnalités). Dans les trois premiers cas, si l'application détecte que le composant ne répond pas aux attentes, ce dernier doit être remplacé par un autre. Le dernier cas peut être résolu en ajoutant des composants permettant d'assurer les nouvelles fonctionnalités. L'adaptation doit être réalisée pendant l'exécution. L'adaptation consiste à contrôler toutes les communications (*i.e.* entrantes et sortantes) entre les composants. Les messages vont être interceptés puis redirigés. Étant donné que ce type d'opération coûte très cher en performance, seul les messages envoyés vers un composant susceptible d'être adaptés seront interceptés. Pour cela, ils définissent trois types de composants : les composants adaptables définis par le concepteur, les composants non adaptables et les composants auto-adaptables.

Si le composant est adaptable et s'il contient une partie de contrôle alors il réalise son auto-adaptation sinon il faut intercepter les messages et effectuer des redirections.

Le processus d'adaptation est constitué de cinq étapes :

1. Définition des règles de correspondances entre le nouveau et l'ancien composant
   Une propriété du nouveau composant peut correspondre à plusieurs de l'ancien.
2. Passivation des deux composants pour assurer la cohérence
3. Transfert d'état de l'ancien vers le nouveau composant
4. Déconnexion de l'ancien et connexion du nouveau composant
   La reconfiguration est réalisée de manière dynamique :
   - Les adaptateurs qui pointent vers l'ancien composant vont pointer vers le nouveau.
   - Les adaptateurs qui sont pointés par le composant adapté doivent se désabonner de celui-ci pour s'abonner au nouveau.
5. Activation du nouveau composant

B.2 Travaux relatifs à l'adaptation de la structure des composants

Dans cette section, nous avons répertorié un échantillon représentatif des approches permettant d'adapter la structure de composants logiciels ou d'applications conçues à base de composants logiciels.

B.2.1 K-component

K-component [50] est un modèle de composants qui vise à créer des architectures logicielles dynamiques pour concevoir des systèmes adaptatifs. Les composants sont spécifiés dans un langage appelé K-IDL qui est un sous-ensemble du langage de définition d'interfaces IDL-3 [50].

Le modèle K-component est basé sur la réflexion architecturale [40]. Ainsi, l'architecture du logiciel est réifiée afin de pouvoir la modifier dynamiquement. Une telle architecture est présentée comme un graphe orienté et connexe, dont les nœuds représentent des interfaces et sont annotés par le nom du composant qui les contient ; chaque arc représente des connecteurs entre deux interfaces et est annoté par des propriétés de connexion.

L'adaptation proposée dans le cadre de ces travaux consiste à modifier l'architecture du composant en réaction à des changements de son contexte d'exécution (*i.e.* règles déclenchées par des événements). Cette adaptation réside dans la transformation de graphe représentant l'architecture du système. Ces transformations doivent être réalisées tout en garantissant la correction de l'architecture du logiciel. Pour cela, les règles de transformations et leurs conditions sont encapsulées dans des contrats d'adaptation appelés *adaptation constracts*.

Cependant, seulement des annotations peuvent être modifiées. Ce système permet donc uniquement le remplacement de composants ou bien la reconfiguration des propriétés de connexion.

Pour gérer ces transformations, K-component utilise un gestionnaire d'adaptation du niveau méta qui est chargé de maintenir les graphes de configuration et d'implémenter les opérations de reconfiguration ainsi que le protocole de reconfiguration.

Par ailleurs, K-component ne supporte pas l'ajout dynamique de contrats d'adaptation ; ce qui fait de lui un système fermé.

B.2.2 Plasma

Plasma (*PLAtform for Self-adaptive Multimedia Applications*) [80, 81] est un *framework* pour la construction d'applications multimédia auto-adaptatives aux variations des ressources matérielles disponibles. Plasma se base sur le modèle de composants Fractal pour proposer une approche permettant de réaliser des reconfigurations hiérarchiques sur les composants en utilisant un ADL dynamique.

Le *framework* Plasma contient principalement trois types de composants : les composants média, les composants de contrôle et les composants de reconfigurations. Les composants média contiennent le code métier de l'application.

On distingue trois types de composants média qui diffèrent suivant leur niveau de composition : tout d'abord, les composants *Media Primitive* (MP) sont les unités de composition de plus bas niveau. Ils implémentent des fonctionnalités multimédia basiques telles que la transmission UDP ou l'encodage MP3. Chaque composant de ce type est doté d'interfaces entrantes et sortantes typées par le format du flux de données qu'elles sont capables de traiter ; Puis, les composants *Media Composites* (MC) sont des composants composites fournissant des fonctionnalités de plus haut niveau tels que la transmission ou l'encodage. Ils permettent d'encapsuler des composants de type MP ; Enfin, les composants *Media session* (MS) sont des composants composites qui encapsulent des composants MC. Ainsi, ils représentent une configuration de l'application et fournit des interfaces de contrôles de l'assemblage qu'il contient.

Une application construite en utilisant le *framework* Plasma correspond à un graphe de flux dont les nœuds sont des composants média de différents niveaux et les arrêtes modélisent les connections entre les interfaces de ces composants. Il existe deux types de connexions entre composants : d'une part, des connexions primitives utilisées entre deux composants manipulant le même type de média (*i.e.* connexion directe) ; d'autre part, des connexions composites utilisées entre deux composants manipulant de média de types différents. Ces dernières sont chargées de réaliser les conversions entre les médias de manière à assurer les interactions.

Les composants de contrôle et de reconfiguration ont pour rôle de coordonner l'adaptation de l'application à son environnement. Ils peuvent être insérés à n'importe quel niveau de composition comme des sous-composants de MC ou de MS.

L'observation du contexte est réalisée par des composants appelés *Probes*. Ils sont de deux types : d'une part, les *QoS Probes* qui sont chargés de fournir des mesures sur la qualité de service et d'autre part, les *Resource Probes* qui sont chargés de fournir des données sur les ressources globales du système telles que la capacité mémoire disponible.

D'autres composants appelés *sensors* sont utilisés pour déclencher des phases d'adaptation suite à une modification significative du contexte d'exécution. Ces composants sont de deux types : d'une part, les *QoS and Resource Sensors* qui sont associés aux composants *Probes* (*i.e.* déclenchement si la valeur dépasse un seuil fixé) et d'autre part, les *External-event Sensors* qui sont chargés de détecter toute modification du contexte hors ressources systèmes et QoS. Ces derniers requièrent une implémentation spécifique.

Enfin, les composants chargés de réaliser l'adaptation sont appelés *Actuators*. Chaque action de reconfiguration est réalisée au travers d'une interface appelé *AC interface*, permettant d'interroger un composant sur la présence d'attributs, de retrouver la valeur d'un attribut par son nom et de changer sa valeur. Chaque composant peut alors interpréter la modification d'un de ses attributs comme une reconfiguration fonctionnelle (modification d'attributs fonctionnels), structurelle (modification des sous-composants) ou autre (*i.e.* politique d'adaptation).

B.2.3 TranSAT

TranSAT [9] est un *framework* pour l'évolution d'architectures logicielles. TranSAT est construit comme un système de tissage d'aspects au niveau d'une description d'architecture logicielle. Chaque aspect appelé plan prend en charge une préoccupation et vient impacter une architecture logicielle appelée plan de base afin d'y insérer les informations relatives à une nouvelle préoccupation. TranSAT est centré autour du concept de patron d'architecture qui contient l'ensemble des informations relatives à une préoccupation à savoir le plan, le masque de point de jonction et les règles de transformation :

- le plan représente un assemblage de composants mettant en oeuvre les services liés à une préoccupation,
- le masque de point de jonction décrit un ensemble de contraintes sur le lieu sur lequel le nouveau plan peut être intégré. Le masque de point de jonction a un double rôle dans TranSAT : premièrement, c'est le contrat entre le plan de base qui est modifié et le patron d'architecture. Il précise sous quelles conditions ce plan de base et le plan contenu dans le patron peuvent être tissés. Deuxièmement, ses éléments sont utilisés pour décrire les modifications à apporter pour intégrer une nouvelle architecture,
- enfin, les règles de transformation spécifient les modifications à apporter au niveau du plan de base et du plan de ce patron afin de permettre leur tissage créant une architecture logicielle enrichie de la nouvelle préoccupation.

Ainsi, TranSAT permet d'améliorer l'architecture du logiciel ou de maintenir l'architecture du logiciel quand il y a un changement de la fonctionnalité logicielle qui peut invoquer la modification de son architecture.

B.2.4 Equipe (DCSUP)

Les travaux de l'université de Pise (DCSUP) sont essentiellement basés sur une méthodologie formelle pour l'adaptation de composants logiciels [32, 33, 34, 35]. La problématique adoptée est la suivante : comment assurer l'interopérabilité des composants au moment de l'assemblage ? Les problèmes majeurs rencontrés sont essentiellement dus à l'incompatibilité entre les signatures des méthodes. Les IDL se révèlent souvent insuffisants, d'où la nécessité de décrire les comportements des composants.

Les quatre principaux aspects de la méthodologie utilisés sont :

1. Les interfaces de composants : ils sont constitués de deux parties : d'une part les IDL qui vont contenir une description des signatures des méthodes fournies par le composant. Et d'autre part,

une description des comportements exprimés en utilisant des sous-ensembles de π-calculus [93].

2. La spécification des adaptateurs : utilisation d'outils de *mapping*. Ils regroupent les caractéristiques des inter-opérations entre deux composants par la mise en correspondance de leurs comportements. Ils sont dédiés à la gestion des correspondances que ce soit pour les actions ($A \rightarrow B$, etc.) ou pour les paramètres ($A_p \rightarrow B_p$, etc.)
3. La dérivation des adaptateurs : un composant destiné à gérer l'adaptation est généré automatiquement en fonction des spécifications.
4. Les propriétés des adaptateurs : les propriétés des adaptateurs contiennent une spécification formelle permettant de vérifier des propriétés de l'adaptateur ainsi que les contraintes et le comportement de l'adaptateur

B.3 Travaux relatifs à l'adaptation du comportement et de la structure des composants

Nous avons répertorié dans cette section un échantillon représentatif des approches permettant d'adapter à la fois le comportement et la structure de composants logiciels ou d'applications conçues à base de composants logiciels.

B.3.1 Safran

Safran [47] est une extension du modèle composant de Fractal visant à supporter le développement des composants auto-adaptatifs. Safran est basé sur l'introduction d'une extension réflective permettant de modifier de manière transparente le comportement d'un composant en fonction de son contexte d'exécution. Il utilise également la programmation par aspects pour développer l'adaptation comme un aspect et le tisser dynamiquement dans les applications. En fait, Safran est constitué d'un langage dédié permettant de programmer l'aspect d'adaptation sous la forme de politiques réactives, et d'un support d'exécution nécessaire au tissage et à l'exécution de ces politiques (aspects) dans les composants Fractal.

Safran fournit un langage permettant de programmer des politiques d'adaptation réactives ainsi que des mécanismes destinés à associer et dissocier dynamiquement ces politiques à des composants Fractal. Chaque politique d'adaptation comporte un ensemble de règles de type Évènement Condition Action (ECA) qui peuvent être mises à jour pendant l'exécution du composant. Les actions d'adaptation résident dans le réassemblage du composant. Ces actions sont réalisées par les reconfigurations du composant qui sont spécifiés en utilisant un script appelé FScript. Les opérations de reconfiguration consistent à ajouter, supprimer ou échanger des nouveaux composants ou services.

Safran se compose de trois systèmes :

- *FScript*
  Tout d'abord, FScript est un langage procédural simple permettant de programmer les reconfigurations de composants Fractal déclenchées suite à la notification d'un événement endogène ou exogène. Il permet notamment de créer dynamiquement de nouveaux composants, de réaliser de l'introspection et de l'intersession sur l'architecture du composant en manipulant le contenu des composites et les connexions entre interfaces. Il offre des garanties sur la préservation de la consistance de l'état de l'application grâce notamment au maintient de l'intégrité transactionnelle (atomicité, consistance de l'état final, isolation) et à la terminaison en temps borné des reconfigurations.

- *WildCAT*
  WildCAT est un système conçu pour faciliter la création d'applications sensibles au contexte. Son rôle est de réifier le contexte (acquisition et modélisation des données) et d'y donner accès aux applications via une API. Dans le cadre de Safran, il est utilisé par des politiques d'adaptation pour détecter les changements du contexte de l'exécution de l'application qui devrait déclencher des phases adaptations. Le contexte d'une application est modélisé sous la forme d'un ensemble de domaines contextuels organisés en une arborescence de ressources et représentant chacun un aspect particulier du contexte. Les nœuds de cet arbre font référence à des catégories de ressources matérielles et les feuilles correspondent aux ressources matérielles possédant un état décrit par un attribut. Le modèle est entièrement dynamique (*i.e.* possibilité de modifier dynamiquement sa structure et ses paramètres) et très facile d'utilisation. Un système proche des *uri* est utilisé pour accéder aux valeurs des attributs. Cependant, il est relativement simple et ne permet pas de modéliser tout type de contexte.
- *un contrôleur Fractal*
  Un contrôleur Fractal est utilisé pour mettre en place l'auto-adaptation du composant. Il fait le lien entre le contexte modélisé à l'aide de l'outil WildCAT et les opérations de reconfiguration du composant définies en utilisant le langage FScript au travers de règles réactives liées aux politiques d'adaptation. Ces règles suivent le modèle ECA, où les événements sont détectés par WildCAT et les actions sont des reconfigurations définies en utilisant FScript. Le contrôleur d'adaptation permet l'attachement dynamique des politiques de Safran à différents composants de Fractal et il est responsable de leur exécution.

Le modèle auto-adaptatif Safran permet aux composants qui y sont conformes de reconfigurer automatiquement leur structure en fonction du contexte. Cependant, ce modèle reste très lié à un modèle spécifique qui est Fractal. Par ailleurs, il utilise comme entité de base à la reconfiguration les composants existants et ne propose aucune solution pour fragmenter ou regrouper les composants bien que ces opérations puissent se révéler indispensables dans certains cas tels que le déploiement de composants dans des environnements à ressources limitées, etc. De plus, toutes les politiques d'adaptation doivent être définies par l'administrateur de l'application ; ce qui peut se révéler très contraignant dans de nombreuses situations. L'administrateur doit connaître la structure de tous les composants assemblés pour former l'application et doit formuler lui-même les opérations de reconfiguration de la structure de l'application qui vont permettre de l'adapter à son contexte d'exécution.

B.3.2 CASA

Le projet CASA (*Constract-based Adaptative Software Architecture*) [95, 96] développé à l'université de Zurich fournit un *framework* permettant l'adaptation dynamique d'applications à base de composants. CASA est basé sur un système appelé CRS (*CASA Runtime System*) permettant de détecter des changements dans l'environnement d'exécution de l'application et de réagir si ces modifications sont significatives. Les politiques d'adaptation sont définies dans ce qu'ils appellent des contrats. Ainsi, le processus d'adaptation mis en place dans CASA est constitué de trois étapes. Chaque fois que le CRS détecte un changement dans l'environnement d'exécution (Étape 1), il évalue les contrats de l'application en cours d'exécution en fonction des changements détectés (Étape 2). Enfin, le CRS décide d'effectuer ou non une phase d'adaptation en fonction des politiques spécifiées dans les contrats. L'adaptation de l'application peut consister à remplacer dynamiquement des services bas niveau, à tisser dynamiquement ou supprimer un aspect, à modifier certains attributs de l'application ou bien à recomposer dynamiquement l'assemblage de composants.

Dans l'implémentation du modèle de composants défini par CASA, chaque composant est l'instance d'une classe. Ainsi, le remplacement d'un composant par un autre va consister à remplacer une instance de classe par une autre de manière dynamique. Une classe est définie comme adaptable si ses instances peuvent être remplaçables. De plus, CASA permet de définir des ensembles de « classe alternatives » qui sont en fait des collections de classes dont les instances peuvent être dynamiquement interchangées. Chacun de ces ensembles est représenté sous la forme d'une seule classe appelée *handle* définissant les mêmes interfaces que les classes qu'elles représentent (*i.e.* abstraction de l'ensemble des classes qu'elles représentent).

Le processus de remplacement d'un composant par un autre s'effectue en cinq étapes. (1) Tout d'abord, le composant à remplacer est désactivé. Cette étape permet de s'assurer que les autres composants de l'application ne pourront plus accéder à un des services qu'il fournit. (2) Puis, l'exécution du composant à remplacer est suspendue. Pour cela, deux stratégies sont possibles. La première solution appelé *lazy replacement* consiste à attendre la fin de l'exécution des services pour effectuer le remplacement alors que la seconde stratégie appelé *eager replacement* permet de figer les services en cours d'exécution. Cette dernière solution n'est pas applicable dans tous les cas. (3) Ensuite, le composant destiné à remplacer l'existant est créé. (4) Puis, l'état du composant arrêté est transféré sur le nouveau composant. (5) Enfin, ce dernier est activé.

CASA propose donc un système d'adaptation dynamique d'applications conçues à partir de composants orientés objets. L'adaptation peut agir à la fois sur les composants (par exemple en configurant ses attributs) ainsi que sur leur assemblage (par exemple en procédant au remplacement de composants). L'objectif est de permettre à l'application de s'adapter en fonction de son contexte d'exécution (*i.e.* adaptation adaptative) de manière transparente vis-à-vis de l'utilisateur. Cependant, le programmeur occupe une place importante pour préparer l'adaptation car il doit spécifier tous les contrats relatifs au comportement de l'application en fonction du contexte. De plus, il doit implémenter et spécifier les ensembles de classes interchangeables.

B.3.3 MADCAR

MADCAR (« *Model for Automatic and Dynamic Component Assembly Reconfiguration* ») [57, 58] est un modèle abstrait de moteur d'assemblage de composants qui permet la construction d'applications auto-adaptables. Une application auto-adaptable est capable de déclencher, planifier et réaliser automatiquement et dynamiquement le ré-assemblage des composants qui la constituent. Lorsque certains changements de contexte sont détectés tels que l'apparition ou la disparition du réseau, des décisions sont prises automatiquement pour ré-assembler l'application.

L'adaptation « dynamique » consiste à faire passer l'application d'un assemblage de composants à un autre sans nécessairement figer toute l'application. Ce passage est une reconfiguration qui peut comporter des ajouts/suppressions/remplacements de composants, des révisions des connexions entre composants et des changements des valeurs des attributs des composants. Pour ce faire, le concepteur de l'application doit fournir la description des fonctionnalités l'application sous forme de configurations alternatives, c'est-à-dire une spécification de l'ensemble des assemblages valides. Cette spécification est abstraite dans la mesure où elle ne fait pas directement référence aux composants à utiliser. Le choix de l'assemblage à réaliser et des composants à assembler se fait selon une politique d'adaptation, également spécifiée par le concepteur de l'application. La politique d'adaptation est un ensemble de règles qui, selon le contexte d'exécution (par exemple, l'état du réseau), génère un ensemble de contraintes. La résolution de ce problème de satisfaction de contraintes débouche sur l'assemblage le plus approprié au contexte courant.

La séparation entre la description des fonctionnalités de l'application et la description de l'adaptation de l'application facilite leur compréhension/évolution. De plus, les descriptions des fonctionnalités et des politiques d'adaptation sont découplées des composants à assembler (pas de référence directe aux composants), car les composants à sélectionner lors d'un ré-assemblage sont en fait désignés par l'ensemble des contrats qu'ils doivent remplir. Cette modularité simplifie la conception des applications auto-adaptables, mais aussi leur évolution. MADCAR est dit abstrait car il est indépendant de tout modèle de composants. Les hypothèses minimales de l'approche sont : l'homogénéité des composants en terme de modèle, l'auto-documentation des composants par contrats, et l'utilisation de composants d'interaction lorsqu'il faut des connecteurs entre composants. Les contrats documentant les composants spécifient au minimum les attributs paramétrables et les interfaces fournies ou requises. Et, les contrats facultatifs peuvent concerner les coûts des composants en mémoire, en CPU ou en énergie.

Une implémentation fonctionnelle existe en la plate-forme AutoFractal[1], qui est une spécialisation de MADCAR pour le modèle de composants Fractal [39].

B.3.4 CADeComp

CADeComp (*Context-Aware Deployment of COMPonents*) [3] est une plate-forme pour le déploiement adaptatif de composants logiciels dans des environnements mobiles. En fait, CADeComp est conçu à l'aide d'un modèle indépendant de la plate-forme qui est constitué d'un modèle de données et d'un modèle d'exécution. Le modèle de données décrit les méta-informations définies par le concepteur du composant et utilisées pour adapter le déploiement au contexte. Ces méta-informations décrivent le contexte de déploiement ainsi que les règles qui définissent les variations des paramètres de déploiement en fonction de ce contexte. Le modèle d'exécution spécifie les entités qui incarnent des mécanismes d'adaptation en s'appuyant sur des algorithmes qui utilisent ces méta-informations.

Le plan d'adaptation du déploiement d'une application à base de composant consiste à déterminer le choix d'implémentation des composants et à placer automatiquement les instances des composants mis en jeu, en fonction du contexte. L'algorithme de placement a pour objectif de maximiser les ressources disponibles du domaine de déploiement après le placement de ces composants ou le nombre de contraintes faibles satisfaites.

Ainsi, l'objectif de CADeComp est de mettre en place un processus d'adaptation d'architecture en se plaçant dans le cadre d'un système réparti. Le logiciel va devoir prendre les mesures nécessaires pour s'adapter à son contexte d'utilisation au moment de son déploiement. Le contexte est défini de la manière suivante : « Le contexte contient toutes les informations sur l'environnement dans lequel l'application va être déployée ainsi que sur l'utilisateur (préférences, etc.) ». La problématique est donc : Comment aider les applications réparties à se déployer en fonction du contexte ? Pour cela, l'application devra être capable de choisir sa structure, de sélectionner la meilleure implémentation pour chaque instance de composant, de placer les composants qu'elle contient et enfin de configurer ces derniers en fonction du contexte.

Le déploiement de l'application peut se diviser en trois étapes : Tout d'abord, le pré-déploiement (*i.e.* déploiement des composants utilisés pour adapter l'application au contexte), puis la prise de conscience du contexte et enfin le déploiement de l'application adaptée.

Pour réaliser l'adaptation, ils définissent cinq variables de déploiement qui sont :

- le choix de la version d'implémentation,
- le choix de placement du composant (en fonction des ressources par exemple),
- la structure de l'application,

[1] http ://csl.ensm-douai.fr/grondin/AutoFractal.

- les valeurs des propriétés configurables,
- les connexions entre composants.

Deux modèles de données sont nécessaires dans la prise en compte du contexte par l'application :

1. *un modèle de données permettant de définir le contexte (i.e. les descripteurs appropriés au contexte)*
   Les informations relatives au contexte sont récupérées par des capteurs puis analysées pour obtenir des données de plus haut niveau. Chaque élément est décrit par son nom, une référence, la méthode pour l'acquérir en traitant le contexte ainsi que l'intervalle de temps à la fin duquel il faut recalculer ces informations. L'objectif reste de définir les contextes pertinents à savoir ceux qui peuvent affecter le déploiement de l'application. Il existe quatre catégories de contextes qui peuvent affecter le déploiement :
   - les informations relatives à l'environnement cible (*i.e.* dans lequel l'application va être déployée),
   - les informations sur l'utilisateur,
   - les informations spécifiques à l'application.
2. *un modèle de données pour définir les règles nécessaires aux choix durant la phase de déploiement (i.e. les descripteurs des plans de déploiement sensibles au contexte)*
   Il contient une définition des règles d'adaptation qui vont être utilisées pour réaliser l'adaptation de l'application en prenant en compte le contexte. Pour cela, il va être créé un composant adaptateur qui va au déploiement être capable d'assembler l'application.
   Afin de réaliser cette tache, pour chaque instance de composant, le modèle doit spécifier :
   - un ensemble de versions d'implémentations possibles et leurs contextes associés,
   - un ensemble de propriétés qui peuvent avoir des valeurs différentes suivant le contexte,
   - un ensemble de sites susceptibles de recevoir le composant et leur contexte associé de manière à justifier les choix de placement.

Le problème de cette approche est qu'elle nécessite la description de toutes les combinaisons possibles. Or, s'il existe beaucoup de composants sensibles à beaucoup de contextes, l'application devient rapidement impossible à gérer du fait du nombre exponentiel de combinaisons à décrire. Il est donc nécessaire de définir uniquement un ensemble de contextes pour lequel l'instance du composant va être utilisée de façon optimale. Ainsi le modèle de données va permettre de définir un ensemble d'associations : instance de composant $\Leftrightarrow$ contextes habilités + ensemble de connexions possibles suivant le contexte. Il doit être possible de spécifier les règles générales d'adaptation avant chaque déploiement. Par exemple, tous les composants doivent être sur le même site. Par ailleurs, il existe un service de découverte de nom qui permet de trouver le site sur lequel peut être implanté un composant.

Ces travaux s'orientent donc uniquement sur de l'adaptation au moment du déploiement. Or dans un environnement ubiquitaire et mobile, l'environnement d'exécution change en permanence. Ce qui est vrai à l'instant t peut être faux à l'instant t+1. Donc prendre en compte le contexte uniquement au moment du déploiement de l'application peut se révéler très insuffisant. Il est donc indispensable de coupler cette technique avec un processus d'adaptation pendant l'exécution.

B.3.5 Concerto

La plate-forme appelée « Concerto » [46] est développée dans le cadre de travaux sur le déploiement et le support de composants parallèles adaptatifs sur des *clusters*. Ce projet traite de l'adaptation automatique de composant (*i.e.* auto-adaptation) dans des environnements répartis. Les ressources sont des objets Java contenant une description de l'environnement par l'intermédiaire de rapports. Des schémas

de données sont utilisés pour permettre aux composants de percevoir leur environnement d'exécution et les variations de cet environnement.

L'adaptation a deux objectifs principaux : le premier réside dans la performance de l'application (*i.e.* il faut qu'elle exploite au mieux une infrastructure spécifique), le deuxième consiste à privilégier la portabilité (*i.e.* plus grande variété possible).

Le problème réside dans le manque d'infrastructure logicielle générique. Deux solutions sont possibles :

1. création de *clusters* virtuels présentant une interface homogène et masquant les propriétés spécifiques sous-jacentes
2. renforcer l'adaptabilité des composants

Pour s'adapter, une application doit prendre des décisions notamment en fonction de l'infrastructure. Ce processus de prise de décision est le suivant : si un composant a la possibilité d'obtenir des informations relatives à sa plate-forme d'implantation, il va devoir se déployer en tenant compte de ces éléments. Par ailleurs, il faut fournir aux composants des informations sur son environnement tout au long de son déploiement. Ces informations vont être obtenues par l'observation des ressources (*i.e.* entité qu'un composant logiciel pourra être amenée à utiliser au cours de son exécution). Tout objet ressource a un identifiant unique enregistré dans un gestionnaire de ressources (identification, localisation et suivi des ressources grâce aux rapports). Les ressources sont classées suivant un certain nombre de critères de manière à faciliter les recherches.

Un composant développé pour fonctionner dans la plate-forme Concerto doit être constitué d'un ensemble de « *threads* » Java coopérants. Il doit posséder trois interfaces : une interface métier (nom, interface, mise en œuvre), une interface cycle de vie (gestion du déploiement et de l'arrêt du composant) et une interface ressource (*i.e.* il contient des informations sur les ressources utilisées par le composant). Dans la plate-forme, un composant est considéré comme une ressource que l'on peut observer. Chaque composant peut fournir son propre rapport d'observation qui consiste à collecter les informations relatives aux différentes ressources. Il en est de même pour les composants composites qui fournissent une agrégation des rapports des sous-composants. Un composant logiciel doit pouvoir vérifier la présence de ressources ou en découvrir l'existence, interroger la plate-forme sur l'état d'autres ressources et enfin s'informer des ressources disponibles sur un nœud particulier du *cluster*.

Pour réaliser le déploiement d'un composant il faut fournir à la plate-forme une description de la structure du composant (fragments[2] et *threads*), les directives de placement des fragments ainsi que les contraintes imposées pour le déploiement.

B.3.6 Satin

Satin (*SAfety model for componenT adaptatIoNs*) [97] est un modèle abstrait de sûreté d'adaptations. Il se veut indépendant de tout modèle ou plate-forme à base de composants. Il se portent essentiellement sur l'adaptation dynamique (*i.e.* à l'exécution) de composants logiciels et sur la sûreté. Le problème réside dans le fait que l'adaptation entraîne généralement une modification du composant. Un des principaux problèmes traités est relatif à la maintenance de la cohérence pendant une phase d'adaptation. Étant donnée que cette dernière peut être provoquée par l'évolution du contexte d'exécution (*i.e.* interaction entre le composant et le contexte), le comportement d'un composant doit avoir la capacité d'évoluer en permanence. Pour réaliser l'adaptation dynamique, il faut pouvoir modifier la structure du composant et faire évoluer ses interfaces. Le sujet de l'adaptation est donc le composant lui-même.

[2] Sous-ensemble de *threads* d'un même composant.

Certains contrôles sur le comportement d'un composant peuvent se révéler incompatibles donc par conséquent, après adaptation, on peut obtenir des incohérences comme la création de boucles infinies, points non déterministes, etc. La solution qui a été adoptée pour assurer la cohérence de l'adaptation consiste à élargir la notion de type pour prendre en compte des propriétés supplémentaires à vérifier. Pour cela, il est nécessaire d'opter pour une notion de type plus large contenant plus d'informations pour en contrôler l'évolution du comportement d'un composant et en valider sa composition.

Étant donné que les composants sont connectés et déconnectés dynamiquement, un composant doit avoir la capacité de modifier ou d'ajouter des types de messages auxquels il sait effectivement répondre. L'évolution du comportement d'un composant passe par la modification du comportement associé aux messages. Ainsi, il est nécessaire de définir un nouveau modèle de composants supportant ce type d'opération. Le rôle d'un composant permet de décrire les propriétés attendues des entités auxquelles elles sont associées. Il est défini par :

- son nom
- un ensemble de ports qu'il contrôle
- un ensemble de ports fournis (*i.e.* services fournis)
- un ensemble de ports émis (*i.e.* messages envoyés vers d'autres composants)
- un ensemble de rôle des composants avec lesquels il interagit

Par exemple, le rôle d'une entité persistante est de contrôler les messages d'accès en lecture et en écriture et d'archiver les messages envoyés. Dans le modèle proposé, il existe deux types de port : les ports instanciés qui ont pour fonction de récupérer et de traiter un message particulier adressé au composant et les ports génériques qui sont considérés comme un ensemble de port instanciés. Le modèle va consister à associer à chaque composant un rôle pouvant évoluer en permanence du fait de l'adaptation. Toutes les instances d'un même composant ont initialement le même rôle.

L'adaptation va consister à modifier le comportement du composant par le contrôle des messages. Cette opération se fait par l'intermédiaire d'assertions. Au moment de l'adaptation, le système doit être dans un état de stabilité afin que les opérations se déroulent correctement. Ensuite, il faut garantir la cohérence des modifications. Pour cela, il est indispensable d'assurer :

1. la sécurité : une opération d'adaptation ne doit pas entraîner de « *crash* » de l'application.
2. la complétude : une opération d'adaptation doit se terminer.
3. la temporisation : une opération d'adaptation doit être déclenchée au bon moment.
4. la réversibilité : le retour à l'état initial doit être possible de manière à éviter les erreurs. Les propriétés sont destinées à garantir la cohérence de l'adaptation :
5. la cohérence structurelle
   (a) Tout composant requis sur un type donné doit toujours correspondre à ce type.
   (b) Pour utiliser un composant, on doit spécifier son rôle.
6. la cohérence comportementale
   (a) Locale à un composant : compatibilité des contrôles exercés. Un même message ne doit pas être délégué à deux entités différentes.
   (b) Locale à une adaptation : une opération d'adaptation est atomique.
   (c) Globale au réseau : gestion des cycles (propagation) / points non déterministes essentiellement dus à la délégation et à la notification.
7. la cohérence sémantique : la même signature pour une méthode n'implique pas forcément les mêmes résultats. Dans la majorité des cas, deux services ayant le même nom ne peuvent pas se remplacer.

ANNEXE C

Modèles de composants logiciels

Afin d'illustrer notre état de l'art sur l'adaptation de composants ou d'architectures logiciels, nous avons étudié un panel représentatif de ces différentes catégories de modèles de composants. Les modèles de composants que nous avons choisis sont présentés ci-dessous.

C.1 Le modèle JavaBean

Le modèle JavaBeans [62] est un modèle de composants logiciels proposé par Sun Microsystems. Les composants qu'il permet de définir, appelés Beans, sont des composants légers, souvent graphiques. L'objectif principal de ce modèle est de simplifier la construction d'applications à travers des méthodes de composition visuelle. En effet, un outil appelé BeanBox, peut être utilisé pour créer des applications à partir de Beans existants. Lors de l'assemblage, des instances sont créées à partir d'un catalogue de classes de composants.

Un Bean est un objet Java standard qui hérite de la spécification associée à JavaBean. En effet, une classe de type JavaBean doit respecter certaines conventions sur le nommage des méthodes, la construction et le comportement. Le respect de ces conventions rendra possible l'utilisation, la réutilisation, le remplacement et la connexion de JavaBeans par des outils de développement. Les Beans communiquent par évènements : lorsqu'un Bean veut communiquer avec un autre composant, il doit déclencher un évènement qui sera capté par le Bean récepteur au moyen d'écouteurs.

Ce modèle ne fournit aucun mécanisme d'introspection qui lui est propre. Il exploite ceux fournis par le langage Java.

C.2 Le modèle EJB (Enterprise Java Beans)

La technologie EJB [86] proposée par Sun permet de concevoir des composants logiciels côté serveur pour la plate-forme de développement J2EE. En fait, elle définit un modèle abstrait de composants logiciels Java côté serveur qui décrit la structure des composants EJB et des modèles de développement et de déploiement des applications à base de ces composants [102]. Dans le modèle EJB, les composants sont des composants Java non visuels, portables, distribués, réutilisables et déployables. En fait, le modèle de composants EJB est une extension du composant visuel JavaBeans, proposé pour supporter les composants serveurs.

Contrairement aux JavaBeans, un EJB est nécessairement représenté par une interface Java appelée *Remote interface* qui définit une vue cliente de l'EJB. Cette interface définit toutes les méthodes

fonctionnelles qu'un client peut invoquer sur le composant. Un EJB offre également des interfaces d'introspections permettant d'accéder à des méta-données relatives aux instances des composants ainsi que des interfaces d'intercession permettant de gérer le cycle de vie d'un composant. Ces dernières fournissent des services permettant à un client de créer, de rechercher ou bien de détruire une instance de composant.

C.3 Le modèle CCM

Le modèle CCM (*Corba Component Model*) a été introduit par l'OMG (*Object Management Group*) en 2002, au travers la publication d'une spécification CORBA 3.0 [59]. Le modèle CCM est constitué d'un ensemble de modèles qui permet de spécifier des composants, de les implémenter, de les empaqueter, de les assembler et enfin de les déployer dans un environnement distribué. En fait, quatre modèles sont proposés : un modèle abstrait de composants qui permet de décrire un composant CCM au travers d'une extension du langage IDL de CORBA (*Interface Definition Language*) ; un modèle d'implantation qui définit la manière d'implanter un composant à l'aide du langage CIDL (*Component Implementation Description Language*) et du *framework* CIF (*Component Implementation Framework*) ; un modèle de déploiement qui définit comment le composant sera distribué, assemblé et déployé ; et enfin, un modèle d'exécution qui définit la structure et l'utilisation des conteneurs de composants.

Un composant est défini par un ensemble d'attributs et de ports. Les attributs de composants sont utilisés pour la configuration au déploiement ou à l'exécution. Les ports définissent les points de communication des composants et sont utilisés pour leur invocation et leur interconnexion. Il existe quatre catégories de ports : les facettes et les réceptacles qui sont des interfaces fonctionnelles synchrones fournies et requises, et les sources et les puits d'évènements qui sont des interfaces fonctionnelles asynchrones fournies et requises. Ainsi, l'assemblage de composants CCM réside dans la création de canaux de communication synchrones entre les facettes et les réceptacles ; et asynchrone entre les sources et les puits définis par les composants à assembler.

Le modèle CCM propose également des mécanismes d'introspection qui fournissent des informations sur le type de composant, les ports et les connexions entre les composants et des outils de gestion de ports utilisés pour établir ou détruire des connexions entre composants.

C.4 Les modèles COM, DCOM et COM+

Le modèle COM (*Component Object Model*) [31] a été proposé au début des années 1990 par Microsoft. Il a pour objectif de résoudre différents problèmes d'interopérabilité au niveau binaire entre des composants COTS [67]. Un composant COM est essentiellement une entité binaire pour laquelle sont définis une interface et son mode d'interaction. Une interface, identifiée de façon unique est décrite dans un langage de description spécifique (IDL). Elle contient un ensemble de méthodes. Le contenu des interfaces ne peut pas évoluer une fois que la description a été publiée. Cependant, un composant COM peut implémenter simultanément plusieurs versions d'une même interface (utilisation du patron de conception « *Strategy* » [56]).

Chaque composant fournit des mécanismes d'introspections permettant de détecter la présence d'une interface particulière ou bien pour voir si le client implémente une nouvelle version d'une interface.

Le modèle COM a été étendu à plusieurs reprises. La première extension appelé DCOM a été proposée afin de prendre en compte des mécanismes de distribution des composants (il permet la communication entre composants situés sur des machines différentes). Ensuite, le modèle COM est devenu

COM+ sous Windows 2000. Il intègre de nouvelles fonctionnalités en matière de gestion des aspects non fonctionnels tels que la gestion des transactions.

C.5 Le modèle Fractal

Fractal [39] est un modèle de composants académique développé par l'INRIA[1] et distribué dans le cadre du consortium ObjectWeb.

Fractal est un modèle de composants modulaire et extensible qui peut être utilisé pour implémenter, déployer ou reconfigurer des systèmes ou des applications. L'objectif de Fractal est de fournir un modèle de composant visant à réduire les coûts de développement, de déploiement et d'entretien de systèmes logiciels. Le modèle Fractal est hiérarchique (les composants peuvent être encapsulés dans d'autres composants appelés composites), réflexif (les composants disposent de mécanismes d'introspection et d'intercession fournis par des interfaces dédiées) et ouvert (de nouveaux services non fonctionnels peuvent être associés à des composants au travers de leur membrane de contrôle). Il existe différentes implémentations du modèle de composant Fractal : à titre d'exemple, nous pouvons citer Julia [38] qui est une implémentation Java et Fractalk [30], une implémentation Smalltalk.

Pour plus de détails sur le modèle de composants Fractal et son implémentation Java, voir section 5.2.

[1] Institut national de recherche en informatique et en automatique. http ://www.inria.fr/

Bibliographie

[1] Gregory D. ABOWD, Christopher G. ATKESON, Jason HONG, Sue LONG, Rob KOOPER et Mike PINKERTON.
Cyberguide : a mobile context-aware tour guide.
ACM Wireless Networks, 3(5) :421–433, 1997.

[2] Gregory D. ABOWD, Anind K. DEY, Peter J. BROWN, Nigel DAVIES, Mark SMITH et Pete STEGGLES.
Towards a better understanding of context and context-awareness.
In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing (HUC '99)*, pages 304–307, London, UK, 1999. Springer-Verlag.

[3] Dhouha AYED.
Déploiement sensible au contexte d'applications à base de composants.
Thèse de Doctorat, Institut National des Télécommunications, France, 2005.

[4] Dhouha AYED, Chantal TACONET et Guy BERNARD.
Component-oriented approaches to context-aware systems.
In *Proceedings of the ECOOP'04 Workshops*, 2004.

[5] Dhouha AYED, Chantal TACONET, Nawel SABRI et Guy BERNARD.
Context-aware distributed deployment of component-based applications.
In *Proceedings of the OTM Workshops*, pages 36–37, 2004.

[6] Henri E. BAL, M. Frans KAASHOEK et Andrew S. TANENBAUM.
Orca : A language for parallel programming of distributed systems.
IEEE Transactions on Software Engineering, 18(3) :190–205, 1992.

[7] Matthias BALDAUF, Schahram DUSTDAR et Florian ROSENBERG.
A survey on context-aware systems.
International Journal of Ad Hoc and Ubiquitous Computing, forthcoming, 2004.
Disponible à l'adresse `citeseer.ist.psu.edu/baldauf04survey.html`.

[8] Dusan BALEK.
Connectors in Software Architectures.
Thèse de Doctorat, Charles University, Czech Republic, 2002.
Disponible à l'adresse `citeseer.ist.psu.edu/balek02connectors.html`.

[9] Olivier BARAIS, Eric CARIOU, Laurence DUCHIEN, Nicolas PESSEMIER et Lionel SEINTURIER.
Transat : A framework for the specification of software architecture evolution.
In *Proceedings of the 1st International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, Oslo, Norway, 2004.

[10] Franck BARBIER.
UML 2 et MDE : Ingénierie des modèles avec études de cas.
Édition Dunod, 2005.

[11] Louise BARKHUUS et Amind DEY.
Is context-aware computing taking control away from the user ? three levels of interactivity examined.

In *Proceedings of the the 6th International Conference on Ubiquitous Computing (UbiComp)*, 2003.
Disponible à l'adresse citeseer.ist.psu.edu/article/barkhuus03is.html.

[12] Gautier BASTIDE.
A refactoring-based tool for software component adaptation.
In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 315–318, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Commode : un modèle pour l'adaptation structurelle de composants logiciels.
In *Proceedings of the french-speaking Workshop OCM-LMO*, 2005.

[14] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Adaptation of monolithic software components by their transformation into composite configurations based on refactoring.
In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, pages 368–375, 2006.

[15] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Adapting software components by structure fragmentation.
In *Proceedings of the 2006 ACM symposium on Applied computing (SAC)*, pages 1751–1758, New York, NY, USA, 2006. ACM Press.

[16] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Dynamic adaptation of software component structures.
In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 404–409, 2006.

[17] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Restructuration de composants logiciels : une approche d'adaptation structurelle statique basée sur la refactorisation de code orienté-objet.
In *Proceedings of the french-speaking Workshop OCM-SI (INFORSID 2006)*, 2006.

[18] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Transformation d'un composant logiciel centralisé en un composant logiciel distribué.
In *Proceedings of the french-speaking Workshop « Journées Composants »*, pages 25–36, 2006.

[19] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Auto-adaptation de composants logiciels : Prise en compte du contexte d'exécution pour l'auto-adaptation structurelle de composants logiciels.
In *Proceedings of the french-speaking Workshop GEDSIP (INFORSID 2007)*, 2007.

[20] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Restructuration de composants logiciels : une approche d'adaptation structurelle de composants logiciels monolithiques basée sur leur refactorisation.
RSTI - L'Objet, 13(1) :81–116, 2007.

[21] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Software component re-engineering for their runtime structural adaptation.
In *Proceedings of the 31st Annual IEEE International Computer Software and Applications Conference (COMPSAC - 2007)*, 2007.

[22] Gautier BASTIDE, Abdelhak-Djamel SERIAI et Mourad OUSSALAH.
Software component reengineering for their context-aware deployment in ubiquitous environments.

In *Proceedings of the 3rd International ERCIM Symposium on Software Evolution*, 2007.

[23] Nicolas BELLOIR.
Composition conceptuelle basée sur la relation Tout-Partie.
Thèse de Doctorat, LIUPPA (Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour), France, 2004.

[24] Jérôme BESANCENOT, Michèle CART, Jean FERRIÉ, Rachid GUERRAOUI, Philippe PUCHERAL et Bruno TRAVERSON.
Les systèmes transactionnels : concepts, normes et produits.
Éditions Hermès-Lavoisier, 1997.

[25] Antoine BEUGNARD, Olivier CARON, jean PHILIPPE THIBAULT et Bruno TRAVERSON.
Assemblage de composants par contrats. le modèle de composants accord.
RSTI - L'Objet, 11(4) :11–16, 2005.

[26] Gorden S. BLAIR et Jean-Bernard STEFANI.
Open Distributed Processing and Multimedia.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[27] Barry W. BOEHM.
Improving software productivity.
Computer, 20(9) :43–57, 1987.

[28] Philippe BOINOT, Renaud MARLET, Jacques NOYE, Gilles MULLER et Charles CONSEL.
A declarative approach for designing and developing adaptive components.
In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, page 111. IEEE Computer Society, 2000.

[29] Jan BOSCH.
Superimposition : A component adaptation technique.
Information and Software Technology, 41(5) :257–273, 25 March 1999.
Disponible à l'adresse `citeseer.ist.psu.edu/281749.html`.

[30] Noury BOURAQADI, Houssam FAKIH, Guillaume GRONDIN et Michaël PIEL.
Fractalk : Fractal components in smalltalk, 2006.
Disponible à l'adresse `http ://csl.ensm-douai.fr/FracTalk`.

[31] Don BOX.
Essential COM.
Addison-Wesley, 1998.

[32] Andrea BRACCIALI, Antonio BROGI et Carlos CANAL.
Systematic component adaptation.
Electronic Notes in Theoretical Computer Science, 66(4), 2002.

[33] Andrea BRACCIALI, Antonio BROGI et Carlos CANAL.
A formal approach to component adaptation.
Journal of Systems and Software, 74(1) :45–54, 2005.

[34] Antonio BROGI, Carlos CANAL et Ernesto PIMENTEL.
Behavioural types and component adaptation.
In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology*, pages 42–56, 2004.

[35] Antonio BROGI, Carlos CANAL et Ernesto PIMENTEL.
Component adaptation through flexible subservicing.
Science of Computer Programming, 63(1) :39–56, 2006.

[36] Peter J. BROWN, D. Bovey JOHN et Xian CHEN.
Context-aware applications : from the laboratory to the marketplace.
IEEE Personal Communications, 4(5) :58–64, October 1997.
Disponible à l'adresse http ://www.cs.kent.ac.uk/pubs/1997/395.

[37] Eric BRUNETON.
Fractalrmi tutorial, 2006.
Disponible à l'adresse http ://fractal.objectweb.org/fractalrmi/.

[38] Eric BRUNETON.
Julia tutorial, 2006.
Disponible à l'adresse http ://fractal.objectweb.org/tutorials/julia/.

[39] Eric BRUNETON, Thierry COUPAYE et Jean-Bernard STEFANI.
Recursive and dynamic software composition with sharing.
In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, 2002.
Disponible à l'adresse citeseer.ist.psu.edu/bruneton02recursive.html.

[40] Walter CAZZOL, Andrea SAVIGNI, Andrea SOSIO et Francesco TISATO.
Architectural reflection : Bridging the gap between a running system and its architectural specification.
In *Proceedings of the Reengineering Forum '98*, 1998.

[41] Humberto CERVANTES.
Vers un Modèle à Composants Orienté Services pour supporter la Disponibilité Dynamique.
Thèse de Doctorat, Université Joseph Fourier Grenoble, France, 2002.

[42] Humberto CERVANTES et Richard S. HALL.
Autonomous adaptation to dynamic availability using a service-oriented component model.
In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Ned CHAPIN, Joanne E. HALE, Khaled Md. KHAM, Juan F. RAMIL et Wui-Gee TAN.
Types of software evolution and software maintenance.
Journal of Software Maintenance, 13(1) :3–30, 2001.

[44] Djalel CHEFROUR.
Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique.
Thèse de Doctorat, Université de Rennes I, France, 2005.

[45] Djalel CHEFROUR et Françoise ANDRÉ.
Aceel : modèle de composants auto-adaptatifs. application aux environnements mobiles.
In *Proceedings of the french-speaking Workshop « Systèmes à composants adaptables et extensibles »*, Grenoble, France, October 2002.

[46] Luc COURTRAI, Frédéric GUIDEC et Yves MAHÉO.
Concerto : gestion de ressources pour composants parallèles adaptables.
In *Proceedings of the french-speaking Workshop GRID'2002*, pages 41–53, Aussois, France, December 2002.

[47] Pierre-Charles DAVID.
Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation.
Thèse de Doctorat, École des Mines de Nantes, France, 2005.

[48] Serge DEMEYER, Sander TICHELAAR et Patrick STEYAERT.
FAMIX 2.0 - the FAMOOS information exchange model.
Rapport technique, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.
Disponible à l'adresse citeseer.ist.psu.edu/demeyer99famix.html.

[49] InstallShield DEVELOPER.
Web site.
Disponible à l'adresse http ://www.installshield.com/isd/.

[50] Jim DOWLING et Vinny CAHILL.
The k-component architecture meta-model for self-adaptive software.
In *Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[51] Stephane DUCASSE et Serge DEMEYER.
The famoos object-oriented reengineering handbook, 1999.
Disponible à l'adresse http ://www.iam.unibe.ch/ famoos/handbook/.

[52] Laurence DUCHIEN et AL.
Projet cper mosaiques (modèles et infrastructures pour applications ubiquitaires), rapport final, programme tac 2004-2007, thème 2.1 logiciel pour la communication ustl/uvhc/emd/inrets.
Disponible à l'adresse http ://www2.lifl.fr/Mosaiques/.

[53] Jean-Marie FAVRE.
Une approche pour la maintenance et la ré-ingénierie globale des logiciels.
Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, 1995.

[54] Régis FLEURQUIN et Chouki TIBERMACINE.
Une assistance pour l'évolution des logiciels à base de composants.
RSTI - L'Objet, 13(1) :9–44, 2007.

[55] Martin FOWLER, Kent BECK, John BRANT, William OPDYKE et Don ROBERTSHOR.
Refactoring : Improving the Design of Existing Code.
Addison-Wesley Object Technology Series, 1999.

[56] Erich GAMMA, Richard HELM, Ralph JOHNSON et John VLISSIDES.
Design Patterns : Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional, 1995.

[57] Guillaume GRONDIN, Noury BOURAQADI et Laurent VERCOUTER.
Assemblage Automatique de Composants pour la Construction d'Agents avec MaDcAr.
In *Proceedings of the french-speaking Workshop on Multiagents and Components (JMAC 2006)*, pages 39–48, Nîmes, France, 21-21 mars 2006. École des Mines d'Alès .

[58] Guillaume GRONDIN, Noury BOURAQADI et Laurent VERCOUTER.
MaDcAr : an Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications.
In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, volume 4063 of *Lecture Notes in Computer Science*, pages 360–367. Springer, 2006.

[59] Object Management GROUP.
Corba components specification, version 3.0, omg tc document formal/2002-06-65, 2002.
Disponible à l'adresse http ://omg.org/cgi-bin/doc?formal/2002-06-65.

[60] Gwladys GUZELIAN, Corine CAUVET et Philippe RAMADOUR.
Conception et reutilisation de composants : une approche par les buts.
In *Proceedings of the french-speaking Conference INFORSID 2004, Biarritz*, 25–28 mai 2004.

[61] HACHETTE.
Le dictionnaire universel francophone en ligne, 2004.
Disponible à l'adresse http ://www.francophonie.hachette-livre.fr.

[62] Graham HAMILTON.
Javabeanstm, version 1.01. sun microsystems, inc., August 1997.
Disponible à l'adresse http ://java.sun.com/products/javabeans/docs/.

[63] George T. HEINEMAN.
A model for designing adaptable software components.
ACM SIGSOFT Software Engineering Notes, 25(1) :55–56, 2000.

[64] George T. HEINEMAN et Helgo M. OHLENBUSCH.
An evaluation of component adaptation techniques.
Rapport technique WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute, February 1999.
Disponible à l'adresse citeseer.ist.psu.edu/408556.html.

[65] Karen HENRICKSEN, Jadwiga INDULSKA et Andry RAKOTONIRAINY.
Modeling context information in pervasive computing systems.
In *Proceedings of the 1st International Conference on Pervasive Computing (Pervasive '02)*, pages 167–180, London, UK, 2002. Springer-Verlag.

[66] Nathalie HERNANDEZ.
Ontologies de domaine pour la modélisation du contexte en recherche d'information.
Thèse de Doctorat, Université Paul Sabatier de Toulouse, France, 2005.

[67] Scott HISSAM, Robert SEACORD et Grace A. LEWIS.
Building systems from commercial components.
In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 679–680, New York, NY, USA, 2002. ACM Press.

[68] Jason I. HONG.
The context fabric : an infrastructure for context-aware computing.
In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '02)*, pages 554–555, New York, NY, USA, 2002. ACM Press.

[69] Galen C. HUNT et Michael L. SCOTT.
The coign automatic distributed partitioning system.
In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 187–200, 1999.

[70] Urs HÖLZLE.
Integrating independently-developed components in object-oriented languages.
In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, pages 36–56, London, UK, 1993. Springer-Verlag.

[71] Vikram JAMWAL et Sridhar IYER.
Bobs : breakable objects.
In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming (OOPSLA), systems, languages, and applications*, pages 98–99, New York, NY, USA, 2005. ACM Press.

[72] S.C. JOHNSON.
Hierarchical clustering schemes.
Psychometrika, 32 :241–245, 1967.

[73] Ralph KELLER et Urs HÖLZLE.
Binary component adaptation.
In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECCOP '98)*, pages 307–329, London, UK, 1998. Springer-Verlag.

[74] Abdelmadjid KETFI.
Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels.
Thèse de Doctorat, Université Joseph Fourier, 2004.

[75] Abdelmadjid KETFI, Noureddine BELKHATIR et Pierre-Yves CUNIN.
Adaptation dynamique concepts et expérimentations.
In *Proceedings of the 15th International Conference on Software and Systems Engineering and their Applications ICSSEA'02*, 2002.

[76] Abdelmadjid KETFI, Noureddine BELKHATIR et Pierre-Yves CUNIN.
Automatic adaptation of component-based software : Issues and experiences.
In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, 2002.

[77] Gregor KICZALES, John LAMPING, Anurag MENHDHEKAR, Chris MAEDA, Cristina LOPES, Jean-Marc LOINGTIER et John IRWIN.
Aspect-oriented programming.
In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
Disponible à l'adresse citeseer.ist.psu.edu/kiczales97aspectoriented.pdf.

[78] Mari KORKEA-AHO.
Context-aware application survey.
Rapport technique, Helsinki University of Technology, 2000.
Disponible à l'adresse users.tkk.fi/ mkorkeaa/doc/context-aware.html.

[79] Bülent KÜÇÜK, M. Nedim ALPDEMIR et Richard N. ZOBEL :.
Customizable adapters for blackbox components.
In *Proceedings of the 3rd International Workshop on Component-Oriented Programming (WCOP'98)*, 1998.
Disponible à l'adresse citeseer.ist.psu.edu/kucuk98customizable.pdf.

[80] Oussama LAYAIDA.
Un support logiciel à composants pour la construction d'applications multimédia adaptatives.
Thèse de Doctorat, INRIA Rhône-Alpes, France, 2005.

[81] Oussama LAYAIDA et Daniel HAGIMONT.
Designing self-adaptive multimedia applications through hierarchical reconfiguration.
In *Proceedings of the 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 95–107, 2005.

[82] Thomas LEDOUX.
État de l'art sur l'adaptabilité. projet rntl arcad d1.1 12/12/2001.

[83] Jeff MAGEE et Jeff KRAMER.
Dynamic structure in software architectures.

In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[84] Raphael MARVIE et et AL.
Projet cper mosaiques (modèles et infrastructures pour applications ubiquitaires), livrable 1, programme tac 2004-2007, thème 2.1 logiciel pour la communication ustl/uvhc/emd/inrets.
Disponible à l'adresse http ://www2.lifl.fr/Mosaiques/.

[85] Raphael MARVIE et Marie-Claude PELLEGRINI.
Modèles de composants, un état de l'art.
Coopération dans les systèmes à objets, special issue of L'objet, 8(3) :61–89, 2002.

[86] Vlada MATENA et Mark HAPNER.
Enterprise javabeans specification, v1.1. sun. microsystems, 1999.
Disponible à l'adresse http ://java.sun.com/products/ejb/docs.html.

[87] Nenad MEDVIDOVIC et Richard N. TAYLOR.
A classification and comparison framework for software architecture description languages.
IEEE Transactions on Software Engineering, 26(1) :70–93, 2000.

[88] Tom MENS, Jim BUCKLEY, Matthias ZENGER et Awais RASHID.
Towards a taxonomy of software evolution.
In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, 2003.
Disponible à l'adresse citeseer.ist.psu.edu/mens02towards.html.

[89] Tom MENS, Serge DEMEYER et Dirk JANSSENS.
Formalising behaviour preserving program transformations.
In *Proceedings of the International Conference on Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.

[90] Tom MENS et Tom TOURWE.
A survey of software refactoring.
IEEE Transactions on Software Engineering, V. 30 :126–139, 2004.

[91] Tom MENS, Niels VAN EETVELDE, Serge DEMEYER et Dirk JANSSENS.
Formalizing refactorings with graph transformations.
Journal on Software Maintenance and Evolution : Research and Practice, 2005.

[92] Bertrand MEYER.
Object-Oriented Software Construction.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[93] Robin MILNER, Joachim PARROW et David WALKER.
A calculus of mobile processes, i.
Information and Computation, 100(1) :1–40, 1992.

[94] Maria-Teresa Segarra MONTESINOS.
Une plate-forme à composants adaptables pour la gestion des environnements sans fil.
Thèse de Doctorat, Université de Rennes I, France, 2000.

[95] Arun MUKHIJA et Martin GLINZ.
The casa approach to autonomic applications.
In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, pages 173–182, 2005.
Disponible à l'adresse citeseer.ist.psu.edu/mukhija05casa.html.

[96] Arun MUKHIJA et Martin GLINZ.

Runtime adaptation of applications through dynamic recomposition of components.
In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS'05)*, pages 124–138, 2005.

[97] Audrey OCCELLO.
Capitalizing safety of applications being adapted dynamically : the Satin executable model.
Thèse de Doctorat, Université de Nice-Sophia Antipolis - UFR Sciences, France, 2006.

[98] Audrey OCCELLO, Anne-Marie PINNA-DÉRY, Mireille BLAY-FORNARINO et Michel RIVEILL.
Vers une adaptation dynamique cohérente des composants.
In *Proceedings of the french-speaking Workshop « Systèmes à composants adaptables et extensibles »*, Grenoble, France, octobre 2002. INRIA.

[99] William F. OPDYKE.
Refactoring Object-Oriented Frameworks.
Thèse de Doctorat, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.
Disponible à l'adresse citeseer.ist.psu.edu/opdyke92refactoring.pdf.

[100] Chabanne OUSSALAH.
Ingénierie objet : Concepts et techniques.
InterÉditions (Dunod), 1997.

[101] Mourad OUSSALAH.
Ingénierie des composants : Concepts, techniques et outils.
Éditions Vuibert, 2005.

[102] Andrew PATZER.
Programmation Java côté serveur : Servlets, JSP et EJB.
Wrox Prss et Éditions Eyrolles, édition Française, 2002.

[103] Renaud PAWLAK, Jean-Philippe RETAILLÉ et Lionel SEINTURIER.
Foundations of AOP for J2EE Development.
APress, 2005.

[104] Marie-Claude PELLEGRINI, Olivier POTONIÉE, Raphaël MARVIE, Sébastien JEAN et Michel RIVEIL.
Cesure : une plate-forme d'applications adaptables et sécurisées pour usagers mobiles.
Evolution des plate-formes orientées objets répartis, special issue of Calculateurs Parallèles, 12(1) :113–120, 2000.

[105] Michael PHILIPPSEN et Matthias ZENGER.
Javaparty - transparent remote objects in java.
Concurrency - Practice and Experience, 9(11) :1225–1242, 1997.

[106] Daniel PÉCHOIN et François DEMAY.
Le petit larousse 1997, 1996.

[107] Gustavo ROSSI, Silvia GORDILLO et Robert LAURINI.
Génération de services dépendant du contexte pour des applications mobiles.
In *Proceedings of the french-speaking Workshop UBIMOB*, Nice, France, October 2004.

[108] Salah SADOU.
Évolution du logiciel.
RSTI - L'Objet, 13(1), 2007.

[109] Yasushi SAITO et Marc SHAPIRO.

Optimistic replication.
ACM Computing Surveys, 37(1) :42–81, 2005.

[110] Daniel SALBER, Anind K. DEY et Gregory D. ABOWD.
The context toolkit : aiding the development of context-enabled applications.
In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '99)*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[111] Bill N. SCHILIT, Marvin M. THEIMER et Brent B. WELCH.
Customizing mobile application.
In *Proceedings of the USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, Cambridge, MA, US, 1993.
Disponible à l'adresse citeseer.ist.psu.edu/schilit93customizing.html.

[112] Albrecht SCHMIDT, Kofi Asante AIDOO, Antti TAKALUOMA, Urpo TUOMELA, Kristof Van LAERHOVEN et Walter Van de VELDE.
Advanced interaction in context.
In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, pages 89–101, 1999.

[113] Jean-Guy SCHNEIDER et Markus LUMPE.
Modelling objects in PICT.
Rapport technique IAM-96-004, University of Berne, Institute of Computer Science and Applied Mathematics, 1996.
Disponible à l'adresse citeseer.ist.psu.edu/schneider96modelling.pdf.

[114] Maria-Teresa SEGARRA et Françoise ANDRÉ.
A framework for dynamic adaptation in wireless environments.
In *Proceedings of the European Conference on the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 336, Washington, DC, USA, 2000. IEEE Computer Society.

[115] Farida SEMMAK.
Réutilisation de composants de domaine dans la conception de systèmes d'information.
Thèse de Doctorat, Université Paris I, France, 1998.

[116] Aline SENART.
Canevas logiciel pour la construction d'infrastructures logicielles dynamiquement adaptables.
Thèse de Doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, novembre 2003.

[117] Abdelhak-Djamel SERIAI, Gautier BASTIDE et Mourad OUSSALAH.
How to generate distributed software components from centralized ones ?
Journal of Computers, 1(11) :40–52, 2006.

[118] Abdelhak-Djamel SERIAI, Gautier BASTIDE et Mourad OUSSALAH.
Transformation of centralized software components into distributed ones by code refactoring.
In *Proceedings of the 6th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 332–346, 2006.

[119] Brian Cantwell SMITH.
What do you mean, meta ?
In *Proceedings of the International Workshop on Reflection and Metalevel Architectures in OO Programming, OOPSLA/ECOOP'90*, 1990.

[120] André SPIEGEL.
Pangaea : An automatic distribution front-end for java.

In *Proceedings of the 11st IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 93–99, London, UK, 1999. Springer-Verlag.

[121] Java Web START.
Web site.
Disponible à l'adresse http ://java.sun.com/products/javawebstart/.

[122] Bill SWARTOUT, Ramesh PATIL, Kevin KNIGHT et Tom RUSS.
Toward distributed use of large-scale ontologies.
In *Proceedings of the AAAI-97 Spring Symposium Series*, pages 138–148, 1997.

[123] Clemens SZYPERSKI.
Component software : beyond object-oriented programming.
ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[124] Michiaki TATSUBORI, Toshiyuki SASAKI, Shigeru CHIBA et Kozo ITANO.
A bytecode translator for distributed execution of legacy Java software.
Lecture Notes in Computer Science, 2072 :236–255, 2001.

[125] Eli TILEVICH et Yannis SMARAGDAKIS.
J-orchestra : Automatic java application partitioning.
In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 178–204, London, UK, 2002. Springer-Verlag.

[126] Yves VANDEWOUDE et Yolande BERBERS.
Meta model driven state transfer in component oriented systems.
In *Proceedings of the International Workshop On Unanticipated Software Evolution*, pages 3–8, Warshau, Poland, 2003.

[127] Sylvain VAUTTIER.
Une Etude du Comportement des Objets Composites.
Thèse de Doctorat, Université des Sciences et Techniques du Languedoc de Montpellier II, Montpellier, France, 1999.

[128] Jean-Baptiste WALDNER.
Nano-informatique et intelligence ambiante.
Éditions Hermès-Lavoisier, 2007.

[129] Sanjiva WEERAWARANA, Francisco CURBERA, Matthew J. DUFTLER, David A. EPSTEIN et Joseph KESSELMAN.
Bean markup language : A composition language for javabeans components.
In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 173–188, 2001.

[130] Peter WEGNER.
Dimensions of object-based language design.
In *Proceedings of the Internationnal Conference on Object-oriented programming systems, languages and applications (OOPSLA '87)*, pages 168–182, New York, NY, USA, 1987. ACM Press.

[131] Mark WEISER.
The computer for the 21st century.
Scientific American, September 1991.
Disponible à l'adresse http ://www.ubiq.com/hypertext/weiser/.

[132] Mark WEISER.
Some computer science issues in ubiquitous computing.
Communications of the ACM, 36(7) :75–84, 1993.

Références bibliographiques relatives à ce travail de recherche

Revues internationales

1. Abdelhak-Djamel Seriai, Gautier Bastide, Mourad Oussalah : **How To Generate Distributed Software Components From Centralized Ones ?** *Journal of Computers, pp. 40-52, Issue 5, ISSN 1796-203X, 2006.*

Revues nationales

1. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Restructuration de composants logiciels : une approche d'adaptation structurelle de composants logiciels monolithiques basée sur leur refactorisation**. *L'Objet, Vol. 13, pp. 83-118, Hermes Editions, 2007.*

Conférences internationales

1. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Software Component Re-engineering for their Runtime Structural Adaptation**. *Proceedings of the 31st Annual IEEE International Computer Software and Applications Conference (COMPSAC - 2007), July 23-27, 2OO7, Beijing, China.*

2. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Dynamic Adaptation of Software Component Structures**. *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration, IRI - 2006, Sep 16-18, 2006, Waikoloa, Hawaii, USA.*

3. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Adaptation of Monolithic Software Components by their Transformation into Composite Configurations Based on Refactoring**. *Proceedings of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006). Västerås near Stockholm, Sweden, July 2006, Lecture Notes in Computer Science 4063 Springer 2006, ISBN 3-540-35628-2.*

4. Abdelhak-Djamel Seriai, Gautier Bastide, Mourad Oussalah : **Transformation of centralized software components into distributed ones by code refactoring**. *Proceedings of the International Conference on Distributed Applications and Interoperable Systems (DAIS 2006), Bologna, Italy, June 2006, Lecture Notes in Computer Science 4025 Springer 2006, ISBN 3-540-35126-4.*

5. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Adapting Software Components by Structure Fragmentation**. *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006) ; Software Engineering : Applications, Practices, and Tools (SE), Dijon, France, April 2006.*

6. Gautier Bastide : **A Refactoring-based Tool for Software Component Adaptation**. *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), March 22-24, Bari, Italy, 2006.*

Workshops internationaux

1. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Software Component Reengineering for their Context-Aware Deployment in Ubiquitous Environments**. *Proceedings of the Third International ERCIM Symposium on Software Evolution, Paris, France, October 2007.*

Conférences nationales

1. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Transformation d'un composant logiciel centralisé en un composant logiciel distribué**. *Journées Composants 2006, Perpignan, Octobre 2006.*

Workshops nationaux

1. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **Auto-adaptation de composants logiciels : Prise en compte du contexte d'exécution pour l'auto-adaptation structurelle de composants logiciels**. *INFORSID 2007, GEDSIP (Gestion de données dans les systèmes d'information pervasifs). Perros-Guirec, 2007.*

2. Gautier Bastide : **Restructuration de composants logiciels : Une approche d'adaptation structurelle statique basée sur la refactorisation de code orienté-objet**. *INFORSID 2006, OCM-SI. Hammamet, Tunisie, 2006.*

3. Gautier Bastide, Abdelhak-Djamel Seriai, Mourad Oussalah : **COMMODE :Un modèle pour l'adaptation structurelle de composants logiciels**. *Session OCM-LMO, Berne, Mars 2005.*

Rapports techniques

1. Laurence Duchien et al. : **Projet cper Mosaïques (modèles et infrastructures pour applications ubiquitaires) : rapport final**. *Programme tac 2004-2007, thème 2.1 logiciel pour la communication ustl/uvhc/emd/inrets. 2007.*

2. Raphaël Marvie et al. : **Projet cper Mosaïques (modèles et infrastructures pour applications ubiquitaires) : Livrable 1**. *Programme tac 2004-2007, thème 2.1 logiciel pour la communication ustl/uvhc/emd/inrets. 2006.*

3. Gautier Bastide : **Résumé des travaux de la première année de thèse : Adaptation de Composants Logiciels dans les environnements ubiquitaires et mobiles**. *Technical Report N˚2005-9-2, Ecole des Mines de Douai, Septembre 2005.*

Divers

1. Gautier Bastide : **Adaptation de Composants Logiciels dans les Environnements Ubiquitaires et Mobiles**. *Journée JDOC 2006, Saint-Nazaire, Avril 2006.*

Liste des tableaux

— *Corps du document* —

— *Annexes* —

Liste des figures

— Corps du document —

— Annexes —

Table des matières

— Corps du document —

— *Annexes* —

— *Pages annexées* —

Scorpio : une Approche d'Adaptation Structurelle de Composants Logiciels

Application aux Environnements Ubiquitaires

Gautier BASTIDE

Résumé

La réutilisation à grande échelle de composants logiciels se révèle être un challenge pour la conception de nouvelles applications. Dans la grande majorité des cas, pour être intégrés à une application, les composants disponibles ont besoin d'être adaptés afin de faire face à la multiplicité des environnements de déploiement dotés de caractéristiques variables. Ainsi, pour éviter le redéveloppement de nouveaux composants et favoriser la réutilisation, de nombreuses approches ont proposé des techniques permettant d'adapter le comportement de composants existants. Cependant, adapter le comportement de composants n'est pas suffisant pour permettre leur réutilisation : il faut également adapter leur structure. Or, aucune approche existante ne permet de répondre pleinement à ces besoins en adaptation structurelle. Ainsi, notre objectif est de proposer une approche, appelée Scorpio, permettant d'adapter la structure de composants. Nous nous focalisons plus particulièrement sur des composants existants. Dans un premier temps, nous nous sommes intéressés à l'adaptation structurelle de composants existants en proposant un processus permettant leur ré-ingénierie vers de nouvelles structures. Puis, pour répondre aux besoins liés à une adaptation sans interruption de l'exécution, nous avons proposé des mécanismes permettant de prendre en charge l'adaptation dynamique de ces composants. Partant du constat qu'un certain nombre d'environnements, tels que les environnements ubiquitaires, nécessite une automatisation du processus d'adaptation, nous avons proposé alors de prendre en charge ces besoins à travers une approche permettant l'auto-adaptation structurelle de composants logiciels. Enfin, nos propositions ont été mises en œuvre d'une part par la réalisation du prototype Scorpio-Tool implémenté en Fractal et d'autre part, par la définition et le développement d'un scénario ubiquitaire permettant l'expérimentation de ces propositions.

Mots-clés : Composant logiciel, réutilisation, ré-ingénierie, adaptation structurelle, adaptation statique, adaptation dynamique, auto-adaptation, restructuration, distribution, refactorisation, orienté objet, Fractal.

Scorpio : an Approach for Software Component Structural Adaptation

Usage for adaptation in Ubiquitous Environments

Abstract

Software component re-use is a challenge for designing new applications. In many cases, the existing components require to be adapted because of the large variety of existing software and hardware environments. To avoid component redevelopments, many approaches proposed techniques allowing an administrator to adapt the component behaviors. However, in certain cases such as in ubiquitous environments, adapting behavior is insufficient to allow its re-use: it also requires adapting its structure. However, few works propose approaches allowing us to adapt the component structure and in these works, only composite component can be adapted. Thus, we aim at defining, in this thesis, an approach for adapting monolithic and composite component structures in order to increase their reusability. First, we propose a static structural adaptation technique which is based on a restructuring process which allows us to transform an existing component into a component whose structure matches with the new needs. Then, we develop an approach for runtime structural adaptation which is based on a runtime adaptable component model. Finally, we introduce self-adaptation mechanisms into our model, dedicated to components deployed in ubiquitous environments. We experiment our approach by implementing a prototype called Scorpio-Tool, which allows us to adapt Fractal components and by creating an ubiquitous scenario where structural adaptation is required.

Keywords: Software component, software reuse, reengineering, structural adaptation, static adaptation, runtime adaptation, self-adaptation, restructuration, distribution, refactoring, object-oriented, Fractal.

Classification ACM

Catégories et descripteurs de sujets : D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*.

Termes généraux : Algorithms, Design, Experimentation, Performance.

Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.