De l'adaptation des composants logiciels vers leur évolution

Le Goaer Olivier

RAPPORT DE STAGE DE MASTER RECHERCHE ALD
7 septembre 2005

Encadré par :

Dalila TAMZALIT
Équipe MOC
Université de Nantes
2, rue de la Houssinière,
BP 92208 - 44322 Nantes
Dalila.Tamzalit@univ-nantes.fr

Abdelhak-Djamel SERIAI
Équipe CSL
École des Mines de Douai
941, rue Charles Bourseul
BP 838 - 59508 Douai Cedex
seriai@ensm-douai.fr

Laboratoire d'Informatique de Nantes Atlantique
2, rue de la Houssinière
B.P. 92208
F-44322 NANTES CEDEX 3

Table des matières

Liste des tableaux

Table des figures

Remerciements

Je tiens tout d'abord à remercier chaleureusement Dalila Tamzalit, Maître de Conférence à l'Université de Nantes, et Abdelhak-Djamel Seriai, Maître de Conférence à l'École de Mines de Douai, mes encadrants, qui m'ont encouragé et m'ont donné l'opportunité d'effectuer ce premier travail d'initiation à la recherche dans les meilleures conditions.

Plus généralement, je remercie tous les membres de l'équipe MOC du LINA pour l'excellente ambiance de travail, et outre les personnes déjà citées, Mourad Oussalah, Professeur à l'Université de Nantes ainsi que Nassima Sadou, doctorante à l'Université de Nantes. J'en profite pour saluer Brice Turcaud, ami et partenaire privilégié de cette aventure enrichissante au sein de l'équipe.

Je souhaite aussi remercier tous les enseignants qui sont intervenus durant ce Master Recherche à l'École Polytechnique de l'Université de Nantes. Leurs enseignements ont été pour moi un premier pas vers une réflexion orienté recherche.

J'adresse tout naturellement mes derniers remerciements à mes parents pour leur soutien et leurs encouragements. Ce travail est aussi le fruit de leur investissement.

Introduction

La complexité croissante des systèmes informatiques et leur évolution de plus en plus rapide ont suscité un intérêt accru pour le développement de logiciels à base de composants. Cet intérêt est motivé par la réduction des coûts et des délais de développement des applications [Szy98]. En effet, on prend moins de temps à acheter (et donc à réutiliser) un composant qu'à le concevoir, le coder, le tester, le déboguer et le documenter [1]. Aujourd'hui, une nouvelle évolution s'annonce dans l'art de concevoir des systèmes logiciels. Après les technologies objet qui ont profondément modifié l'ingénierie des systèmes logiciels en améliorant leur analyse, leur conception et leur développement, une nouvelle ère de conception de systèmes débute : l'orienté composant. Il s'agit de concevoir et de développer des systèmes par assemblage de composants réutilisables, à l'image par exemple des composants électroniques ou des composants mécaniques. Plus précisément, il s'agit de : concevoir et développer des systèmes à partir de composants préfabriqués, préconçus et pré-testés ; réutiliser ces composants dans d'autres applications.

Toutefois, une caractéristique intrinsèque d'un logiciel, représentant une activité du monde réel, est la nécessité d'être *mis à jour* pour satisfaire de nouvelles exigences. La première loi de Lehman, issue de constatations sur le terrain, stipule ainsi qu'un logiciel doit nécessairement être mis à jour faute de quoi il devient progressivement inutile [ML85]. En conséquence, les activités de maintenance, dont les *mises à jour* constituent une grande part, représentent une part très importante du chiffre d'affaire des sociétés développant des logiciels. Il est donc nécessaire de proposer des méthodes, des techniques et des outils facilitant ces activités tout en diminuant leur coûts.

Dans la littérature orientée composant, les mises à jour se déclinent en deux techniques : *l'évolution* et *l'adaptation*. En effet, un des axes majeurs de recherche de ces dernières années a été de proposer des *techniques d'évolution* et des *techniques d'adaptation*, qui permettent —plus ou moins facilement— de mettre en oeuvre les changements désirés, ainsi que des mécanismes garantissant la cohérence[TFS04, Szy04, OBFDR02] et la sûreté de la mise à jour. Mais les opérations d'évolution et d'adaptation n'ont pas clairement été mises en relation, c'est ce que nous tentons de faire dans le cadre de cette étude. Plus généralement, notre étude doit pouvoir servir de guide à un architecte logiciel en lui permettant d'identifier la mise à jour dans un premier temps, puis de mettre en oeuvre cette mise à jour de manière optimale.

Après avoir exposé notre problématique, nous présentons les hypothèses de travail qui seront le socle de l'ensemble des réflexions de notre étude. Fort de ces notions de bases, nous pouvons définir un cadre générique aux approches d'évolution et d'adaptation. Puis nous tentons d'établir une taxonomie des opérations de mises à jour, par l'étude des différentes techniques d'évolution et d'adaptation trouvées dans la littérature. Enfin, nous identifions les relations entre opérations d'adaptation structurelle et opérations d'évolution, appuyé par une validation pratique sous le modèle de composant Fractal.

[1] on parle dans ce cas de conception *from scratch* : tout est à faire

Chapitre 1

PRÉSENTATION DU SUJET

1.1 Problématique

Une importante problématique s'impose de plus en plus : celle d'assurer le cycle de vie des composants, principalement en assurant leur mise à jour. Nous proposons d'étudier l'évolution et l'adaptation des applications à base de composants logiciels, qui sont les deux déclinaisons majeures de la mise à jour. Bien souvent, l'architecte discerne difficilement ce qui relève de l'évolution de ce qui relève de l'adaptation. Il nous faut fournir un guide à l'architecte pour lui permettre de caractériser ses mises à jour et de décider quelles solutions sont adaptées à son problème. Il nous faut surtout lui faire prendre conscience des possibilités de mises à jour offertes par son système et du rôle actif qu'il doit jouer dans le processus de mise à jour.

1.2 Hypothèses de travail

1.2.1 Terminologie

Nous utilisons le terme *mise à jour* pour désigner indifféremment l'évolution et l'adaptation. Ce terme disparaîtra au fur et à mesure que notre étude se concentrera sur l'évolution et l'adaptation.

1.2.2 Niveaux d'abstraction

Notre étude concerne les architectures à base de composants nées du *paradigme des composants*[Szy96]. L'ensemble de notre travail est basé sur une architecture bien spécifiée, c'est à dire dont les éléments constitutifs ont été identifiés et clairement définis.

Présentation

Notre système à base de composants peut-être représenté à trois niveaux[1] d'abstraction [OTS05] :

[1] éventuellement un quatrième : le méta-méta-niveau

1. A partir des éléments architecturaux communément admis par la majorité des ADLs[2][MT00], un *méta-modèle* des concepts de bases des ADLs peut être établi. Il constitue le niveau d'abstraction le plus élevé appelé *Niveau Méta* .
2. Le *Niveau Architecture* permet de décrire une architecture en utilisant un ou plusieurs éléments architecturaux du niveau méta.
3. Enfin, le *Niveau Application* décrit la structure d'une application conformément à son architecture.

Tout comme le MDA[OMG97] définit *l'espace technique* des modèles, nous raisonnons dans un autre espace technique, celui des architectures à base de composants. L'application est une *instance* de son architecture. L'architecture se *conforme* à son méta-modèle. Cette hiérarchie est importante car une mise à jour opérée à un niveau impacte directement sur le niveau inférieur.

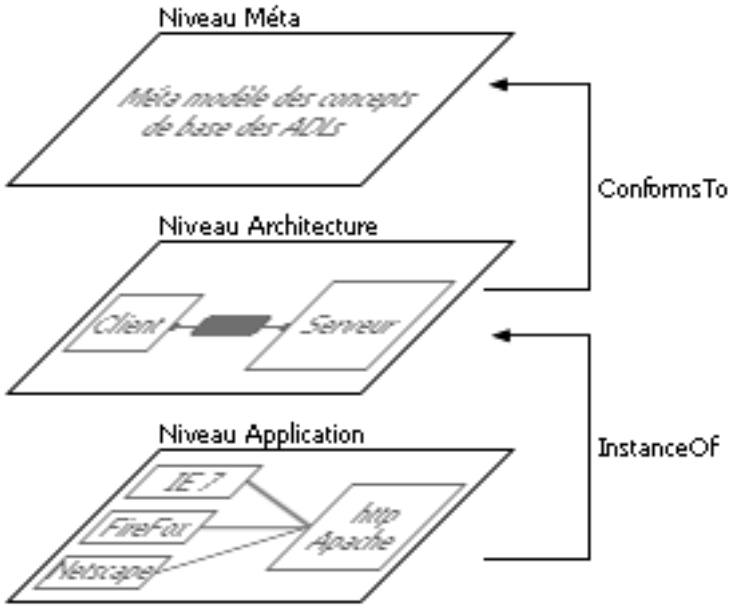

FIG. 1.1 – Niveaux d'abstraction dans les architectures à composants

Niveaux considérés

Pour notre étude, nous raisonnons sur les deux derniers niveaux : niveau architecture et niveau application. Le niveau architecture est la description architecturale du système et constitue la vue statique de ce dernier. En revanche, le niveau application est l'instance de la description architecturale et constitue la vue dynamique du système. Les mises à jour s'appliquent aussi bien au niveau architecture qu'au niveau application.

[2]Architecture Description Langage

Remarque On peut néanmoins vouloir mettre à jour le méta-niveau, ce qui reviendrai à modifier les concepts actuels des ADLs[3], mais cela dépasse le sujet de notre étude. Nous avons basé celle-ci sur les concepts issus du consensus actuel de la communauté de recherche sur les ADLs.

1.2.3 Concepts du paradigme composant

Pour pouvoir définir un cadre générique à l'évolution et l'adaptation des architectures logicielles à base de composant, il est nécessaire d'identifier les éléments qui seront sujets au changement. Les concepts que nous retrouvons dans les différents langages de description d'architecture [MT00, GMW68, SOK03] sont au nombre de six et vous sont présentés ci-dessous.

Le composant — Un composant est une unité de calcul ou de stockage. Il peut être *primitif* ou *composé* ; on parle alors dans ce dernier cas de composite. Sa taille peut aller de la fonction mathématique à une application complète. Deux parties définissent un composant. Une première partie, dite externe, comprend la description des interfaces fournies et requises par le composant. Elle définit les interactions du composant avec son environnement. La seconde partie correspond à son contenu et permet la description du fonctionnement interne du composant.

L'interaction — L'interaction représente le type de communication entre composants. Les formes simples d'interaction comme : appel de procédures/méthodes, évènements, pipes en sont des exemples. Cependant, les connecteurs peuvent aussi représenter des interactions plus complexes telles que : protocoles C/S, ou bien un lien SQL entre une Base De Donnée et une application.
Nous avons volontairement choisi le terme "interaction" pour ne pas présupposer des concepts utilisés. En effet, certains modèle de composants permettent de manipuler explicitement cette interaction en la réifiant, c'est à dire en la considérant comme entité de première classe : on parle alors souvent de connecteur. Le connecteur possède une interface et encapsule son comportement (i.e son implémentation via la glue) qui est la mise en oeuvre du protocole associé à l'interaction. Si l'interaction n'est pas considérée comme entité de première classe (on parle parfois de connecteur implicite) elle ne possède pas explicitement les notions d'interfaces et de comportement.

La configuration — Une configuration représente un graphe connexe de composants, de connecteurs et définit la façon dont ils sont reliés entre eux. On parle aussi de topologie. Certains modèle considère la configuration comme un composant composite racine (ex : Fractal[BCS04]). Dès lors, une configuration possède des interfaces et un contenu.

L'interface — Elle est le point de communication qui permet d'interagir avec l'environnement. On la retrouve donc associée aux composants et aux connecteurs. Elle spécifie des *ports* dans le cas des composants et des *rôles* dans le cas des interactions.

[3] c'est même souhaitable du point de vue de la recherche en génie logiciel

Le service — L'interface d'un composant spécifie les services fournis et services requis. On distingue les services décrivant d'une part le comportement fonctionnel ou non fonctionnel du composant : les services fournis, et d'autre part les fonctionnalités dont il a besoin pour fonctionner : les services requis.

Le contenu — Il varie suivant la *granularité* de l'élément observé. Il peut s'agir d'un comportement (i.e l'implémentation) lorsque l'on considère un composant primitif ou un connecteur, ou bien d'autres éléments dans le cas d'un élément composite.

1.3 Nos objectifs

Un état de l'art sur les mises à jour dans les architectures à composant doit permettre d'aboutir à une *caractérisation et une classification des opérations de mises à jour* pour permettre à l'architecte logiciel d'y voir plus clair. Puis, il est important de mener une réflexion sur *les relations qui existent entre les deux déclinaisons de la mise à jour : l'évolution et l'adaptation*. C'est la relation "adaptation vers évolution" qui nous intéresse ici, mais le travail proposé devrait permettre à terme d'étudier la relation inverse, à savoir "évolution vers adaptation". En particulier, nous cherchons à savoir si *une opération d'adaptation peut être exprimée sous forme d'une ou plusieurs opérations d'évolution*, ou en d'autres termes, si l'on peut ramener – totalement ou partiellement – un problème d'adaptation à un problème d'évolution, la finalité étant de *capitaliser les opérations de mises à jour* afin de développer de nouvelles opérations au moindre effort. Nous étudierons également *les évolutions déclenchées en réponse à une adaptation afin de converser un système stable et cohérent*. Enfin, cette démarche globale doit être validée sur un modèle de composant existant : la plate-forme *Fractal* et son implémentation Java de référence nommée *Julia*.

Chapitre 2

ÉTAT DE L'ART : MISE A JOUR DES COMPOSANTS LOGICIELS

2.1 Mise à jour des composants logiciels

"Pour s'améliorer, il faut changer.
Donc, pour être parfait, il faut avoir changé souvent."
Winston Churchill

Ce qui est vrai pour les Hommes l'est aussi pour les Composants logiciels. Pour nous, le *changement* s'appelle la *mise à jour* mais garde la même signification. C'est autour de ce concept que s'articule toute notre étude. Il nous faut cerner la mise à jour sous ses différentes dimensions pour fournir un cadre générique d'étude.

2.1.1 Objectifs de la mise à jour

Nous répondons ici, à une question essentielle et légitime : pourquoi mettre à jour ? Un système logiciel qui n'est pas mis à jour devient rapidement obsolète. De nouvelles fonctionnalités sont régulièrement requises et les versions de ses éléments constitutifs les plus à jour sont nécessaires pour garantir un fonctionnement optimal. Ces motivations sont liées à des préoccupations économiques : les mises à jour du système logiciel sont guidées par l'évolution des technologies et des environnements. L'obsolescence d'un système logiciel ou son inadaptation signifie sa mort.

L'ingénierie des composants (CBSE[1]) participe à cet objectif, en ce sens qu'elle spécifie des entités (idéalement) réutilisables qui peuvent s'assembler entre elles, ce qui autorise plus intuitivement les mises à jour : remplacer une entité par une autre plus récente ou plus complète est nativement intégré au paradigme de programmation par composant.

Nous établissons une première classification, en partie inspirée de [KBC02], des mises à jour pouvant être requises. Ces mises à jour peuvent être classées selon leur cause respective : les défauts, les nouveaux besoins, les performances, l'environnement.

[1]Component Based Software Engineering

Mise à jour correctionnelle — dans certains cas, on remarque que l'application en cours d'exécution ne se comporte pas correctement ou comme prévu. La solution est d'identifier l'élément de l'application qui pose le problème et de le remplacer par une nouvelle version supposée correcte. Cette nouvelle version se contente simplement de corriger ses défauts.

Mise à jour fonctionnelle — au moment du développement de l'application, certaines fonctionnalités ne sont pas prises en compte. Avec l'évolution des besoins de l'utilisateur, l'application doit être étendue avec de nouvelles fonctionnalités. Cette extension peut être réalisée en ajoutant un ou plusieurs éléments pour assurer les nouvelles fonctionnalités ou en étendant les éléments existants.

Mise à jour perfective — l'objectif de ce type de mise à jour est d'améliorer les performances de l'application. A titre d'exemple, on se rend compte que l'implémentation d'un composant n'est pas optimisée. On décide alors de remplacer l'implémentation du composant en question. Un autre exemple peut être un composant qui reçoit beaucoup de requêtes et qui n'arrive pas à les satisfaire. Pour éviter la dégradation des performances du système, on diminue la charge de ce composant en installant un autre composant qui lui partage sa tâche.

Mise à jour adaptative — l'environnement d'exécution, les composants matériels ou d'autres applications ou ressources dont un élément dépend changent. Dans ce cas, l'élément est adapté en réponse aux changements affectant son environnement d'exécution.

Nous remarquons que cette dernière mise à jour est déclenchée en réponse à une modification de l'environnement alors que les trois premières servent à améliorer l'application. Intuitivement, cette nuance nous permet déjà de considérer les mises à jour *correctionnelles*, *fonctionnelles* et *perfective* comme de l'*évolution* et les mises à jour *adaptatives* comme de l'*adaptation*. Nous verrons aux sections suivantes si les définitions confirment notre intuition.

2.1.2 Gestion de la mise à jour

Un système logiciel va subir une série de mise à jour au cours de sa vie. Cette série de mise à jour peut s'effectuer *sans rupture* ou *avec rupture*. Cela indique si la version d'un élément est sauvegardé avant qu'une mise à jour lui soit appliqué. Une série de mises à jour sans rupture conserve une trace de toutes les mises à jour effectuées : c'est l'*historique des changements*[2].
Par ailleurs, dans ce processus, certaines mises à jour ont été explicitement déclenchées, d'autres sont leurs conséquences. Ces conséquences correspondent aux *propagations des mises à jours*.

[2]permet de garder un lien entre le modèle de départ et le modèle d'arrivé

Nous constatons que les techniques de mise à jour étudiées, peuvent ou non, gérer l'historique et gérer la propagation des mises à jour. Celles-ci sont présentées en sections 2.3.2 et 2.4.2.

Historique des mises à jour — L'historique des mises à jour d'un système logiciel représente l'historique de toutes les mises à jour qui ont été faites au logiciel. Des outils rendent cet historique explicitement disponible et sont utilisés pour une grande variété de buts. On peut distinguer les mécanismes qui supporte cette gestion des versions et ceux qui ne fournissent pas les moyens de distinguer les nouvelles des vieilles versions. Dans les systèmes qui ne gèrent pas du tout les versions, les mises à jour sont appliquées de manière destructive, c'est à dire que les nouvelles versions des éléments écrasent les anciennes. Dans ce scénario, les vieilles versions sont perdues dans le processus de mise à jour.

Propagation des mises à jour — La mise à jour d'un élément architectural peut entraîner la mise à jour d'autres éléments architecturaux : c'est l'impact de la mise à jour. Ces impacts diffèrent suivant l'élément considéré. Si la mise à jour d'un élément est restreinte à celui-ci, on parle d'impact local. Si en revanche, le changement d'un élément conduit à changer d'autres éléments, on parle d'impact global. Ainsi, la mise à jour d'un élément peut se voir augmenté par des mises à jour d'éléments supplémentaires, afin de respecter la cohérence de l'architecture. Ce sont les liens entre ces éléments qui vont permettre de découvrir dynamiquement les nouveaux éléments concernés par la mise à jour, c'est à dire de *propager les mises à jour*.

`MiseAJour(élément1)` $\Rightarrow$ {`MiseAJour(élément2), MiseAJour(élément3), ...`}

TAB. 2.1 – Propagation des mises à jour

Par exemple :
- Supprimer un composant implique la suppression des interactions associées
- La suppression d'une interaction n'implique pas forcement la suppression des composants qui l'utilisaient
- Supprimer un composant composite entraîne la suppression de tous les éléments qu'il contient
- Supprimer un service nécessite la suppression de son implémentation

Cette propagation des mises à jour est nécessaire pour maintenir la cohérence du système. Elle devrait être systématiquement prise en compte par les techniques de mises à jour.

2.1.3 Définitions

Une mise à jour se traduit par l'évolution et/ou l'adaptation, deux notions qui se distinguent l'une de l'autre même si leur frontière est floue. Si l'on se réfère à la définition de ces deux mots dans le dictionnaire (Petit Robert - Édition de 1998), nous trouvons :

2.1.3.1 Évolution

Évolution n. f. : "Suite de transformations dans un même sens..."

Conformément à cette définition, *l'évolution* représente l'ensemble des modifications appliquées à un système logiciel pour atteindre un objectif. Ajouter un nouveau composant à une configuration en est un exemple simple. Cette définition, très générale, illustre bien la difficulté à définir la nuance avec l'adaptation, en ce sens que tout changement, même minime, peut être considéré comme une évolution.

2.1.3.2 Adaptation

Adaptation n. f. : "Modifications pour répondre harmonieusement à des situations nouvelles."

Le processus qui consiste à modifier un élément pour l'utiliser dans une application particulière, pour un besoin particulier, est donc désigné sous le nom *d'adaptation.*
Typiquement, l'ingénierie logicielle à base de composants prévoit de construire des applications en assemblant des éléments architecturaux réutilisables. On imagine qu'il suffit de choisir un ensemble de composants qui fournissent certaines fonctionnalités requises par l'application et puis d'assembler ces composants en reliant les entrées aux sorties. Cependant, la recherche dans la réutilisation de logiciel a prouvé que la réutilisation directe est extrêmement faible et que les éléments architecturaux doivent généralement être modifiés d'une manière quelconque pour s'intégrer à l'application : c'est un exemple concret du besoin d'adaptation.

2.1.3.3 Comparatif

C'est donc l'objectif qui différencie l'évolution de l'adaptation :

L'objectif de l'adaptation est d'*assurer le bon fonctionnement* d'un composant confronté à un *changement de contexte.*
Pour assembler différents composants "prêt à l'emploi", l'enjeu est l'*inter-opérabilité* des composants, chacun constituant une part de l'application. Cette adaptation est nécessaire car ces composants sont typiquement développés par des fournisseurs différents et que la probabilité pour qu'ils puissent communiquer entre eux directement est faible.
On peut également adapter un composant déjà présent dans l'architecture à un nouvel environnement ou des nouvelles ressources. Par exemple, la diminution de l'espace de stockage doit conduire à adapter les composants pour qu'ils puissent encore fonctionner.

L'évolution désigne la *suite de modifications successives* qui mène aux amélioration désirées. Son objectif est donc essentiellement de répondre à un *besoin d'améliorations*, mais sa définition nous autorise à considérer certaines adaptations comme *des formes d'évolution.*

2.2 Caractérisation des mises à jour

2.2.1 Dimensions des mises à jour

Les mises à jour dans un systèmes logiciel sont donc inévitables et nous avons déjà répondu à la question du "pourquoi ?" en section 2.1.1 : la raison constituent une dimension de la mise à jour. Mais de nouvelles interrogations apparaissent et il convient de s'interroger également sur l'objet du changement (quoi ?), le moment du changement (quand ?), et enfin le support du changement (comment ?). En introduisant ces nouvelles dimensions, nous espérons définir un cadre suffisamment générique dans lequel les approches d'évolution et d'adaptation pourront être positionnées.

2.2.1.1 L'objet du changement (Quoi ?)

Il s'agit de répondre à la question la plus intuitive : Qu'est ce qui est susceptible de changer dans une architecture ?

L'enjeu est *d'identifier* les éléments concernés par le changement, c'est à dire d'être capable de repérer ces éléments au milieu d'autres éléments. En s'inspirant de [Tam00], nous considérons une architecture à base de composant comme un graphe constitué de noeuds et d'arcs :

Noeud — Un noeud représente les éléments architecturaux considérés comme entités de première classe. Ces éléments ont été présentés dans la section "Concepts de bases" et sont la *configuration*, le *composant*, l'*interaction*, l'*interface* et le *service*. Tous les noeuds ont un contenu.

Contenu d'un noeud— Selon la granularité du noeud, le contenu peut être une implémentation et/ou d'autres noeuds.

Arc — Un arc est tout type de lien entre deux noeuds. Il peut s'agir des liens de composition (d'un composant à une interface, d'une interface à un service,...) ou de comportement (d'un service à un autre service : i.e un appel de méthode).

Noeud	Contenu du noeud
Configuration	- [Interface] - noeuds ∈{Composant, Interaction}
Composant	- Interface (Ports) - Implémentation OU noeuds ∈{Composant, Interaction}
Interaction	- Interface (Rôles) - Implémentation
Interface	- Services
Service	- Implémentation

TAB. 2.2 – Noeuds et leur contenu

Légende : [...] élément optionnel

Remarque – Le contenu d'un noeud définit sa structure. Certains noeuds contiennent une implémentation, c'est à dire la réalisation d'un comportement. Un noeud dispose également de propriétés non fonctionnelles assurant une qualité de service (sécurité, persistance, transaction, ...). *Structure*, *comportement* et *propriétés non fonctionnelles* constituent les *vues* d'un noeud, c'est à dire les différents angles de sa mise à jour.

2.2.1.2 Le moment du changement (Quand ?)

La question concerne les caractéristiques temporelles : quand une mise à jour peut être faite ?

Nous pouvons distinguer, d'une part les approches de mises à jour statiques et, d'autre part celles qui sont dynamiques. En effet, les opérations de mises à jour peuvent être réalisées à l'arrêt (*approche statique*) ou à l'exécution (*approche dynamique*).
- Dans le cas des mises à jour statiques, les modifications sont prises en compte à la compilation ou au chargement et nécessitent l'arrêt du système.
- Dans le cas des mises à jour dynamiques, les modifications sont prises en compte pendant l'exécution et ne nécessitent donc pas l'arrêt du système. Cela est nécessaire pour les applications critiques et à haute disponibilité (applications bancaires, internet, télécoms...)

En théorie, un système est développé, déployé et exécuté. Dans cette hypothèse, les mises à jour prises en compte pendant le développement sont statiques (le système n'a pas encore été exécuté) et toutes les mises à jour prises en compte durant la maintenance du logiciel sont dynamiques (le système tourne).

	Statique	**Dynamique**
Expression des besoins	•	
Analyse	•	
Conception & Spec.	•	
Implémentation	•	
Tests		•
Déploiement		•
Maintenance		•

Tab. 2.3 – La MAJ dans le cycle de vie

Il est bien évident, qu'en réalité, les mises à jour qui interviennent au moment de la maintenance nécessitent souvent l'arrêt du système et sont donc statiques. Le tableau 2.3 est une vision idéale (utopique ?) du processus de mise à jour dans le cycle de vie.

2.2.1.3 Le support du changement (Comment ?)

Divers mécanismes/supports peuvent être fournis pour effectuer une mise à jour. Les mécanismes proposés peuvent être très variés : solutions automatisées[GKL04], techniques informelles, représentations formelles[Ous02], et beaucoup d'autres encore.

Nous traitons ici du degré d'automatisation supporté par un mécanisme. Nous pouvons distinguer le support *automatisé*, *semi-automatisé*, et *manuel* pour effectuer des changements. Dans le domaine de la re-ingénierie logicielle, de nombreuses tentatives ont été faites pour automatiser, ou du moins automatiser partiellement, les tâches d'entretien du logiciel. En réalité, ces évolutions automatisées requièrent certaines vérifications manuelle et ainsi, peuvent seulement être considérées comme partiellement automatisées. Dans le domaine spécifique du refactoring (c.a.d, la restructuration du code source orienté objet), la gamme d'outils s'étend de l'entièrement manuel à l'entièrement automatisé.

Par ailleurs, il est clair que le degré d'automatisation est orthogonal au degré de formalisme. Plus les modifications peuvent être formalisées, plus la mise à jour est automatisable. L'acteur de la mise à jour est également lié au degré d'automatisation : en effet, un support manuel, requière, par définition, l'implication d'un opérateur humain pour le déclenchement et la réalisation de la mise à jour. A l'opposé, dans le cas d'un support entièrement automatisé, les mises à jour peuvent être déclenchées et réalisées par une entité logicielle. Par exemple, le déclenchement des modifications peut s'opérer grâce à des *sondes* pour l'observation des ressources et des notifications de changements. Cela met en avant les notions *d'auto-évolution* et *d'auto-adaptation* puisque le logiciel peut modifier son propre comportement.

2.2.2 Bilan : Positionnement des mises à jour

En représentant chacune des caractéristiques précédentes par un axe, nous obtenons la caractérisation tridimensionnelle d'une mise à jour dans une architecture à base de composants.

Cette représentation tryptique, inspirée des travaux de [Tam00] sur les objets, a pour axe principal *l'objet de la mise à jour*, c'est à dire sur quoi porte la mise à jour ou quel est le noeud concerné par la mise à jour. Ensuite, se situent le *support du changement* (le Comment ?) et le *moment du changement* (le quand ?) représenté par les étapes du cycle de vie de l'application.

En conclusion, le triplet (Quoi, Quand, comment) positionne *les techniques de mise à jour* mais également *les problématiques de mise à jour*. Un architecte logiciel peut positionner sa problématique de mise à jour dans le repère et la comparer avec les techniques positionnées elles aussi dans ce repère : si les points correspondent, la ou les technique(s) de mise à jour peuvent résoudre son problème.

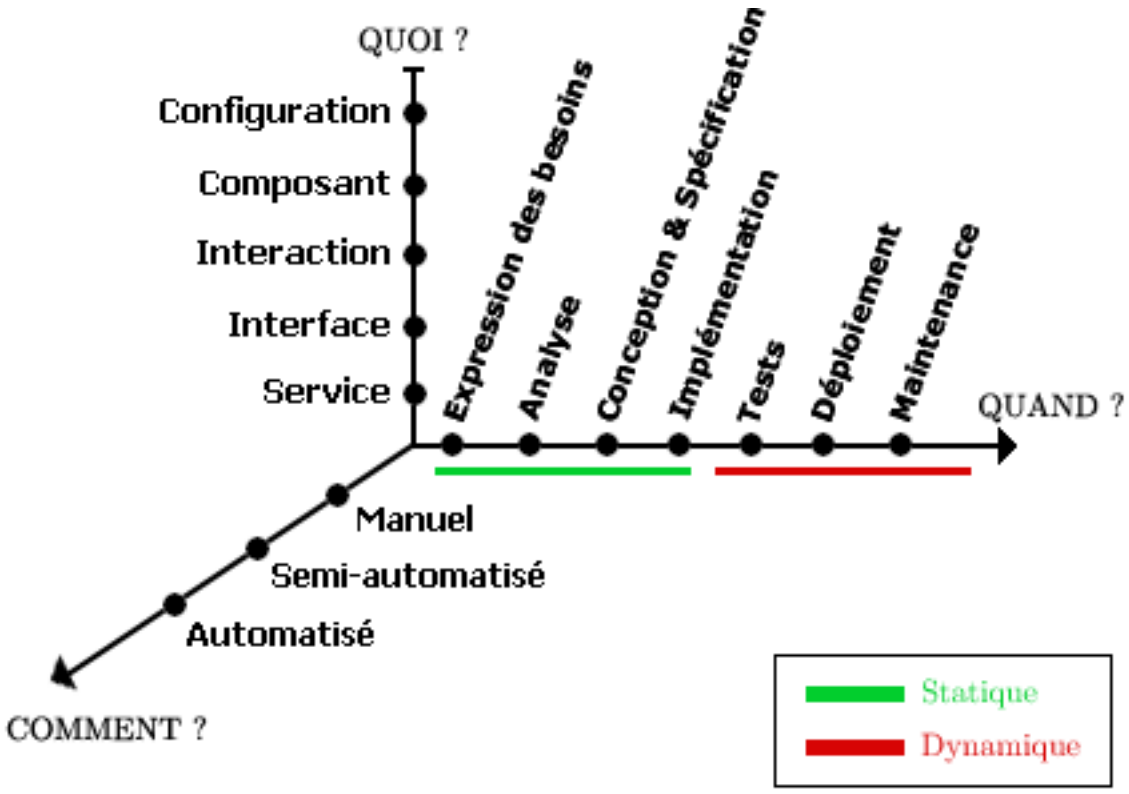

FIG. 2.1 – Les trois dimensions de la mise à jour

Nous allons maintenant présenter les deux techniques de mises à jour qui nous intéressent, à savoir l'évolution et l'adaptation. Chaque présentation situe la place de la mise à jour dans le cycle de vie du logiciel et propose une étude des diverses techniques trouvées dans la littérature en les positionnant suivant nos trois axes :

- **Quoi ?** Configuration, Composant, Interaction, Interface, Service
- **Quand ?** Statique, Dynamique (Cf.moment dans le cycle de vie, section 2.2.1.2)
- **Comment ?** Manuel, Semi-automatisé, Automatisé

2.3 Évolution des composants logiciels

Nous avons défini l'évolution comme une suite d'améliorations apportées aux éléments constitutifs d'un système logiciel. Après avoir brièvement indiqué la place de l'évolution dans le cycle de vie du logiciel, nous listons les principales techniques d'évolution recensées.

2.3.1 L'évolution dans le cycle de vie

On trouve des approches d'évolution statique et dynamique. Le tableau 2.4 montre à quelles étapes du cycle de vie, l'évolution est le plus souvent introduite.

L'évolution peut être prise en compte à différents stade du cycle de vie du logiciel. Cela donne naissance à des approches d'évolution complètement différentes. En effet, l'évolution

	Expressions des besoins	Analyse	Conception & spécification	Implémentation	Tests	Déploiement	Maintenance
EVOLUTION	•	•	•	•			•

TAB. 2.4 – L'évolution dans le cycle de vie

peut être prise en compte dès l'expression des besoins ou de l'analyse[BMBvZ04, JR01]. Cependant, dans cette étude, nous privilégions les techniques proposées à partir de l'étape de conception et spécification. Voici une liste non exhaustives de techniques d'évolution issues de la littérature.

2.3.2 Techniques d'évolution

Nous avons étudié les évolutions supportées par certains ADLs (ACME, C2, Wrigth, Rapide, COSA) et par quelques modèles d'évolutions (SAEV, MAE).

Chaque technique d'évolution est présentée de la manière suivante :

1. Une rapide présentation
2. Son positionnement tridimensionnel (Quoi, Quand, Comment)
3. Les opérations qu'elle offre

Les opérations sont présentées sous forme d'un tableau dont voici le prototype et la légende associé.

	Noeud1	Noeud2	...
Opération1	•		
Opération2		–	
...			

TAB. 2.5 – Tableau prototype

•	Supportée
	Non supportée
–	Indéfinie

2.3.2.1 ACME

Présentation

Cet ADL provient de l'école américaine en 1997, et est présenté comme un langage d'échange d'architecture logicielle. Pour cela, ACME est en fait une base commune pour tous les ADLs, avec 7 types d'entités : les composants, les connecteurs, les systèmes, les ports, les rôles, les représentations, et les cartes de représentation. Il permet des spécifications structurelles, et donc se concentre plutôt sur les architectures statiques pour délaisser leur dynamicité.
L'héritage, la généricité, la composition et le raffinement sont les supports de l'évolution dans ACME.

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interaction, Interface, Service (Structure/comportement)
2. QUAND : statique
3. COMMENT : manuel

	Configuration	Composant	Interaction	Interface	Service
Ajout		•	•	•	•
Suppression		•	•	•	•
Modification	•	•	•	•	•
Substitution		•	•		
Composition		•	•		
Décomposition		•			

Tab. 2.6 – ACME : Les opérations d'évolution

2.3.2.2 C2

Présentation

Issu de l'école américaine et créé en 1996, C2 est présenté comme un ADL gérant les besoins des applications d'interfaces graphiques. Basé sur les messages que s'envoient des composants au sein d'une architecture logicielle, cette forme de communication permet aux composants de fournir des services, ou d'en requérir. Un composant C2 a une interface constituée d'un domaine haut (services requis) et d'un domaine bas (services fournis). Ces 2 ports sont liés réciproquement entre 2 composants. En cas de réception d'évènements (messages) envoyés par un autre composant via ses interfaces, les composants exécutent

des actions. De plus, il faut souligner qu'un composant a une architecture interne et peut donc intégrer un, ou des composants existants. Pour C2, les connecteurs sont vus comme des filtres de messages, comportant des politiques de filtrage. Une configuration C2 est une liste de composants et connecteurs, présentant une topologie comme un assemblage statique de connecteurs et composants.
Le sous-typage et le raffinement sont les supports de l'évolution dans C2.

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interaction, Interface, Service (Structure/comportement)
2. QUAND : statique, dynamique[3]
3. COMMENT : manuel

	Configuration	**Composant**	**Interaction**	**Interface**	**Service**
Ajout		•	•	•	•
Suppression		•	•	•	•
Modification	•	•	•	•	•
Substitution		•	•	•	•
Composition		•	•		
Décomposition		•			

Tab. 2.7 – C2 : Les opérations d'évolution

2.3.2.3 Wrigth

Présentation

ADL créé en 1997 par une université américaine, il a la caractéristique de bien séparer les concepts de composants et connecteurs. Ainsi, les composants Wright sont spécifiés indépendamment des connecteurs Wright. Sa deuxième spécificité est la description comportementale des composants et connecteurs grâce aux notations CSP. Les composants ont des interfaces comportant des ports. Ces derniers sont vus par certains comme des spécifications partielles de composants, car il est possible de spécifier des protocoles d'interaction pour chaque port, avec les notations CSP. Avec la notion de *Computation* d'un composant, qui décrit ce qu'il fait, on a une description complète du comportement d'un composant. Un connecteur Wright est composé d'un ensemble de rôles et d'une glue. Cette dernière définit comment les composants, que relie le connecteur, communiquent

[3]extension SADL

entre eux. Ainsi, on peut dire que les connecteurs Wright sont des entités de première classe. Les configurations sont explicites dans Wright car ce sont des collections de composants et connecteurs, utilisant la notion d'attachement pour décrire la topologie des éléments au sein d'une architecture. De plus, avec la description des comportements des instances de composants et connecteurs, on a une définition dynamique, et l'on peut effectuer des créations et destructions dynamiques d'instances.
La composition est le support de l'évolution dans Wrigth.

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interaction, Interface, Service (Structure/comportement)
2. QUAND : statique, dynamique
3. COMMENT : manuel

	Configuration	**Composant**	**Interaction**	**Interface**	**Service**
Ajout		•	•	•	•
Suppression		•	•	•	•
Modification	•	•	•	•	•
Substitution					
Composition		•	•		
Décomposition					

Tab. 2.8 – Wrigth : Les opérations d'évolution

2.3.2.4 Rapide

Présentation

Cet ADL a été conçu pour pouvoir spécifier au mieux des architectures de systèmes distribués, et de permettre la simulation et l'analyse comportementale d'un système. Pour cela, il fournit des moyens pour exprimer une architecture dans une forme exécutable, et ainsi offrir la possibilité au développeur d'effectuer une simulation avant l'implémentation. Il faut donc capturer le comportement distribué, et les contraintes formelles des architectures. Un composant Rapide contient deux parties : une interface, pour définir les interactions avec les autres composants, et un module, qui encapsule un modèle exécutable du composant. L'interface est explicite car elle définit des actions et services. Les types de composants sont exprimés dans des langage séparés : pour les interfaces, et pour les modules exécutables. On peut dire que les composants dans Rapide sont manipulables et réutilisables. La principale caractéristique des connecteurs est qu'ils ne sont pas des

entités de première classe, ils sont juste des liens directs entre interfaces de composants. Il n'y a pas de types de connecteurs, ils ne sont donc pas réutilisables. La conséquence de connecteurs implicites est que les configurations dans Rapide sont dites en ligne, et spécifient des connexions entre composants à travers des protocoles d'interaction.
Le sous-typage et le raffinement sont les supports de l'évolution dans Rapide.

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interface, Service (Structure/comportement)
2. QUAND : statique, dynamique[4]
3. COMMENT : manuel

	Configuration	**Composant**	**Interaction**	**Interface**	**Service**
Ajout		•	–	•	•
Suppression		•	–	•	•
Modification	•	•	–	•	•
Substitution		•	–		
Composition			–		
Décomposition			–		

Tab. 2.9 – Rapide : Les opérations d'évolution

2.3.2.5 COSA

Présentation

Partant du principe que la modélisation orientée objet et les descriptions architecturales ont des concepts très similaires, ces deux paradigmes – objet et composant– constituent la base de l'ADL COSA [KSO04]. Les composants, les connecteurs et les configurations sont définis par des classes (au sens du paradigme objet) et sont considérés comme entités de première classe. Le composants possède des interfaces pouvant être, soit des services, soit des ports, où les premiers utilisent les second pour être transmis. Les connecteurs possèdent également des interfaces, nommés rôles ou services, ainsi que la réalisation d'un comportement à travers une glue. Une configuration COSA est explicite, et a comme structure interne un graphe d'instances de composants et connecteurs, grâce à la notion d'attachements. Une configuration a aussi des interfaces pour communiquer.
L'héritage et le sont les supports de l'évolution dans MAE.

[4]extension SADL

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interaction, Interface, Service (Structure/comportement)
2. QUAND : statique
3. COMMENT : manuel

	Configuration	Composant	Interaction	Interface	Service
Ajout		•	•	•	
Suppression		•	•	•	
Modification	•	•	•	•	
Substitution		•	•		
Composition		•	•		
Décomposition		•	•		

Tab. 2.10 – COSA : Les opérations d'évolution

2.3.2.6 SAEV

Présentation

SAEV[OTS05] est un modèle proposant de répondre à la problématique de l'évolution des architectures logicielles, qui est de gérer tous les impacts et changements induits par l'évolution d'un élément architectural. Pour cela, SAEV propose une évolution générique de la structure des architectures logicielles grâce à :

- une gestion des évolutions statiques et dynamiques
- une prise en compte de la propagation des impacts d'une évolution d'un élément
- une abstraction de l'évolution, pour la rendre réutilisable, manipulable mais aussi surtout indépendante des langages de description

SAEV ne s'astreint pas à un ADL particulier, mais aux descriptions architecturales en général. Le modèle SAEV est ouvert à toutes nouvelles évolutions et s'y adapte durant l'exécution de celle-ci, pour respecter les contraintes et la cohérence des architectures logicielles qui évoluent. SAEV gère la propagation des changements.
Le gestionnaire d'évolution, les règles et stratégies d'évolutions sont les supports de l'évolution dans SAEV.

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interaction, Interface, Service (Structure/comportement)

2. QUAND : statique, dynamique
3. COMMENT : manuel, automatique

	Configuration	Composant	Interaction	Interface	Service
Ajout		•	•	•	•
Suppression		•	•	•	•
Modification	•	•	•	•	•
Substitution		•	•	•	•
Composition		•	•		
Décomposition		•			

Tab. 2.11 – SAEV : Les opérations d'évolution

2.3.2.7 MAE

Présentation

MAE[RvdHMRM04] propose, pour gérer efficacement l'évolution, un modèle d'architecture couplé à un gestionnaire de configuration (CM). Le fait de combiner les concepts architecturaux et les concepts liés à la gestion de configuration en une même représentation, permet de capturer les changements architecturaux, aussi bien à un niveau de fine granularité (un élément pris individuellement) qu'à un niveau de forte granularité (une configuration). L'environnement MAE permet de spécifier des architecture de manière traditionnelle et de gérer les évolutions via des mécanismes de suivi des changements. Le modèle MAE capture tous les changements par la gestion des versions (versionning) et le sous-typage. Le versionning est utilisé pour identifier différente version d'un même élément ; le sous-typage est utilisé pour annoter chaque changement et la nature de ce changement. Le système MAE peut être implémenté de diverses façons, comme une librairie spécialisée d'un langage répandu comme Java ou C++ par exemple. Par ailleurs, pour ne pas limiter les futures extension de leur modèle, les concepteurs de MAE ont créé xADL 2.0, un ADL basé sur XML. xADL 2.0 est une collection de schémas XML qui représente les éléments architecturaux. MAE gère l'historique et la propagation des changements.
Le versionnement et le sous-typage sont les supports de l'évolution dans MAE.

Positionnement tridimensionnel

1. QUOI : Configuration, Composant, Interaction, Interface, Service (Structure/comportement)
2. QUAND : statique
3. COMMENT : manuel, automatique

	Configuration	Composant	Interaction	Interface	Service
Ajout		•	•	•	•
Suppression		•	•	•	•
Modification	•	•	•	•	•
Substitution		•	•		
Composition		•	•		
Décomposition		•			

TAB. 2.12 – MAE : Les opérations d'évolution

2.3.3 Liste des opérations d'évolution

Nous obtenons la liste des opérations d'évolution en récapitulant les opérations rencontrées au cours de l'études des techniques d'évolution :

	Configuration	Composant	Interaction	Interface	Service
Ajout		•	•	•	•
Suppression		•	•	•	•
Modification	•	•	•	•	•
Substitution		•	•	•	•
Composition		•	•		
Décomposition		•			

TAB. 2.13 – Liste des opérations d'évolutions

Toutes les techniques étudiées, que ce soit les ADLs ou les modèles d'évolution, autorisent les opérations de bases (ajout, suppression, modification) sur les noeuds considérés comme entité de première classe. L'opération de composition est également supportée : elle constitue une opération essentielle du paradigme des composants. La décomposition en revanche, n'est possible que s'il existe des noeuds composites. La modification d'une configuration est synonyme de re-configuration. Enfin, la substitution dépend directement des mécanismes supports de l'évolution.

2.4 Adaptation des composants logiciels

Nous avons défini l'adaptation comme un processus visant à garantir le fonctionnement d'un élément architectural confronté à un nouveau contexte. Comme pour l'évolution, nous indiquons brièvement la place de l'adaptation dans le cycle de vie du logiciel, puis nous listons les principales techniques d'adaptation. Nous conservons la même présentation que pour les techniques d'évolutions.

2.4.1 L'adaptation dans le cycle de vie

On trouve des approches d'adaptation statique et dynamique. Le système a été analysé et spécifié en fonction d'un contexte. Mais ce contexte va probablement changer au cours de la vie du logiciel. L'adaptation intervient donc durant la phase de maintenance du logiciel et éventuellement au moment du déploiement.

	Expressions des besoins	Analyse	Conception & spécification	Implémentation	Tests	Déploiement	Maintenance
ADAPTATION						•	•

TAB. 2.14 – L'adaptation dans le cycle de vie

2.4.2 Techniques d'adaptation

C'est la problématique de l'adaptation d'un composant tiers pour l'intégrer à un système existant qui est la plus traitée dans la littérature. Les techniques étudiées nous ont permis de distinguer deux grandes approches d'adaptation :

1. Une première approche qui consiste à fournir des techniques qui permettent au *client* d'adapter un composant : seul le client du composant produit un effort pour l'adaptation. C'est le cas des *MOPs*, de la *BCA*, de l'*héritage* et du *wrapping*.
2. Une seconde approche où le *fournisseur* du composant autorise une souplesse d'utilisation (configurable, ouvert, ...) de son composant par le client. L'adaptation est facilité car prévue par le concepteur du composant : l'effort d'adaptation est produit/supporté à la fois par le client et le fournisseur. C'est le cas de la *Superposition*, des *Interfaces Actives* et de l'*implémentation ouverte*.

Remarque selon le modèle de composants dans lequel les techniques suivantes s'appliquent, il se peut que certaines opérations présentées ne soient plus possible. En effet certains noeuds peuvent ne pas être considérés comme entité de première classe par le modèle.

2.4.2.1 MetaObject Protocol (MOP)

Présentation

Le besoin adaptation est considéré comme une "préoccupation transverse"[5], qui *cross-cut* le code métier d'une application, et qui doit être traité séparément du reste [DL03, DL04]. Cette approche est inspirée par les concepts de séparation des préoccupations (separation of concerns) et la programmation par aspects [Kea97]. Plus généralement, il est intéressant d'utiliser une approche basée sur les MOPs. Cette technique réflexive va permettre d'adapter dynamiquement un composant en modifiant son comportement, sans forcement accéder au source code : par exemple, certains ont implémenté un mécanisme en Java pour automatiser la construction de wrappers pour adapter un composant tiers [WSSJ98]. Dans cette approche, le méta-programmeur (celui qui programme au méta-niveau) est en charge du développement des méta-objets, donc de l'implémentation des comportements non-fonctionnels désirés.

Positionnement tridimensionnel

1. QUOI : Composant, Interface, Service (Structure/comportement/prop. n.f)
2. QUAND : dynamique
3. COMMENT : manuel, automatique

	Configuration	Composant	Interaction	Interface	Service
Ajout		•		•	•
Suppression					
Modification		•		•	•
Substitution		•		•	•
Composition					
Décomposition					

Tab. 2.15 – MOP : Les opérations d'adaptation

[5]donc considérée comme une propriété non fonctionnelle

2.4.2.2 BCA

Présentation

La BCA (Binary Component Adaptation) est une technique d'adaptation qui applique des adaptations sur les composants binaires sans exiger l'accès au code source[KH98a]. Le système BCA est actuellement mis en application avec Java[6][KH98b]. Un constructeur d'application qui souhaite adapter un composant implémenté en Java construit les spécifications du Δ-file (Delta-file), un fichier contenant des informations sur les changements désirés d'une classe (ajouter ou retirer une interface, une méthode, un champ). On peut même changer les super-classes d'un composant. Un compilateur de Δ-files (DFC) crée un Δ-file binaire qui contient les ajustements nécessaires au byte-code du composant à adapter. Le mécanisme BCA utilise le Δ-file binaire sur le composant binaire pour créer un nouveau composant binaire adapté. Une fois qu'un composant est adapté, d'autres classes qui se rapportent à ce composant, doivent être à leur tour adaptés : la technique BCA gère automatiquement la propagation des changements.

Positionnement tridimensionnel

1. QUOI : Composant, Service (Structure/comportement)
2. QUAND : statique
3. COMMENT : semi-automatique

	Configuration	Composant	Interaction	Interface	Service
Ajout					•
Suppression					•
Modification		•			•
Substitution					
Composition					
Décomposition		•			
Transfert					•

Tab. 2.16 – BCA : Les opérations d'adaptation

Remarque La BCA introduit une nouvelle opération d'adaptation : le transfert ou déplacement.

[6]Pour que le BCA fonctionne, il est nécessaire que le composant binaire contienne suffisamment d'informations de haut niveau (méta-données) pour permettre l'introspection et la modification de sa structure. Le byte-code Java contient de telles données.

2.4.2.3 Héritage

Présentation

L'héritage et la relation de sous-typage associée est intrinsèque à la programmation par objets. Elle constitue tout d'abord une forme d'adaptation, hautement statique, puisque l'on peut ainsi étendre et modifier le comportement d'une classe en ajoutant des attributs ou des méthodes, et pour ces dernières, ajouter une nouvelle définition à celles existantes. Cette redéfinition de méthode, associée à la liaison dynamique, est une pierre angulaire de l'évolution des logiciels. L'adaptation se fait ici par polymorphisme : on substitue une occurrence d'un certain type par une autre, supposée mieux adaptée à une situation différente voire nouvelle. Un système n'est ainsi jamais complètement fermé car on pourra l'étendre et l'adapter par de nouvelles classes répondant, par redéfinition de méthodes existantes, à de nouvelles exigences.

Positionnement tridimensionnel

1. QUOI : Composant, Interface, Service (comportement)
2. QUAND : statique
3. COMMENT : manuel

	Configuration	Composant	Interaction	Interface	Service
Ajout		•		•	•
Suppression					
Modification		•		•	•
Substitution		•		•	•
Composition					
Décomposition					

Tab. 2.17 – Héritage : Les opérations d'adaptation

2.4.2.4 Wrapping

Présentation

La wrapping peut être utilisé pour modifier le comportement d'un composant existant. Un wrapper (enveloppe) est un conteneur qui encapsule totalement le composant et fourni une interface[7] qui augmente ou étend les fonctionnalités du composant initial. Bosh[Bos97] fait la distinction entre le wrapping, où le comportement d'un composant est adapté, de l'agrégation, où les nouvelles fonctionnalités sont composées à partir d'autres composants existants.

[7]puisque le wrapper constitue un nouveau composant

Positionnement tridimensionnel

1. QUOI : Composant, Interface, Service (comportement)
2. QUAND : statique
3. COMMENT : manuel

	Configuration	Composant	Interaction	Interface	Service
Ajout		•		•	•
Suppression					
Modification		•		•	•
Substitution		•		•	•
Composition					
décomposition					

TAB. 2.18 – Wrapping : Les opérations d'adaptation

2.4.2.5 Superposition

Présentation

La superposition (ou superimposition) est une technique d'adaptation qui permet à un constructeur d'application d'adapter un composant en utilisant des types prédéfinis et configurables d'adaptation[Bos97]. Ces types d'adaptation sont beaucoup plus expressifs que BCA mais sont aussi plus complexes. Le principe de la superposition est qu'un composant et la fonctionnalité adaptant ce composant devraient être découplées. La superimposition a été mise en application en utilisant un modèle d'objets en couches[Bos97].

Positionnement tridimensionnel

1. QUOI : Interface, Service (comportement)
2. QUAND : statique
3. COMMENT : manuel

Remarque La superposition introduit une nouvelle opération d'adaptation : le masquage.

	Configuration	Composant	Interaction	Interface	Service
Ajout					
Suppression					
Modification				•	•
Substitution					
Composition					
Décomposition					
Masquage				•	•

Tab. 2.19 – Superposition : Les opérations d'adaptation

2.4.2.6 Interfaces Actives

Présentation

L'interface est programmable, et le client, pour chacune des méthodes de l'interface du logiciel, peut ajouter un appel (callback) vers ses propres composants, avant et/ou après l'appel. La méthode du client peut alors analyser les arguments et les modifier, rediriger l'appel vers une autre méthode, refuser l'appel, exécuter des actions avant/après etc. Le choix des politiques de gestion des messages est donné à celui qui implémente l'interface active, le client peut donc obtenir ce pouvoir.

Positionnement tridimensionnel

1. QUOI : Interface, Service (comportement)
2. QUAND : statique
3. COMMENT : manuel

	Configuration	Composant	Interaction	Interface	Service
Ajout					
Suppression					
Modification				•	•
Substitution					
Composition					
Décomposition					

TAB. 2.20 – Interfaces Actives : Les opérations d'adaptation

2.4.2.7 Implémentation Ouverte

Présentation

Le concept d'implémentation ouverte, introduit dans [Kic96], tente de concilier l'abstraction boîte noire et les besoins en termes de performances[KLM97]. En effet, si le concept de boîte noire permet de faciliter grandement la réutilisabilité du code, c'est bien l'utilisateur qui sait quelle est la meilleure stratégie pour traiter les données qu'il présente au composant. Prenons un exemple simple : un composant permettant de gérer des ensembles d'éléments et disposant d'une interface simple (ajouterElement, retirerElement, trier) peut avoir des stratégies d'implémentation différentes : listes chaînées, B-Arbres, tables de hashages, etc. Seul le client connaît quelle stratégie sera optimale dans son cas. Le client doit donc être en mesure d'agir sur l'implémentation s'il souhaite maximiser les performances, ce qui va à l'encontre de la boîte noire. De plus, agir sur l'implémentation est l'essence même de l'adaptation des logiciels, puisque il s'agit de changer le comportement en fonction de l'environnement d'exécution.
Pour concilier ces deux aspects, les implémentations ouvertes, qui sont les prémices de la programmation réflexive, ont été introduites. Elles se caractérisent par un double jeu d'interfaces : les interfaces primaires qui donnent l'accès aux fonctionnalités (use code) ; les méta-interfaces qui permettent de contrôler l'implémentation (ISC code).

Positionnement tridimensionnel

1. QUOI : Interface, Service (comportement)
2. QUAND : statique
3. COMMENT : manuel

	Configuration	Composant	Interaction	Interface	Service
Ajout					
Suppression					
Modification				•	•
Substitution					
Composition					
Décomposition					

TAB. 2.21 – Implémentation Ouverte : Les opérations d'adaptation

2.4.3 Liste des opérations d'adaptation

Comme pour l'évolution, nous récapitulons l'ensemble des opérations supportées par les techniques d'adaptation que nous avons étudiées. On y retrouve les opérations de l'évolution mais également de nouvelles opérations propres à certaines techniques d'adaptation.

	Configuration	Composant	Interaction	Interface	Service
Ajout		•		•	•
Suppression				•	•
Modification		•		•	•
Substitution		•		•	•
Composition					
Décomposition		•			
Transfert					•
Masquage				•	•

TAB. 2.22 – Liste des opérations d'adaptation

Les techniques d'adaptation se focalisent naturellement sur le composant et proposent essentiellement de modifier le comportement des services et des interfaces qu'il propose. On retrouve à travers ces opérations les deux grands objectifs de l'adaptation : modifier un comportement pour l'adapter au contexte et modifier les interfaces pour faciliter l'interopérabilité avec d'autres composants. Notons l'apparition de deux nouvelles opérations, propres aux techniques d'adaptation étudiées : le transfert (déplacement) et le masquage (restriction) de services.

2.5 Le point sur les mises à jour des composants

2.5.1 Taxonomie globale des opérations

Si nous fusionnons en un même tableau la liste des opérations d'évolution et la liste des opérations d'adaptation, nous obtenons cette taxonomie globale :

	Configuration	**Composant**	**Interaction**	**Interface**	**Service**
Ajout		E/A	E	E/A	E/A
Suppression		E	E	E/A	E/A
Modification	E	E/A	E	E/A	E/A
Substitution		E/A	E	E/A	E/A
Composition		E	E		
Décomposition		E/A			
Transfert					A
Masquage				A	A

Tab. 2.23 – Taxonomie globale : opérations d'évolution et d'adaptation

Légende — E :Évolution, A :Adaptation

Les approches d'évolution et d'adaptation, de par leurs objectifs différents, offrent des opérations de mise à jour qui leur sont spécifiques, même si elles partagent un grand nombre d'opérations. Ainsi, certaines opération sont considérées comme étant des opérations d'évolution (E), des opérations d'adaptation (A), ou les deux (E/A). Ces considérations sont le reflet de l'ensemble des techniques de mise à jour étudiées mais aussi de la sémantique accordée aux notions d'évolution et d'adaptation. Nous l'avons vu, les mises à jour se déclinent en deux approches : l'évolution et l'adaptation. Chacune de ces approches peut être représentée par l'ensemble des opérations qu'elle offre. A l'intersection de ces deux ensembles[8] , on trouve les opérations considérées indifféremment comme des opérations d'évolution ou d'adaptation.

[8]c'est cet ensemble commun qui est la source de l'ambiguïté entre l'évolution et l'adaptation

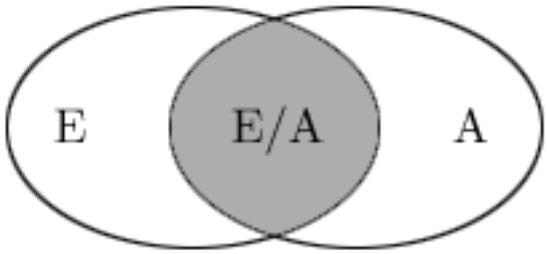

FIG. 2.2 – Représentation ensembliste des opérations de mise à jour

Comme `card(A)<card(E)`, il est légitime, dans un premier temps, d'étudier le passage de l'adaptation vers l'évolution. En particulier, pour exprimer les opérations d'adaptation en fonction des opérations d'évolutions.

2.5.2 Positionnement tridimensionnel : synthèse

Nous avons positionné chaque technique d'évolution et d'adaptation étudiée dans notre repère tridimensionnel. En récapitulant ces informations, nous constatons que la mise à jour dans les architectures à base de composants est majoritairement statique(72%) et manuelle(78%). Ainsi, la mise à jour dynamique et automatisée reste une problématique ouverte et encore peu traitée, qui offre de nombreuses perspectives. C'est d'ailleurs l'objectif ultime à atteindre : les systèmes auto-évolutif et auto-adaptatif.

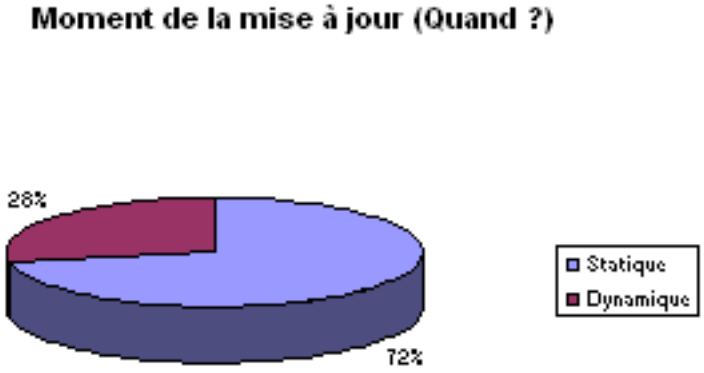

FIG. 2.3 – Moment de la mise à jour : synthèse

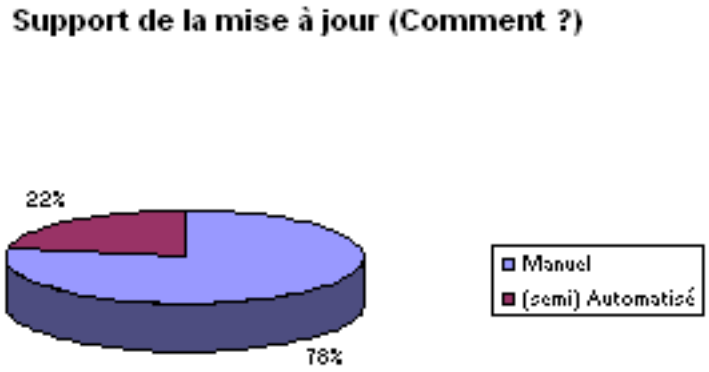

FIG. 2.4 – Support de la mise à jour : synthèse

2.5.3 Au coeur de notre étude : la réutilisation

Notre étude des mises à jour et de ses techniques crédite et étends encore davantage le concept de réutilisation dans les architectures à base de composant. Nous considérons la réutilisation à deux niveaux (Figure 2.5) :

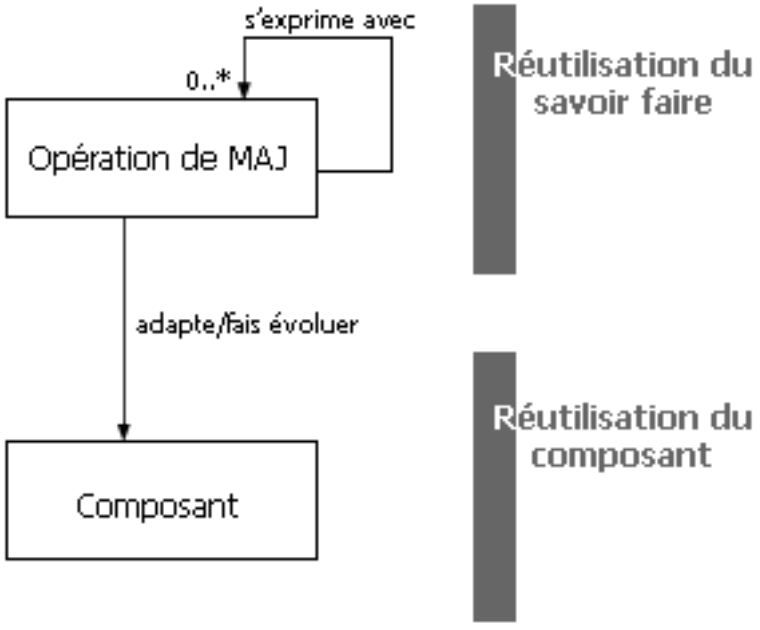

FIG. 2.5 – Les deux niveaux de la réutilisation

Réutilisation du composant — La notion de composant ou de brique logicielle est intrinsèquement lié à la notion de réutilisation. Mais nous avons vu que la réutilisation est, le plus souvent, soit impossible directement (c'est souvent le cas des composants sur

étagères), soit éphémère dans le cycle de vie du logiciel (de nouveaux besoins apparaissent sans cesse). Le paradigme composants logiciels, dont le coeur est la réutilisation, ne prend véritablement son sens qu'en intégrant le concept de mise à jour. En d'autres termes, la réutilisation des composants est d'autant plus efficace que la logique de mise à jour est supporté par le système.

$$(\textit{ Composants + logique de mise a jour }) \Downarrow \textit{Réutilisation}$$

Réutilisation du savoir faire — Dans un système offrant des opérations de mise à jour, lorsque que naît un besoin de mise à jour, si cette opération n'est pas déjà offerte par le système, nous préconisons que l'architecte exprime son opération de mise à jour en fonction des opérations disponibles. Sa nouvelle opération est une combinaison et une réutilisation d'opérations existantes. Si cette réutilisation n'est pas possible, l'architecte doit développer son opération ad hoc qui vient enrichir le système et qui constitue à son tour une opération potentiellement réutilisable. La figure 2.6 illustre cette démarche.

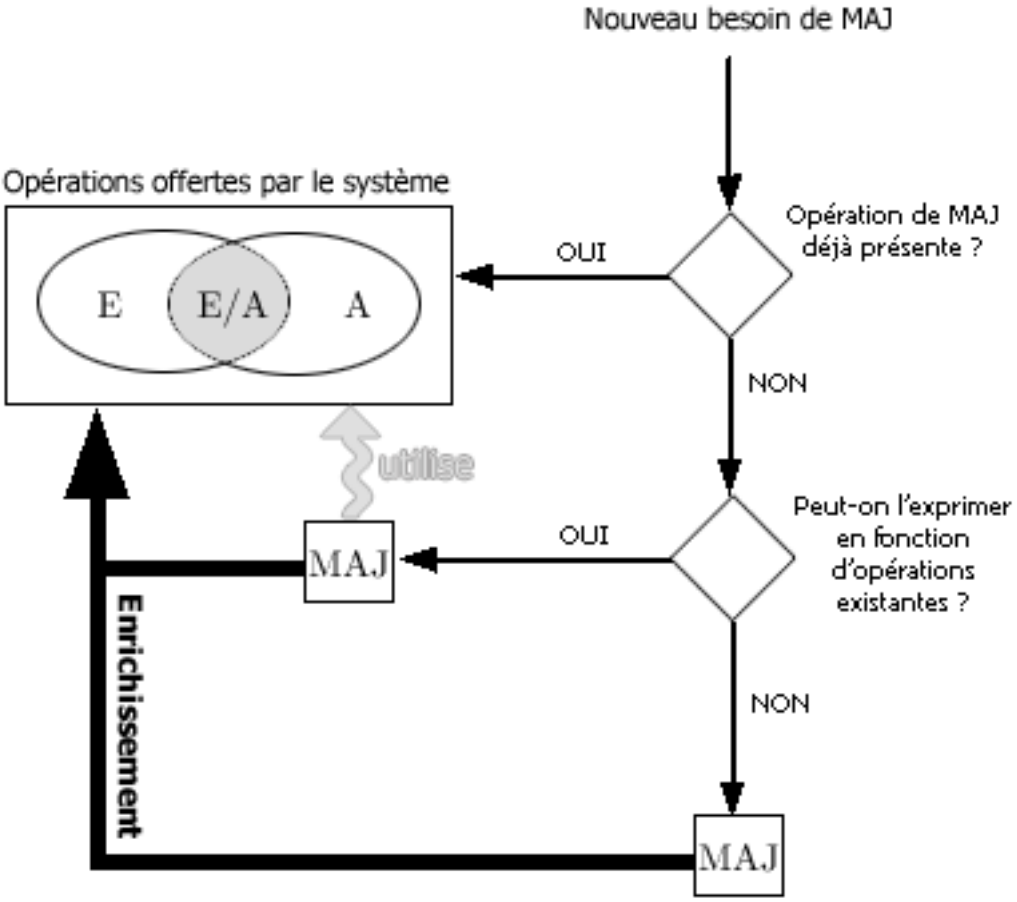

FIG. 2.6 – Réutilisation du savoir faire

Chapitre 3

LIENS ENTRE ADAPTATION ET ÉVOLUTION

L'intitulé de notre étude, "De l'adaptation vers l'évolution. . ." présuppose qu'il existe une relation entre l'adaptation et l'évolution. Nous définissons dans ce chapitre les liens qui existent entre l'adaptation et l'évolution, et qui sont, à nos yeux, de deux types :

- lien de description (*i*)
- lien de conséquence (*ii*)

Notre objectif est donc double : (*i*) montrer qu'une opération d'adaptation peut être exprimée en fonction d'opérations d'évolutions, (*ii*) et qu'une opération d'adaptation peut déclencher d'autres opérations de mise à jour afin de respecter des contraintes architecturales[1].

Nous avons vu précédemment quelles étaient les trois vues d'une mise à jour (Section 2.2.1.1) : le comportement, les propriétés non fonctionnelles et la structure. Dans le cadre de notre sujet, nous nous restreignons désormais à l'adaptation structurelle d'un composant, étudiée à l'Université de Nantes dans [BSO05], qui est l'adaptation du composant sous son angle structurel.

Après avoir présenté l'adaptation structurelle, nous tentons de ré-écrire une opération d'adaptation structurelle, en l'occurrence une décomposition d'un composant monolithique, en une combinaison d'opérations d'évolution pour illustrer la réutilisation du savoir faire par la mise en évidence du lien de description. Puis, nous proposerons différents scénarios pour montrer comment une opération d'adaptation structurelle déclenche d'autres opérations de mise à jour afin d'illustrer le lien de conséquence.

[1] ces contraintes peuvent être de toutes sortes

3.1 Adaptation Structurelle

3.1.1 Définition

Nous définissons l'adaptation structurelle comme l*a possibilité de modifier la structure d'un composant*. En effet, l'adaptation structurelle ne s'intéresse pas à modifier le comportement, mais plutôt à réorganiser la structure de celui-ci.

Nous avons vu que les concepts du paradigme composant peuvent être assimilés aux noeuds d'un graphe (Section 2.2.1.1). Ces noeuds sont reliés entre eux par des arcs. En particulier, ces liens peuvent être structuraux (liens de composition), c'est ceux qui nous intéressent ici. La structure d'un composant peut ainsi être représenté par un *graphe structurel*.

Supposons qu'un composant possède trois interfaces (notées Itf). La première interface définit deux services, la deuxième et la troisième interface définissent chacune un unique service. Nous pouvons représenter ce cas par un graphe, comme le montre la figure 3.1.

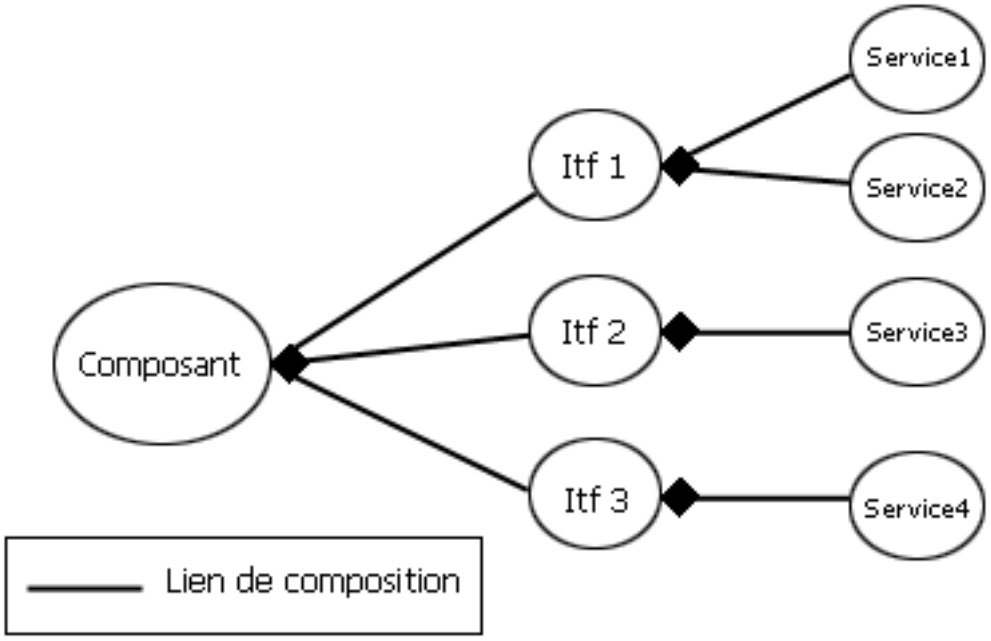

FIG. 3.1 – Graphe structurel d'un composant

Par exemple, l'adaptation structurelle peut consister en la *mise à jour de l'ensemble des interfaces* ou à *changer l'ensemble des sous-composants* du composant à adapter. Aussi, une forme d'adaptation structurelle est la *décomposition d'un composant en un ensemble constitué d'autres composants* où chacun de ces derniers offre une partie des services du composant initial et où *l'assemblage de ses nouveaux composants permet d'offrir la totalité des services offerts par le composant adapté.*

L'adaptation structurelle agit donc sur la vue structurelle des noeuds. Ces noeuds peuvent être : le composant, l'interface, le service. Nous considérons, dans cette étude, l'adaptation structurelle comme un processus de mise à jour statique et manuel. Comme toute opération de mise à jour, nous pouvons positionner l'adaptation structurelle dans notre repère tridimensionnel :

Positionnement tridimensionnel

1. QUOI : Composant, Interface, Service [vue = structure]
2. QUAND : statique
3. COMMENT : manuel

3.1.2 Opérations

Dans notre état de l'art sur les techniques d'adaptation recensées dans la littérature, nous avons listé dans un tableau les opérations d'adaptation offertes, proposé en section 2.4.3. Suite à notre définition de l'adaptation structurelle, nous pouvons dégager de ce tableau, le sous-ensemble d'opérations que nous considérons comme étant des opérations d'adaptation structurelles :

	Composant	**Interface**	**Service**	**Interaction**
Ajout		•	•	
Suppression		•	•	
Modification	–	–	–	–
Substitution	–	–	–	–
Composition	–	–	–	–
Décomposition	•			
Transfert			•	
Masquage		•	•	

Tab. 3.1 – Opérations d'adaptation structurelle

Légende :

•	Supportée
	Non supportée
–	Indéfinie

Remarque Les opérations de modification, de substitution et de composition n'ont pas de sens pour l'adaptation structurelle. Nous les avons donc considérées comme indéfinies.

3.1.3 Objectifs

Selon [BSO05], si l'adaptation d'un composant, en général, permet l'ajustement d'une des vues (comportement, propriétés non fonctionnelles, structure) de ce dernier pour une meilleure adéquation avec les besoins de son utilisation, l'adaptation structurelle permet entre autre :

L'adaptation du déploiement des composants logiciels

Le déploiement d'un composant est dépendant de la structure de celui-ci. 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 donc séparable. Ainsi, l'adaptation structurelle permet un déploiement flexible d'un composant logiciel. Ce déploiement pourra être adapté à la configuration de l'environnement qui peut être centralisé, distribué, sous forme de grappe (i.e. cluster), etc... Il est possible d'assurer cette propriété en permettant 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 différentes.

Adaptation par rapport aux ressources disponibles (optimisation)

L'infrastructure d'exécution d'un composant peut, par exemple, permettre ou non des exécutions parallèles, disposer ou non des ressources nécessaires pour l'installation et l'exécution de l'ensemble des services de ce composant. Ainsi, dans le cas d'infrastructure permettant des exécutions parallèles (e.g. multiprocesseurs, cluster), il serait possible, grâce à l'adaptation structurelle, de partager l'ensemble des services du composant en différents sous-ensembles définis par différents sous-composants pouvant être exécutés comme des processus parallèles. Aussi, dans le cas de ressources non satisfaisantes ou limitées, il serait possible, dans une première étape, de définir certains services du composant (les moins critiques) dans une nouvelle entité (sous-composant) créée par adaptation structurelle et ensuite de déconnecter ce dernier du reste de l'application.

Dans le cas de l'adaptation structurelle statique, celle-ci est décidée et réalisée par le constructeur d'application qui est conscient des ressources disponibles. Dans le cas d'une adaptation structurelle dynamique, c'est au système lui-même d'être conscient des ressources disponibles[2], et comme ces systèmes adaptables se basent sur la perception qu'ils ont de leur environnement, le mécanisme de surveillance de l'environnement est essentiel pour assurer la qualité de l'adaptation.

L'adaptation pour l'interfaçage avec composants structurellement hétérogènes

L'interfaçage 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, bien 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 peut être due au fait que les composants logiciels, structurant leurs services en interfaces, peuvent proposer d'organiser ces services de manières différentes (interfaces différentes), de sorte que les interfaces requises et fournies ne peuvent pas être mises en correspondance[3].
Dans ce cas, l'adaptation structurelle permet de restructurer les services des composants assemblés en redéfinissant les interfaces requises et fournies. Cette mise à jour de la structure des interfaces permet, ainsi, d'offrir une correspondance exacte entre les interfaces

[2]C'est la notion de "context-aware"

[3]problème souvent réglé par l'utilisation de wrappers

des composants. Cette adaptation réalisée sans changement de la sémantique des services des composants leur permet d'être correctement assemblés.

Séparation des préoccupations métiers

Les fonctionnalités d'un composant logiciel représentées par ses interfaces peuvent varier et correspondre à différentes vues ou préoccupations par rapport à leurs utilisations (et donc par rapport à leurs utilisateurs). Par exemple, un composant logiciel permettant la gestion d'un établissement universitaire peut être restructuré en différentes entités structurelles correspondant chacune à une vue/préoccupation métier telle que l'enseignement, la recherche ou l'administration dans l'établissement en question. Les nouvelles entités structurelles sont définies et créées à travers un processus d'adaptation structurelle pour inclure l'ensemble des services nécessaire à chaque préoccupation prise séparément.

3.2 Lien de description : la vue statique

Dans le cadre de cette étude, notre démarche consiste à montrer qu'une opération d'adaptation peut s'exprimer en fonction d'opérations d'évolution [4] :

```
opération d'adaptation = fct°(opérations d'évolution)
```

On choisi comme adaptation structurelle la *décomposition structurelle* d'un composant primitif[5] afin de l'éclater en d'autres composants qui pourront être déployés sur des sites différents, par exemple pour des problématiques de performances (Figure 3.2). Cette décomposition ne modifie pas les services proposés par le composant, elle permet simplement d'adapter le composant à son nouvel environnement distribué.

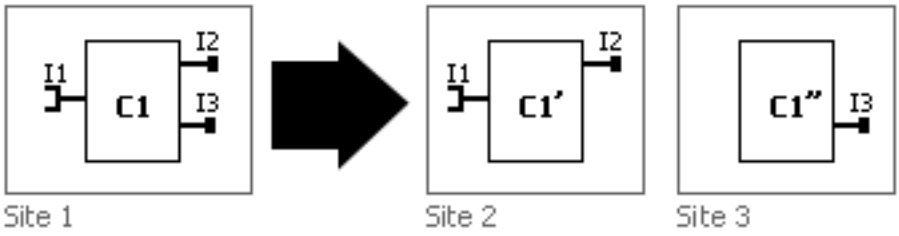

FIG. 3.2 – Décomposition structurelle d'un composant pour son déploiement

[4] la relation inverse est tout à fait possible

[5] cette décomposition d'un composant monolithique est plus complexe que le décomposition d'un composant composite

Quatre étapes sont nécessaires pour mener à bien cette décomposition structurelle, telle qu'elle est présentée dans [BSO05]. Nous tentons d'établir les étapes d'évolutions équivalente aux étapes de l'adaptation structurelle.

Étape	**Adaptation struct.**	**Évolution**
1	La spécification de l'adaptation du composant	Identifier quels noeuds vont être concernés
2	La mise à jour de la structure du composant	Ajouter les nouveaux composants à l'architecture
3	L'assemblage des nouveaux éléments structuraux résultants de la mise à jour	Ajouter les interactions entre les nouveaux composants
4	L'intégration des nouveaux éléments structuraux à l'application	Ajouter les interactions entre les nouveaux composants et le reste de l'application

TAB. 3.2 – Étapes de la décomposition structurelle

Remarque : La garantie de l'intégrité structurelle et fonctionnelle des sous-composants issus de l'"éclatement" du composant d'origine est gérée respectivement par les liens comportementaux et structuraux entre les entités (graphe de dépendance). Ces liens sont les arcs que nous avons définis en Section 2.2.1.1.

De ces quatre étapes, nous faisons ressortir deux grandes phases :

1) Formulation de l'objectif (Étape 1) La spécification de la décomposition structurelle permet de lister les noeuds qui seront ajoutés à l'architecture mais les arcs/liens entre ces noeuds représentent des dépendances qui vont nécessiter, par propagation, l'identification et l'ajout de nouveaux noeuds. Un exemple :

- Un composant C possédant une interface I1 est structurellement lié à cette dernière. Les noeuds C et I1 sont identifiés. NOEUDS={C,I1}
- Si cette interface I1 est composée d'un service s1 qui fait appel à un service s2 offert par une interface I2, alors les interfaces I1 et I2 sont liées par un lien de dépendance comportementale. Par propagation, le noeuds I2 est identifié. NOEUDS={C,I1} $\cup$ {I2}
- Ainsi de suite...

2) Mise à jour (Étapes 2,3 et 4) Une fois tous les noeuds identifiés, il suffit d'utiliser des opérations d'évolution pour les ajouter à l'architecture. Finalement, les opérations requises par l'adaptation structurelle d'un composant sont des opérations d'évolution.

Il est donc bien possible d'exprimer certaines opération d'adaptation (ici structurelle) en fonctions d'opérations d'évolutions offertes par le système. C'était notre premier objectif.

3.3 Lien de conséquence : la vue dynamique

Idéalement, l'adaptation structurelle d'un composant n'a qu'un impact local (Cf. Section 2.1.2). En réalité, l'adaptation structurelle d'un composant peut nécessiter la mise à jour d'autres éléments logiciels, pour respecter des contraintes. Une opération d'adaptation structurelle sur un composant est le plus souvent suivie d'autres opérations : conformément à notre classification, ces opérations sont considérées comme des opérations d'adaptation, d'évolution, ou les deux.

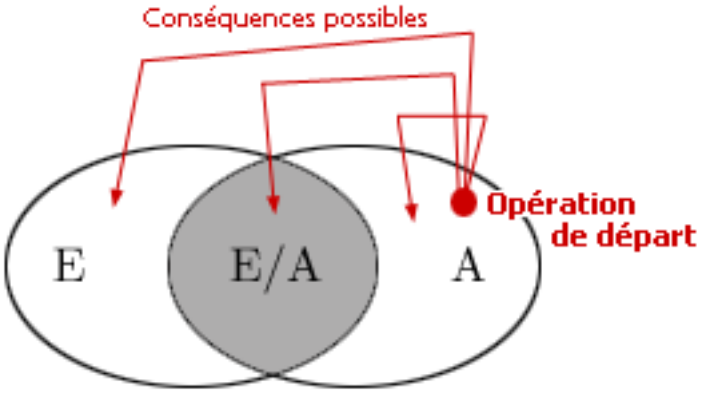

FIG. 3.3 – Les trois conséquences possibles d'une adaptation structurelle

Nous nous proposons d'illustrer chacune des trois conséquences (E, E/A, A) par un petit scénario. Il s'agit d'un exemple, une situation plausible à laquelle peut être confronté l'architecte logiciel et qui est représentatif de notre problématique. Pour chaque exemple, nous partons systématiquement d'une opération de décomposition structurelle sur un composant "Agenda partagé", et proposons une opérations de mise à jour qui en est la conséquence.

3.3.1 Exemple support : l'agenda partagé

L'"Agenda partagé" est un composant qui a été développé par Gautier Bastide pour sa thèse en cours [Gau04]. Ce composant constitue, pour nous aussi, un exemple support. Voici les services offert par ce dernier :

- L'interface *Réunion* pour la gestion des réunions de l'agenda : ajout, consultation, confirmation, ...
- L'interface *Absence* pour la gestion des absences d'une personne : ajout, consultation, ...
- L'interface *serveur* pour la gestion de l'agenda : création (instanciation), consultation, ...

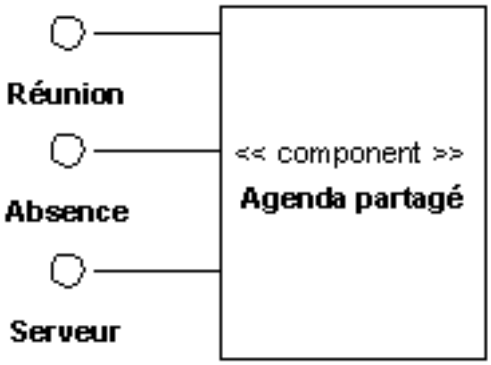

FIG. 3.4 – Agenda Partagé (Notation UML 2.0)

Pour les scénarios qui suivent, les figures représentent uniquement les composants et leur interfaces (Notation UML 2.0). Pour plus de clarté, les interactions entre ces composants et le reste du système ne sont volontairement pas représentés.

3.3.2 De l'adaptation structurelle à l'évolution (E)

Scénario : prenons le cas d'une petite application pour gérer l'organisation au sein d'une entreprise. Certains choix architecturaux sont garantis sous forme de contraintes [TFS04, CC04], grâce à une syntaxe et une sémantique particulière. En particulier, l'architecte souhaite bien séparer les préoccupations. Dans son architecture, nous retrouvons notre composant Agenda ainsi qu'un composant de gestion du personnel de l'entreprise offrant deux interfaces : *Employés* (qui travaille ?) et *Postes* (à quel poste ?).
Le composant Agenda rempli parfaitement son rôle mais il propose l'interface *absence* qui relève du domaine de la gestion du personnel, ce qui viole ses contraintes de séparation des préoccupations. Notre architecte décide alors d'effectuer une décomposition structurelle sur le composant Agenda (vue en section 3.2) afin de produire deux nouveaux composants aux préoccupations bien séparées : l'agenda, la gestion du personnel.
Malgré cette adaptation structurelle, les choix architecturaux ne sont toujours pas respectés : l'application contient deux composants relevant du domaine de la gestion du personnel. Il faut effectuer une dernière opération : composer les deux composants en un seul composant englobant (wrapper). Cette opération d'évolution a donc été déclenchée en conséquence de l'opération d'adaptation structurelle initiale, et requise par des contraintes architecturales.

Opération d'adaptation exécutée	**Opération d'évolution déclenchée**
Décomposition structurelle	Composition

FIG. 3.5 – a) situation initiale

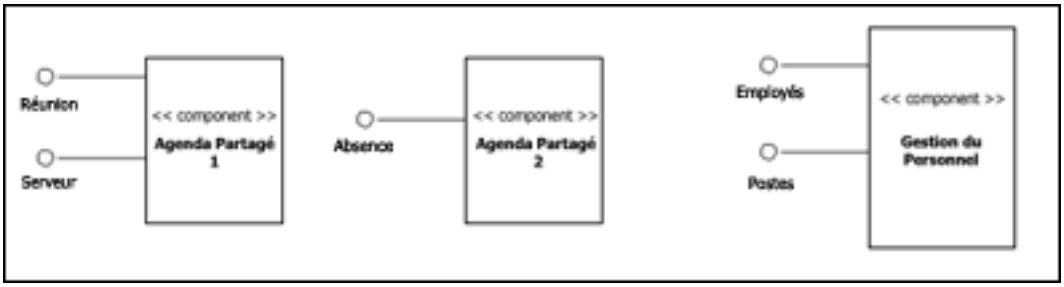

FIG. 3.6 – b) après décomposition structurelle

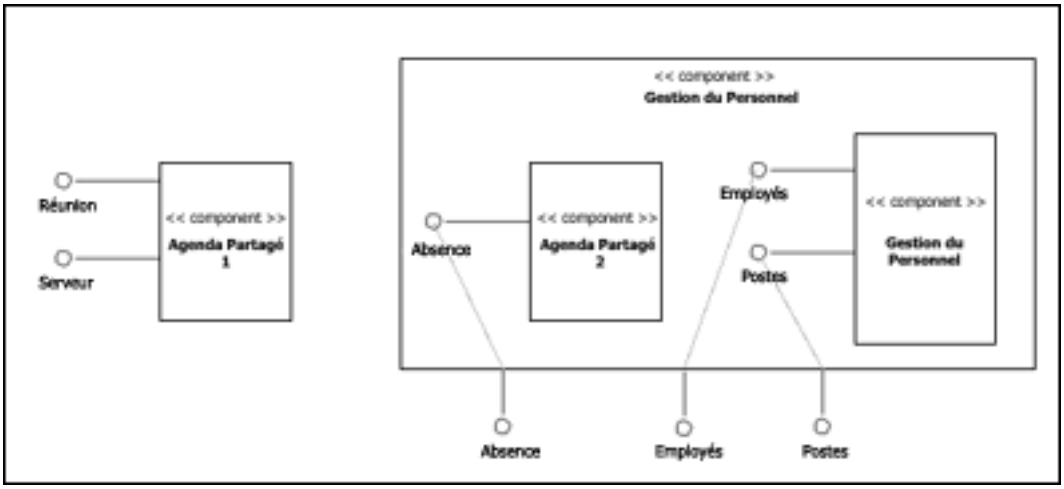

FIG. 3.7 – c) après composition

3.3.3 De l'adaptation structurelle à l'adaptation (A)

Scénario : reprenons exactement la même architecture que pour le scénario précédent. Si le composant pour la gestion du personnel propose déjà une interface pour la gestion des absences, mais moins complète que celle du nouveau composant issu de la décomposition structurelle de l'agenda, on décide de masquer l'interface *absence* du composant *Gestion du personnel* au profit de celle du nouveau composant.

Opération d'adaptation exécutée	Opération d'adaptation déclenchée
Décomposition structurelle	Masquage d'une interface

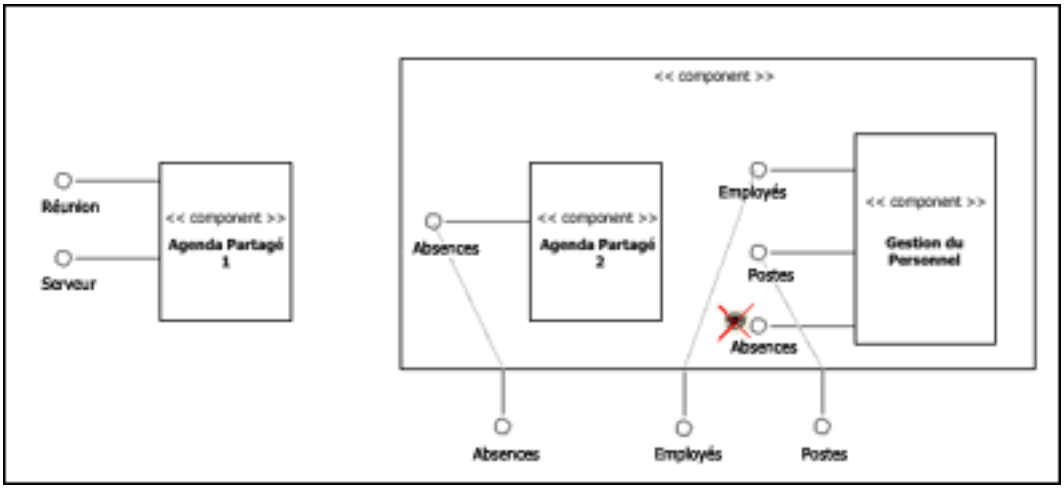

FIG. 3.8 – Masquage d'une interface

3.3.4 De l'adaptation structurelle à l'évolution/l'adaptation (E/A)

Scénario : supposons une architecture dans laquelle chacun des composants doit proposer une interface particulière *lifeCycle* (pour gérer l'état du composant par exemple), conformément au souhait de l'architecte. Notre composant Agenda fait partie de cette architecture et possède cette interface (elle aura été ajouté au préalable par l'architecte). La décomposition structurelle appliquée au composant Agenda produit deux nouveaux composants, mais un seul possède l'interface *lifeCycle*[6]. Or chaque composant doit avoir sa propre interface *lifeCycle*, il faut donc ajouter cette interface manquante au composant qui ne la possède pas.

Opération d'adaptation exécutée	Opération d'évolution/adaptation déclenchée
Décomposition structurelle	Ajout d'une interface

[6]car la décomposition structurelle conserve exactement les services du composant d'origine : rien de plus, rien de moins

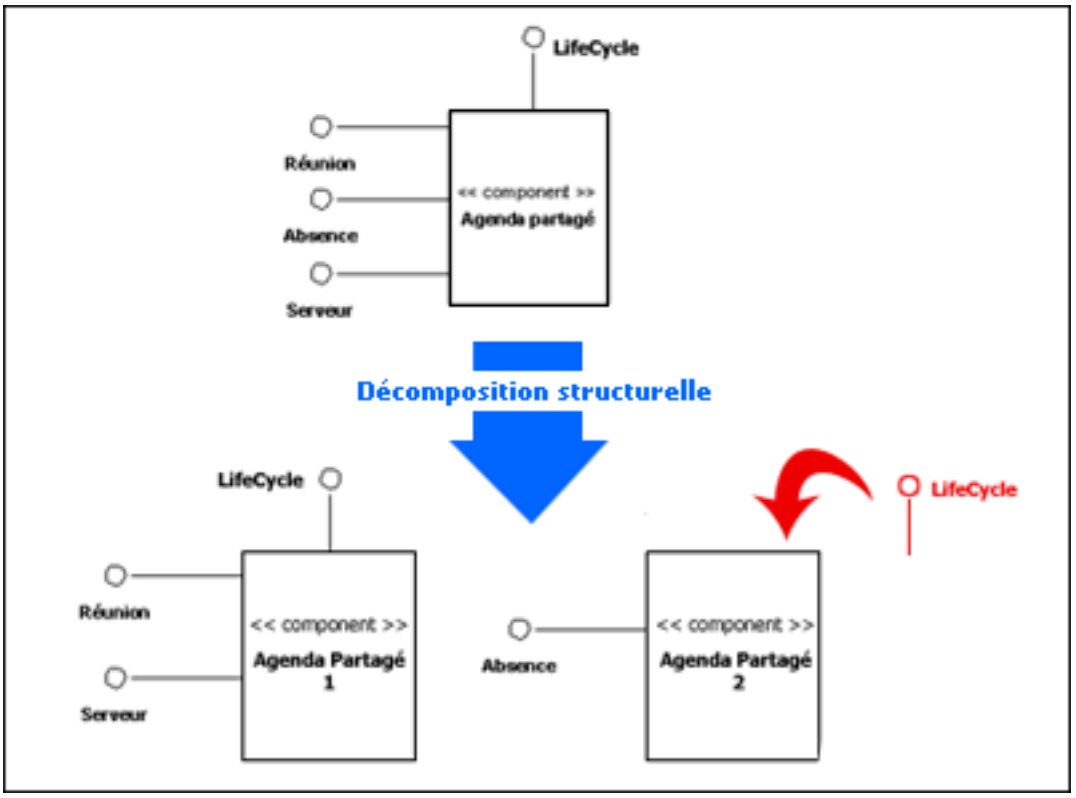

FIG. 3.9 – Ajout d'une interface

3.4 Bilan

Les petits scénarios précédents illustrent les conséquences de l'adaptation en termes de mises à jour sur une architecture où des contraintes sont exprimées. Ces mises à jour sont déclenchées pour ne pas laisser le système dans un état incohérent ou qui violerait les contraintes. Seule l'adaptation initiale est déclenchée manuellement par l'architecte, les autres mises à jour en sont les conséquences et doivent donc – idéalement – être déclenchées et réalisées de manière automatisée.

Nous nous proposons de valider notre étude par une application pratique sous le modèle Fractal, et ainsi d'implémenter un noyau d'opération d'adaptation vérifiant nos deux propriétés (lien de description et lien de conséquence) :

1. Une opération d'adaptation peut s'exprimer en fonction d'opérations d'évolutions
2. Une opération d'adaptation peut déclencher d'autres mises à jour

Chapitre 4

MISE EN PRATIQUE SOUS FRACTAL

4.1 Le modèle Fractal : présentation

Fractal est un modèle de composants conçu conjointement par France Telecom R&D et l'INRIA Rhône-Alpes dans le cadre du consortium ObjectWeb en Janvier 2002. Ses mots d'ordre sont modularité, extensibilité et réflexivité. Un des objectifs principaux de Fractal est de fournir un support pour la construction et la configuration dynamique de systèmes. Fractal est un modèle suffisamment général (il n'est pas orienté vers un domaine d'application particulier) pour pouvoir être utilisé de différentes manières [BCS04] comme :

- Support à la définition et la configuration : Fractal est une base pour les différents langages et outils tel que les outils de configuration graphique, les langages de description d'architectures (ADLs) et leur compilateur et outils associés, ...
- Modèle général de composition : Fractal permet de nombreux types de composition tels que la composition structurelle, opérationnelle, comportementale, ...
- Support d'administration : Fractal peut être étendu aux modèles de composants administrables et utilisé comme une base pour la création d'outils d'administration (supervision, diagnostiques, etc.)

4.1.1 Spécifications

A la base, un composant fractal est formé d'une membrane (contrôleur) et d'un contenu. Les composants peuvent être imbriqués, d'où les notions de sous-composants et de composants composites.

4.1.1.1 Contrôleur

Le contrôleur peut implémenter un nombre variable d'interfaces de contrôle dont un certain nombre d'interfaces sont définies dans la spécification du modèle à composants. Le contrôleur agit sur le contenu.

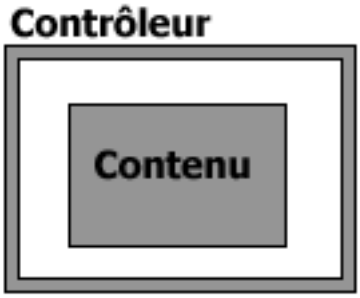

FIG. 4.1 – Un composant Fractal

Parmis les interfaces élémentaires de contrôle qui permettent la configuration d'un composant Fractal, on peut citer :

- *NameController* : permet d'associer un nom à chaque composant.
- *AttributeController* : permet la configuration de l'état du composant. Il contient les opérations de lecture et d'écriture pour chaque attribut.
- *BindingController* : pour connecter de déconnecter les composants (d'une interface client vers une interface serveur).
- *ContentController* : pour gérer les sous-composants d'un composant composite. (lister, rajouter et supprimer des sous-composants d'un contenu du composite)
- *LifeCycleController* : cette interface de contrôle est très importante pour la gestion de cycle de vie du composant. Elle contient les méthodes permettant de démarrer ou arrêter l'exécution d'un composant.
- *SuperController* : permet de s'informer sur le composant père (composite) qui contient les différents sous-composants.

4.1.1.2 Contenu

Peut contenir du code fonctionnel ou bien être un ensemble d'autres contrôleurs ; de cette manière, le modèle supporte la composition hiérarchique. On trouve ainsi les trois notions de composant primitif, composite et partagé :

- *Primitifs* : un composant Fractal qui encapsule du code exécutable.
- *Composites* : un composant Fractal qui contient un ensemble de composants avec un niveau d'emboîtement quelconque.
- *Partagés* : un composant Fractal qui appartient simultanément à des composants composites.

4.1.1.3 Interfaces

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : les interfaces métiers et les interfaces de contrôle (la membrane est l'ensemble de ces interfaces de contrôle). L'ensemble des interfaces métiers et de contrôle d'un composant définit son type. Les interfaces métiers sont les points d'accès externes au composant.

Une interface Fractal métier est du type *client* ou *serveur* : une interface *serveur* reçoit des opérations d'invocation alors qu'une interface *cliente* peut en émettre. De cette manière on définit une *liaison* (binding) Fractal comme une connexion entre deux composants. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée.

4.1.1.4 Type de composant

Le type de composant Fractal est défini par l'ensemble des types de ses interfaces qui sont :

1. Le **nom** de l'interface
2. La **signature** qui représente le type du langage
3. Le **rôle** de l'interface qui est soit client, soit serveur
4. La **contingence** qui spécifie une garantie sur la disponibilité des fonctionnalités de l'interface (*obligatoire*/*optionnel*)
5. La **cardinalité** de l'interface selon sa liaison : *singleton* ou *collection*

4.1.2 Exigences du modèle Fractal

La spécification du modèle à composant Fractal est motivée par sept exigences essentielles [BCS04] à la réalisation d'un modèle à composant général. C'est l'association des ces exigences qui dicte le modèle à composant général Fractal.

4.1.2.1 Encapsulation, abstraction, identité

- L'encapsulation signifie qu'un composant doit interagir avec son environnement, uniquement à l'aide de points d'accès et d'opérations prédéfinies.
- L'abstraction signifie qu'un composant ne doit pas révéler plus que nécessaire les détails de son implémentation aux entités de son environnement.
- L'identité signifie d'un composant doit avoir une identité permettant de la désigner de façon non ambiguë par rapport aux autres composants.

4.1.2.2 Composition

C'est la possibilité de composition (structurelle ou opérationnelle) et d'assemblage des composants permettant de créer de nouveaux composants de plus haut niveau. La composition doit être explicitement maintenue et modifiable à l'exécution.

4.1.2.3 Partage

L'exigence d'une certaine forme de partage de composant est dictée par les motivations de Fractal que sont la description et la construction de configurations de systèmes (c'est-à-dire l'assemblage de composants). Par exemple, le partage d'un composant processeur entre différents processus...

4.1.2.4 Cycle de vie et déploiement

La quatrième exigence est la gestion du cycle de vie et des différents aspects du déploiement des composants (incluant les différentes formes d'installation, d'instanciation, d'initialisation, d'activation, etc.). Cette exigence découle directement du besoin d'automatisation du processus de gestion de systèmes à grande échelle.

4.1.2.5 Manipulation d'activité

La cinquième exigence est la nécessité de rendre explicite et de supporter la manipulation d'activités mis en oeuvre dans un système (processus, threads, transactions, ...) et pouvant impliquer plusieurs composants. Cette exigence se rapporte à la deuxième et la troisième en ce sens qu'une activité peut être perçue comme une composition d'autres activités (i.e composants) elles-mêmes partagées par d'autres.

4.1.2.6 Contrôle de comportement

Cela concerne la possibilité de créer différentes formes de composants de contrôles, c'est-à-dire des composants qui fournissent des capacités d'introspection pour observer un ensemble de (sous)composants et exercer un certain contrôle sur leur exécution.

4.1.2.7 (Re)Configuration et mobilité

C'est-à-dire que le contenu et les dépendances de ressources des composants doivent pouvoir évoluer dans le temps (de façon spontanée ou en réaction à diverses interactions de composants). Cette évolution doit pouvoir se faire de façon dynamique[1] à l'exécution.

4.1.3 Implémentation type : JULIA

Julia, comme Think[2], proActive[3] ou encore FracTalk[4] sont des implantations du modèle de composants Fractal. Julia se distingue d'être une implémentation de référence en Java de la spécification Fractal développée par France Télécom, elle permet de décrire le contrôleur à partir d'un langage spécialisé et réalise ensuite un mélange du code de contrôle et du code fonctionnel à travers une approche à mixins. Un des aspects intéressants de la plate-forme Julia est qu'elle n'impose pas un nombre fixe d'interfaces de contrôle [Bru04].

4.1.3.1 Structure d'un composant Julia

Tout composant Julia suit les spécifications du modèle concret de Fractal. Un composant possède donc au moins l'interface de contrôle *NameController* qui fournit l'identifiant du composant. Si le composant est un composite il possède en plus l'interface *ContentController*. Toute interface additionnelle est facultative. La membrane d'un composant

[1] implique la gestion du cycle de vie (Interface *LifeCycleController* !)
[2] `think.objectweb.org`
[3] `proactive.objectweb.org`
[4] `csl.ensm-douai.fr/FracTalk`

Julia peut contenir deux types d'entités : des objets de contrôle et des objets d'interception. Les objets de contrôle sont accessibles grâce à une référence d'interface de contrôle (i.e *NameController*). Un objet d'interception (ou intercepteur) est une sorte d'objet de contrôle qui intercepte des appels de méthodes [Bru04].

4.2 Projection de notre approche vers Fractal

Nous avons établi qu'il existe des liens, de description et de conséquence, entre les opérations d'adaptation et les opérations d'évolution. Nous présentons, dans cette section, une réflexion sur les possibilités offertes par le modèle Fractal pour mettre en oeuvre quelques opérations d'adaptation structurelles.

4.2.1 Hypothèses de travail

4.2.1.1 Processus d'adaptation

Nous nous positionnons dans le contexte suivant : un composant doit être adapté pour correspondre à un environnement ou un besoin particulier. Nous optons pour une approche de type "boite noire"[5] et statique : nous travaillons à partir d'un composant binaire et sur un système à l'arrêt. Pour conserver une approche boîte noire en toute circonstance, notre logique d'adaptation est extérieure aux composants, contrairement aux techniques qui encapsulent la logique d'adaptation dans les composants [Göb04]. Par ailleurs, la mise en oeuvre des interactions entre le composant adapté et le reste de l'application est à la charge de l'architecte logiciel.

4.2.1.2 Nos opérations

Dans notre approche, une opération d'adaptation prend en entrée un composant et rend un composant adapté, voici la signature d'une telle opération :

$$\texttt{OpAdaptation : Comp} \rightarrow \texttt{Comp}$$

Il s'agit de regrouper dans un paquetage un ensemble d'opérations d'adaptation structurelles prédéfinies. Ces opérations devront être composables, c'est à dire que différentes opérations pourront être appliquées successivement à un composant noté `C` :

```
C-adapt = OpAdaptation1(OpAdaptation2(OpAdaptation3(C)))
```

Le but est d'implémenter un ensemble minimal d'opérations d'adaptation structurelle : les opérations d'adaptation de base. Ces opérations de base, composées entre elles, fourniront de nouvelles opérations d'adaptation structurelles qui viendront à leur tour enrichir le système. Nous désignons cet ensemble minimal d'opérations sous le terme "noyau d'opérations".

[5] pas d'accès au code source

4.2.2 Les concepts de Fractal

Nous avons vu en section 2.2.1.1 qu'un composant et son contenu peuvent être assimilés à un graphe. Pour réaliser une adaptation structurelle, il est nécessaire de naviguer dans ce graphe à travers les liens structuraux (i.e les arcs) afin d'identifier et de manipuler les éléments qui constituent la structure du composant. Voyons jusqu'à quel point le modèle Fractal permet cette navigation.

4.2.2.1 Les concepts réifiés et non réifiés

Le *composant* est considéré comme entité de première classe dans Fractal. Les *interfaces* le sont également, mais pas les services. En effet, les services sont les méthodes fournies par l'interface (au sens Java) correspondant à l'interface (au sens Fractal) du composant : ils ne sont donc pas manipulables. Quant aux interactions, elles ne sont pas véritablement considérées comme entité de première classe en ce sens qu'elle ne sont pas manipulables. Une interaction est un appel de méthode. La notion de binding que l'on trouve dans Fractal permet simplement de fournir à une interface cliente la référence d'une interface serveur. Enfin, une configuration est considérée comme entité de première classe puisqu'elle est assimilée à un composant composite racine (root).

Concepts réifiés	**Concepts non réifiés**
Configuration	Service
Composant	Interaction
Interface	

TAB. 4.1 – Les concepts dans Fractal

Tous les concepts du paradigme composant ne sont donc pas réifiés et cela influence nos choix techniques.

4.2.2.2 Les mécanismes d'introspection

Le modèle Fractal propose un jeu d'interfaces particulières, les interfaces de contrôle (voir section 4.1.1.1). Elles se distinguent des interfaces métiers par leur aspect non-fonctionnel (voir figure 4.2). Ces interfaces de contrôle sont au nombre de six et, comme n'importe quelles interfaces, offrent des services : certains sont déjà définis (i.e implémentés) par l'API Fractal, d'autres doivent être implémentés par l'architecte logiciel.

Remarque Un composant ne possède pas forcément toutes ces interfaces de contrôle, seulement celles qui ont un intérêt pour lui. Ex : un composant primitif ne contient pas l'interface de contrôle ContentController.

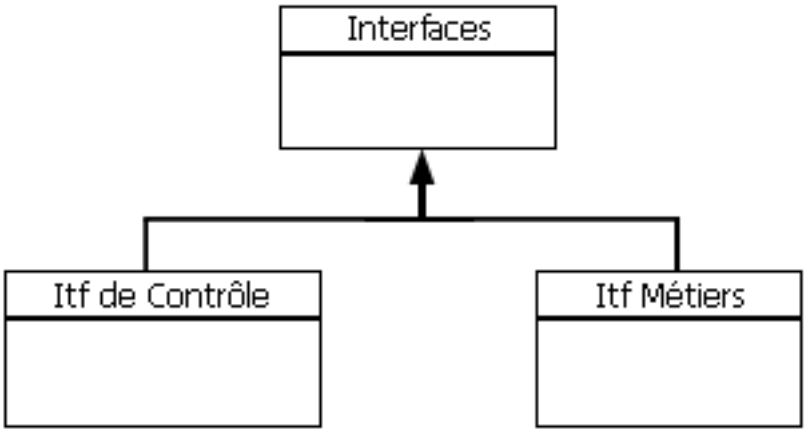

FIG. 4.2 – Les interfaces sous Fractal

L'interface qui nous intéresse plus particulièrement est la ContentController, qui offre un mécanisme d'introspection du contenu du composant. Outre ces interfaces de contrôle, l'API Fractal peut permettre de connaître les interfaces (de contrôle et métier) d'un composant, et inversement, à partir d'un interface de retrouver le composant auquel elle appartient.

On peut donc, à partir d'un composant, accéder à ses interfaces et à ses sous-composants. Le graphe structurel offert par Fractal est suffisant pour pouvoir développer quelques opérations d'adaptation structurelles, mais il n'est pas aussi complet que le graphe structurel théorique présenté en section 3.1.1.

4.2.3 Les opérations de Fractal

Le modèle Fractal n'offre aucune opération d'adaptation (c'est notre travail), mais fournit, comme tout modèle de composant, des opérations d'évolutions de base, ne serait-ce que pour permettre à l'architecte de créer son application.

Opérations proposées
Ajout composant
Suppression composant
Ajout interaction
Suppression interaction
Composition composant

FIG. 4.3 – Opérations d'évolutions proposées par Fractal

Les interfaces sont spécifiées et figées lors de la création d'un composant, il est donc impossible d'en rajouter ou d'en supprimer à un composant existant. Nous proposons d'ailleurs une solution à base de wrapper en section 5.1.3 pour offrir l'opération <Ajout Interface>.

4.2.4 La distribution des composants Fractal

La problématique de distribution de composants Fractal n'est pas traité dans la littérature. Cela est sans doute lié à la jeunesse du modèle Fractal.

La figure 4.4 montre comment un composant Fractal et ses interfaces sont représentés sous leur forme Objet en Java. Le modèle Fractal spécifie que les implémentations des interfaces et des composants doivent être séparés. Nous avons pris l'exemple simple d'un composant *Foo* possédant une interface *Bar*.

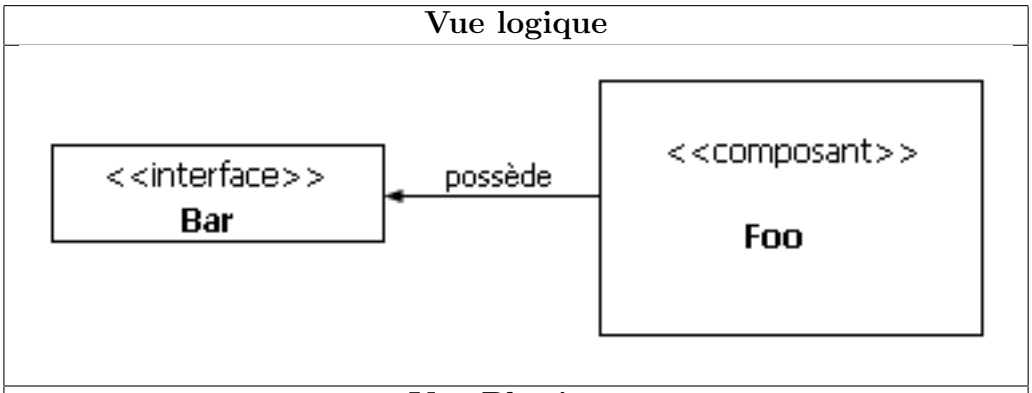

FIG. 4.4 – Correspondance Composant/objet sous Fractal

Dans nos hypothèses, l'architecte reçoit un composant binaire : c'est notre approche boîte noire[6]. De manière générale, s'il existe un marché de composant Fractal, ce sont les objets correspondant au composant et à ses interfaces qui doivent être distribués (dans notre exemple : `foo.class` et `bar.class`) . L'architecte peut alors re-construire un composant Fractal à partir de ces deux "types" d'objets (l'objet composant et les objets interface) et lui appliquer nos opérations d'adaptation structurelle.

Mais après réflexion, il s'avère qu'il manque des informations cruciales à l'architecte pour pouvoir construire un composant Fractal à partir des objets distribués.

- Comment savoir quel est l'objet qui code le composant et les objets qui codent les interfaces ?
- La création des interfaces sous Fractal nécessite de connaître certaines propriétés comme par exemple le rôle de l'interface : client ou serveur ?

[6]par opposition à l'approche boîte blanche qui nécessiterai la modification statique de code Java. Néanmoins il existe des outils pour remanier du code source Java, et en particulier un analyseur de code nommé Barat (`http ://homepages.hype.de/boris/barat/`)

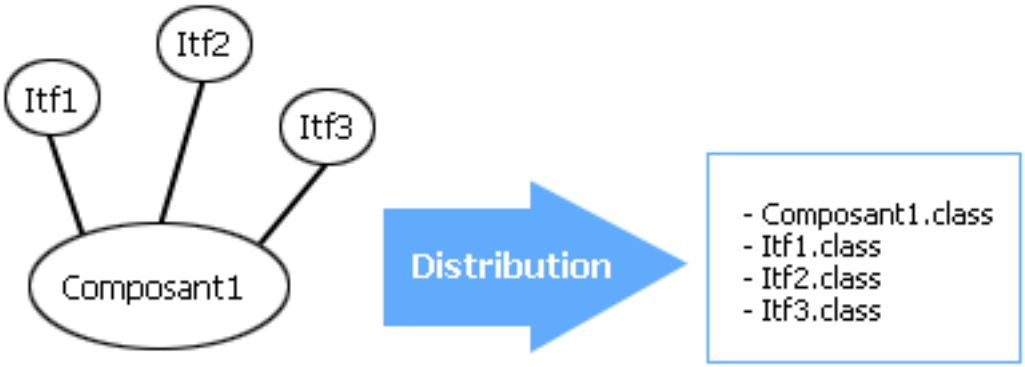

FIG. 4.5 – Distribution des composants Fractal

Autre problème, le concept de composant composite existe au niveau logique, c'est à dire au niveau de la description architecturale, mais n'existe pas au niveau physique (excepté ses interfaces). Cela s'explique par le fait qu'un composant composite n'a pas d'implémentation[7] à la différence d'un composant primitif. Cela signifie qu'il n'est pas possible de distribuer des composants composites sous leur forme binaire.

Il serait possible de résoudre certains de ces problèmes en accompagnant les objets d'une documentation ou d'un protocole pour guider l'architecte dans la re-construction du composant Fractal à partir des objets distribués, dans l'idée des composants auto-documentés [MEY92]. On peut également penser à l'outil d'introspection offert par l'environnement d'exécution Java : la commande «javap ». En effet, les objets java étant pseudo-compilés, ils contiennent des méta-données et s'auto-décrivent. Cette commande, appliquée à un objet, affiche toutes les caractéristiques de la classe de l'objet (signature des méthodes, classes héritées, interfaces implémentées, etc.). Mais pour l'heure, aucune stratégie n'est proposée...

4.3 Bilan

4.3.1 Évaluation du modèle Fractal

Fort des constats précédents, nous pouvons lister, les qualités et les défauts du modèle de composant Fractal, par rapport à notre objectif :

⊕ Modèle de composant hiérarchique qui permet l'utilisation de wrappers (Cf. technique d'adaptation en section 2.4.2.4).

⊕ Offre un mécanisme d'introspection par le biais des interfaces de contrôle (Navigation).

⊖ Ne réifie pas tous les concepts du paradigme composant. (Uniquement la *configuration* le *composant* et l'*interface*)

[7] donc pas de classe Java qui lui est propre

⊖ Sa conception objet sous-jacente. (Le paradigme composant doit combler les défauts du paradigme objet, mais de nombreux modèles de composants ont une conception objet sous-jacente, ce qui ne fait que reporter le problème.)

⊖ Pas de stratégie de distribution proposée.

4.3.2 Notre objectif

Puisque le modèle Fractal ne réifie pas l'ensemble des concepts du paradigme composant, notre tableau d'opération structurelle se restreint aux seules opérations d'adaptation structurelles qui pourront être supportées :

	Composant	**Interface**	**Service**	**Interaction**
Ajout		•	–	–
Suppression		•	–	–
Modification	–	–	–	–
Substitution	–	–	–	–
Composition	–	–	–	–
Décomposition	•		–	–
Transfert			–	–
Masquage		•	–	–

TAB. 4.2 – Opérations d'adaptation structurelle sous Fractal

Légende :

•	Supporté
	Non supporté
–	Indéfini

Au vu de nos hypothèses initiales et des limitations imposées par le modèle de composant Fractal, notre objectif est de réaliser nos opérations d'adaptation structurelles :

1. sur des composants Fractal bien définis (on ne s'occupe pas de leur construction à partir d'objets Java distribués – Cf. section 4.2.4)
2. sans accès au code source pour conserver notre approche boîte noire. De plus, si la problématique de distribution des composants Fractal sous forme binaire est résolue, nos opérations seront toujours valides

Chapitre 5

IMPLÉMENTATION EN JAVA

5.1 Un noyau d'opérations

Nous présentons ici la façon dont nous avons procédé pour réaliser les trois opérations d'adaptation que nous avons pu dégager (Section 4.3, tableau 4.2). Nous utilisons la technique du wrapping (Cf. Section 2.4.2.4) pour sa facilité et son approche résolument boîte noire.
Nos opérations ne s'occupent que des interactions internes aux composants adaptés alors que notre IHM (voir section suivante) s'occupe aussi – dans la mesure du possible – des interactions entre les composants fraîchement adaptés et le reste de l'application. Ces opérations ne fonctionnent qu'à l'arrêt de l'application (approche statique). L'interface de contrôle *LifeCycleController* permet justement d'arrêter un composant puis de le redémarrer.

5.1.1 Positionnement tridimensionnel

Nous proposons une approche d'adaptation : elle peut donc être positionnée dans notre repère tridimensionnel présenté en section 2.2.2.

Positionnement tridimensionnel

1. QUOI : Composant, Interface
2. QUAND : statique
3. COMMENT : manuel

5.1.2 Masquage d'une interface

La véritable suppression d'une interface (opération présente dans le tableau) nécessite l'accès au code. Notre approche boite noire nous interdit cette opération. Comme alternative, nous proposons l'opération de masquage d'interface également présente dans le tableau.

Objectif : Le composant d'origine propose les interfaces I1 et I2. On souhaite masquer I2.

Procédure : On enveloppe (wrapping) le composant d'origine dans un composant composite n'offrant que l'interface I1. Les services de I1 du wrapper sont simplement délégués aux services de I1 du composant d'origine.

Signature Julia :

```
public static Component Masquage(Component C, String interfaceName)
```

Notre opération d'adaptation	Opérations d'évolution utilisées
Masquage interface	Ajout composant
	Ajout binding

TAB. 5.1 – Masquage d'une interface : lien de description

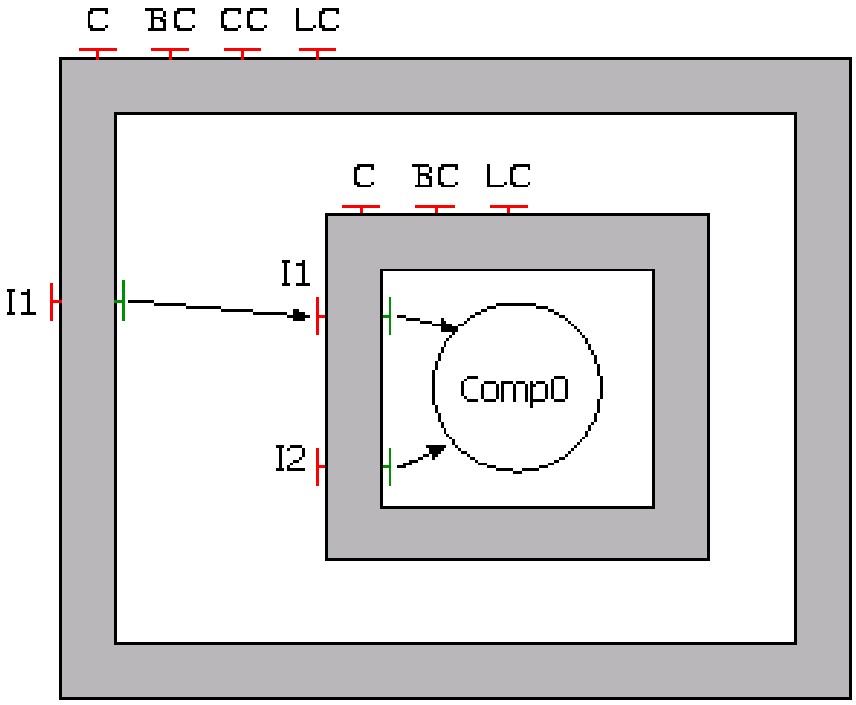

FIG. 5.1 – Masquage d'interface : le composant adapté

5.1.3 Ajout d'une interface

Objectif : Le composant d'origine propose l'interface I1. On souhaite lui ajouter une nouvelle interface I2.

Procédure : Les nouveaux services de l'interface doivent être implémentés. Cette adaptation nécessite donc que l'utilisateur déclare et implémente les nouveaux services de l'interface sous forme de deux classes Java (une pour la nouvelle interface et une pour le nouveau composant). Ce nouveau composant va fournir ces services à travers l'interface I2, lié à cette implémentation. Le nouveau composant et le composant d'origine vont être enveloppé dans un composant composite offrant les deux interfaces I1 et I2.

Signature Julia :

```
public static Component AjoutInterface(Component C, String implementation,
InterfaceType IftType)
```

Notre opération d'adaptation	Opérations d'évolution utilisées
Ajout interface	Ajout composant
	Ajout binding

TAB. 5.2 – Ajout d'une interface : lien de description

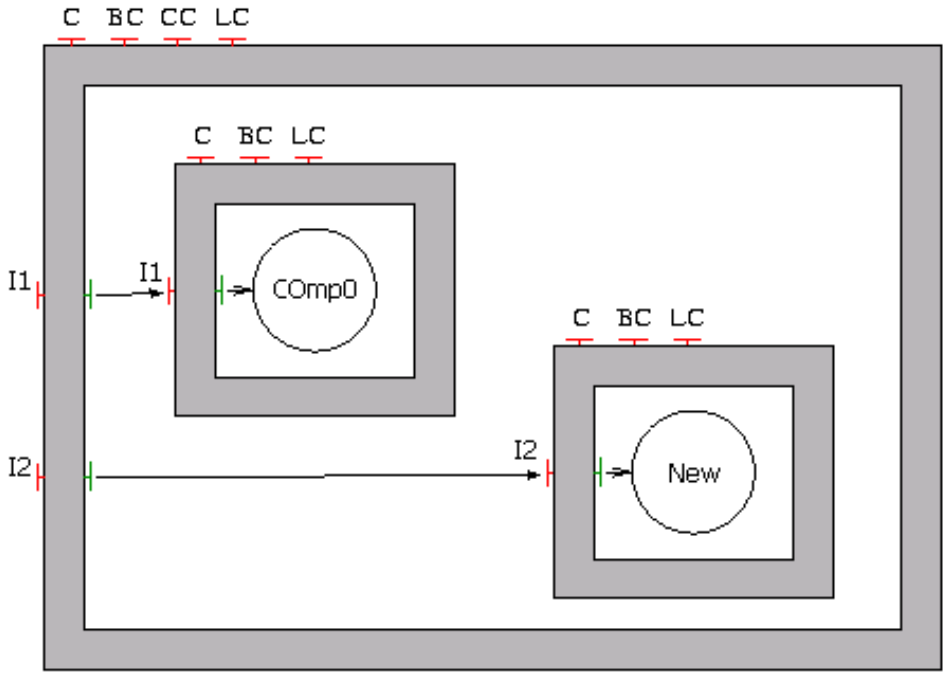

FIG. 5.2 – Ajout d'une d'interface : le composant adapté

5.1.4 Décomposition d'un composant

Objectif : Décomposer un composant composite en faisant ressortir ses sous-composants. (Cette décomposition d'un composant composite ne doit pas être confondue avec la décomposition structurelle d'un composant monolithique présentée en section 3.2)

Procédure : On supprime l'enveloppe autour des éléments constitutifs et les liaisons uniquement de l'enveloppe aux sous-composants. Les liaisons entre les composants internes sont conservées.

Signature Julia :

```
public static Component[] Decomposition(Component C)
```

Notre opération d'adaptation	Opérations d'évolution utilisées
Décomposition	Suppression binding

TAB. 5.3 – Décomposition : lien de description

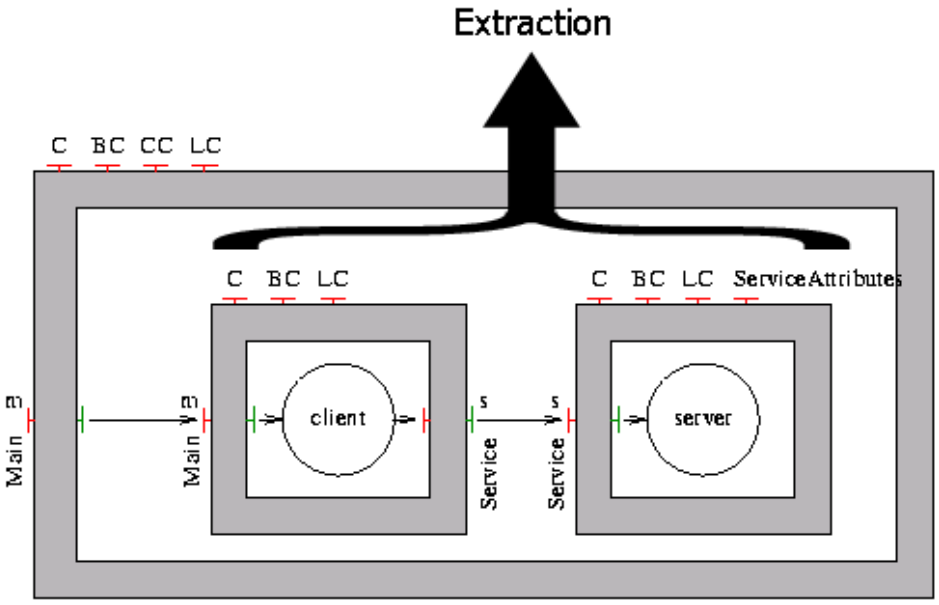

FIG. 5.3 – Décomposition : extraction des sous-composants

5.2 Notre Contribution : UMF

UMF (*Update Manager for Fractal*[1]) est une modeste "extension" du modèle Fractal, proposant des opérations d'adaptation reposant majoritairement sur les opérations d'évolution offertes initialement par Fractal. Ces opérations de mises à jour sont appliquée de façon ergonomique et simplifiée par le biais d'une interface homme-machine.

[1]Gestionnaire de mise à jour pour Fractal

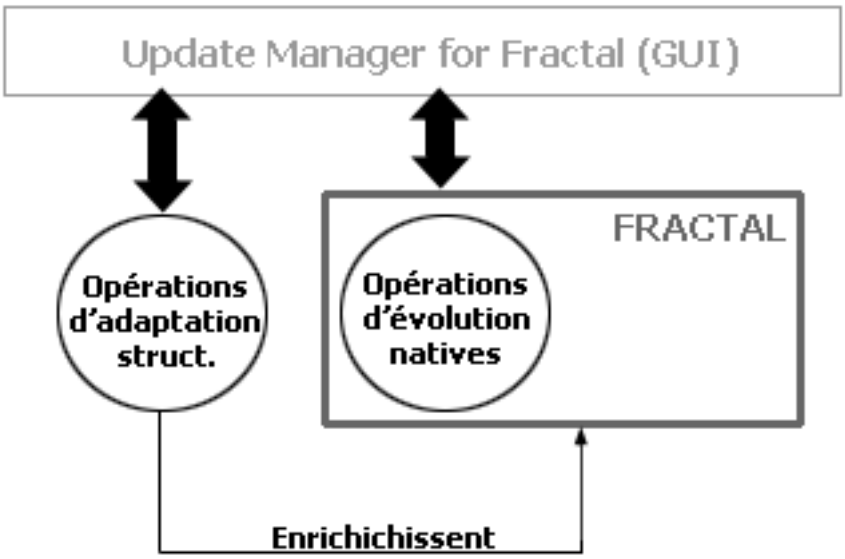

FIG. 5.4 – Update Manager for Fractal

Nos opérations d'adaptation structurelles sont regroupées dans un paquetage, l'UMF dans un autre. En effet, un architecte peut vouloir utiliser nos opérations d'adaptation directement dans son code, sans passer par l'interface UMF. Le paquetage `adaptation` contient nos opérations et peut être assimilé à une "boite à outil" pour adapter des composants logiciels.

Le lancement d'UMF s'effectue par l'instanciation d'un objet *Explorer*. Son constructeur requière le composant Fractal de plus haut niveau hiérarchique : le composant racine. Pour utiliser l'UMF, l'architecte n'a donc que cette simple ligne a ajouter dans son code :

```
Explorer e = new Explorer(rootComponent) ;
```

5.2.1 Interface Homme-Machine

UMF offre une couche graphique afin de faciliter la création, la gestion et et la maintenance d'une architecture Fractal, que ce soit pour les opérations d'évolution de bases ou pour nos nouvelles opérations d'adaptation.

Fractal étant un modèle hiérarchique de composants, il nous a semblé naturel de représenter les éléments de l'architecture par un arbre (javax.swing.JTree) où les composants sont *les noeuds* et les interfaces sont *les feuilles*. Nous avons abordé cette représentation (graphe structurel) en section 3.1.1. Les opérations de mise à jour sont appliquées aux noeuds ou aux feuilles via les boutons situés sur la droite et les formulaires de saisies adéquats.

Lorsqu'un composant doit être adapté et que celui-ci est lié au reste de l'application (par les bindings), UMF prend en charge la deconnexion du composant. Cet isolement vis à vis du reste de l'application est une condition nécessaire au bon déroulement de l'adaptation. Puis, le composant adapté est reconnecté au reste de l'application, quand cela est possible.

Des captures d'écrans d'UMF sont présentées en Annexes de ce mémoire.

5.2.2 Intégration à l'IDE Eclipse

Nous avons utilisé l'IDE Eclipse[2] pour réaliser UMF. Les fichiers nécessaires au fonctionnement d'une application Fractal sont disponibles sous forme d'une archive *Jar* à inclure en tant qu'archive externe dans l'environnement de travail Eclipse (le *workspace*).

Remarque En considérant Eclipse comme espace de travail exclusif pour développer sous Fractal, il pourrait être intéressant de convertir UMF en un plug-in pour Eclipse, qui serait proposé sur les marchés de plug-ins, comme par exemple sur `www.eclipseplugin-central.com`.

5.3 Lien de conséquence

Le déclenchement et la réalisation automatique d'une opération de mise à jour suite à une adaptation structurelle n'est pas de notre ressort pour cette étude. Cela constitue d'autres travaux sur le respect des contraintes dans les architectures logicielles et sur les mises à jour dynamiques et autonomes, etc. Notons, encore une fois, que différents thèmes de recherche se rassemblent autour d'un seul thème plus global : la mise à jour dans les architectures à composants. Pour notre modeste validation pratique, les mises à jour seront déclenchées et réalisées manuellement (Cf Section 2.2.1.3).

5.4 Ce qu'il reste à faire

- Les saisies via l'IHM ne sont pas toutes contrôlées, on considère que l'architecte a une connaissance minimum du modèle Fractal.
- Il subsiste quelques problèmes de reconnexion automatique entre un composant nouvellement adapté et le reste de l'application.

[2]http ://www.eclipse.org

Conclusion et perspectives

Cette étude se veut être une réflexion sur la mise à jour dans les architectures à base de composants. C'est une remise en cause permanente du paradigme composant et des modèles de composants étudiés. C'est cet esprit critique qui a guidé le travail d'initiation à la recherche de ce mémoire. La réutilisation est la pierre angulaire du paradigme composant mais nous sommes convaincus que cette réutilisation n'est optimale que si la logique de mise à jour est prise en compte.

Nous avons proposé dans cette étude, de lever l'ambiguïté qui existe entre l'évolution et l'adaptation, qui sont les deux techniques majeures de la mise à jour des composants logiciels. Nous avons proposé une caractérisation de la mise à jour divisée en trois dimensions appelées : Quoi (l'objet), Quand (le moment), Comment (le support). Nous avons utilisé cette caractérisation à la fois pour des techniques et des problématiques de mises à jour. Puis, l'étude des principales techniques d'adaptation et d'évolution a permis d'établir une taxonomie des opérations de mise à jour. La caractérisation et la classification sont les deux piliers de notre étude globale.

En montrant dans un deuxième temps que l'évolution et l'adaptation ne se sont pas deux concepts distincts mais qu'il existe au contraire des relations que nous avons appelés lien de description et lien de conséquence, nous préconisons une unification plus forte du processus de recherche autour de cette problématique plus globale et aux enjeux considérables : la mise à jour. Des sujets de recherche cloisonnés n'aboutiront pas à un solution complète.

Notre démarche globale permet de prendre conscience que les modèles de composants actuels qui offrent des opérations d'évolutions de bases, offrent potentiellement des opérations d'adaptation, et ce, au moindre effort. Elle permet aussi à l'architecte logiciel d'y voir plus clair. Notre application UMF illustre bien ce constat : on peut aisément ajouter des opérations d'adaptation au modèle de composant Fractal, par réutilisation des opérations d'évolution existantes.

Par ailleurs, les perspectives sont nombreuses. Ce sont les mises à jour statiques et manuelles qui sont les plus nombreuses, la mise à jour dynamique et automatique reste donc un enjeu considérable. Nous avons étudié les relations adaptation[3] vers évolution, mais l'étude de la relation inverse est tout à fait possible. Concernant notre validation pratique avec UMF, idéalement, le déclenchement et la réalisation des mises à jour suite à une adaptation structurelle se fera de manière totalement automatisé. C'est le but ultime : une architecture consciente d'elle même et des contraintes qui la régissent, complètement autonome, auto-adaptative et auto-évolutive. Cet objectif met en jeu beaucoup de compétences et de thématiques de recherche, mais c'est ce à quoi nous aspirons pour demain...

[3]structurelle dans notre cas précis

.1 Annexe : Captures d'écrans d'UMF

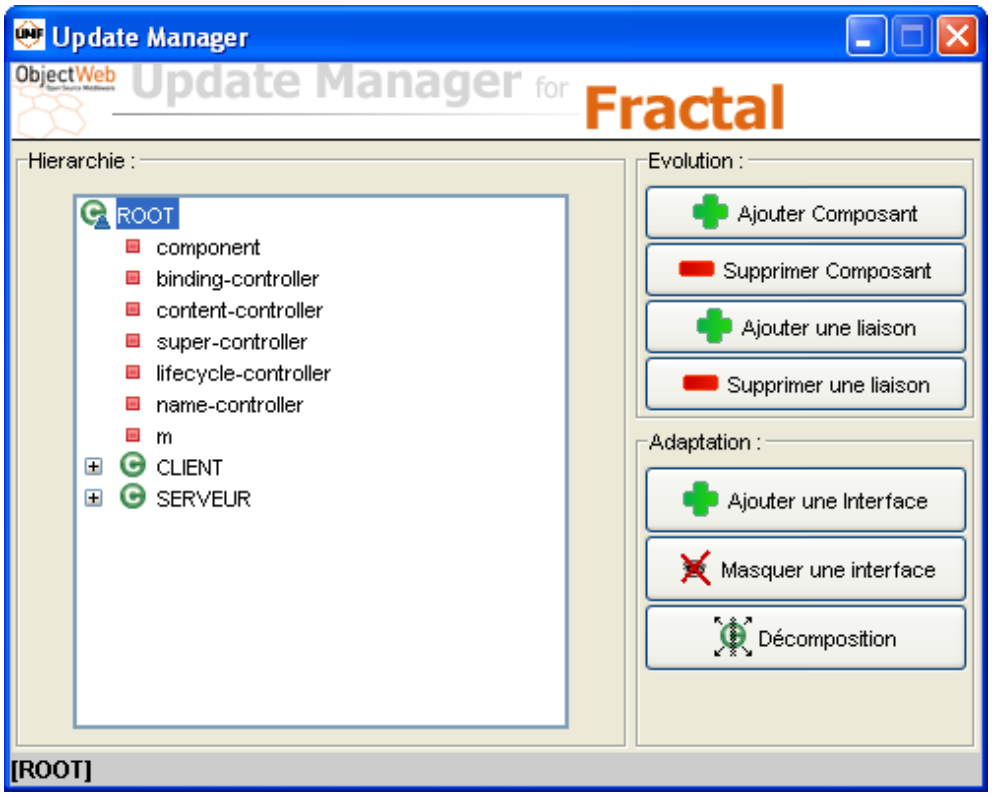

FIG. 5 – UMF : Le gestionnaire

FIG. 6 – UMF : Le détail des interfaces d'un composant

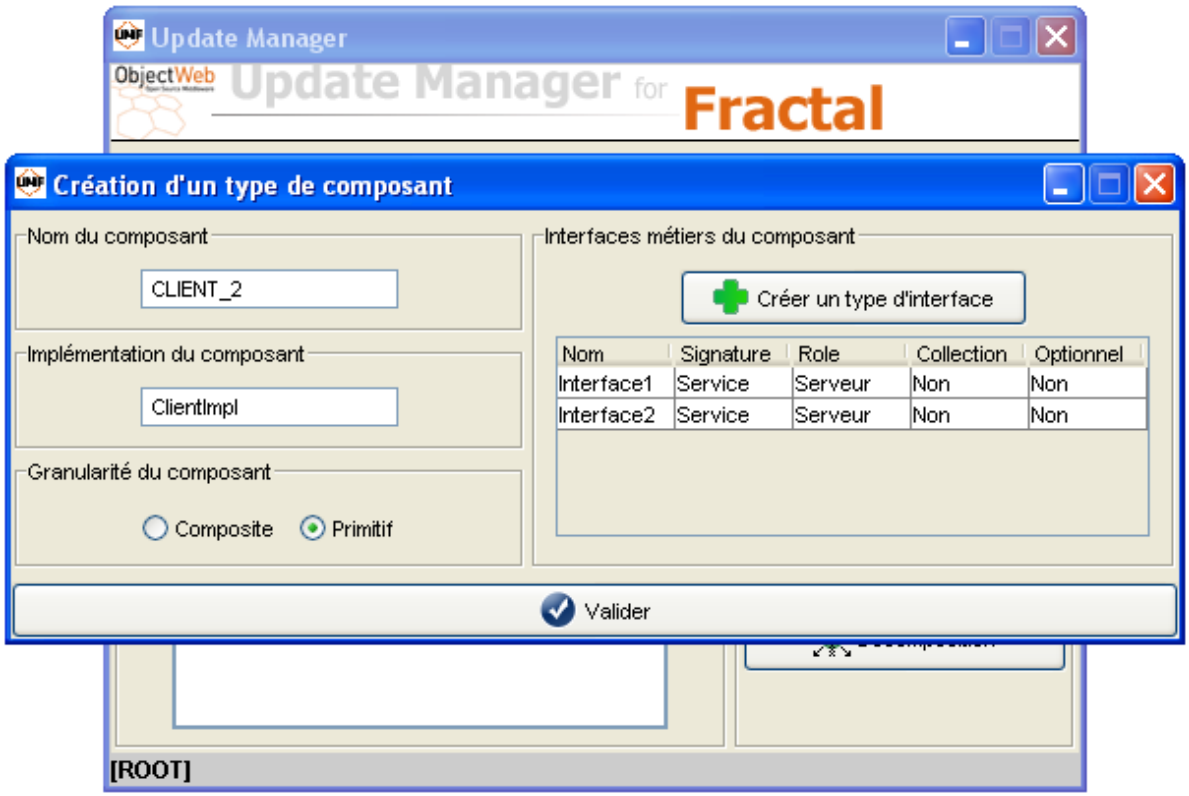

FIG. 7 – UMF : Exemple d'opération d'évolution (Ajouter un nouveau composant)

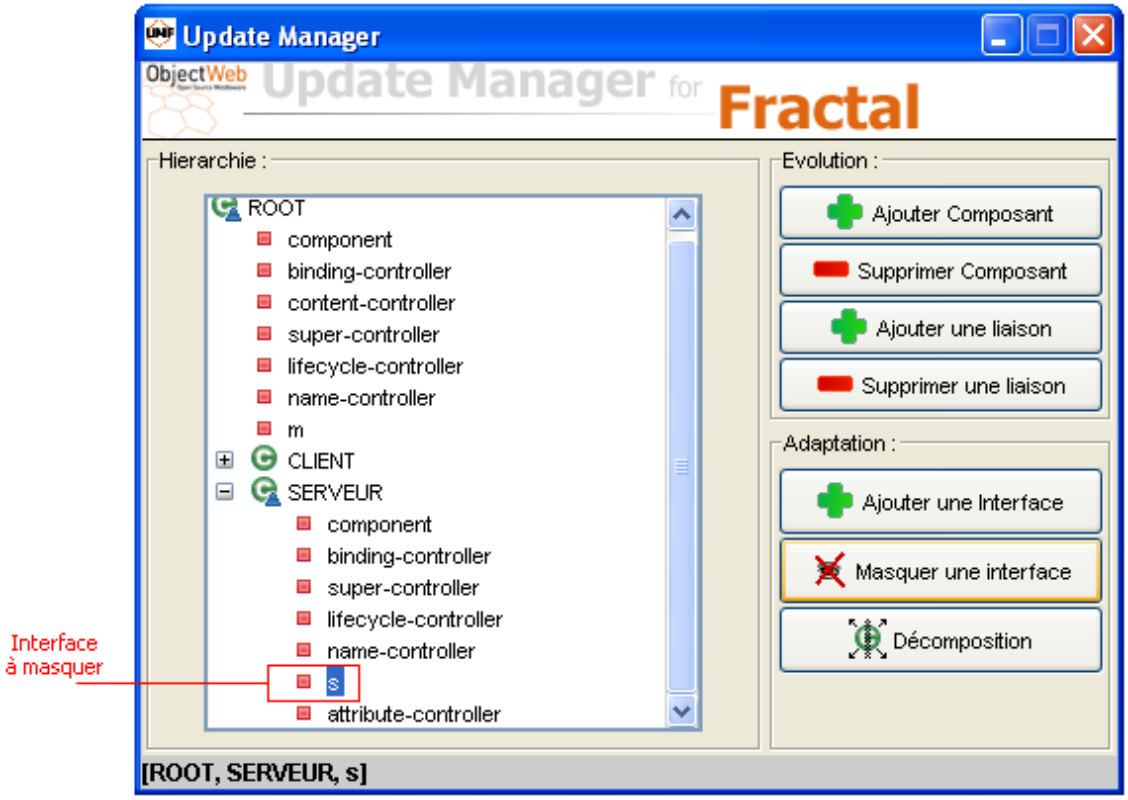

FIG. 8 – UMF : Exemple d'opération d'adaptation (Masquer une interface 1/2)

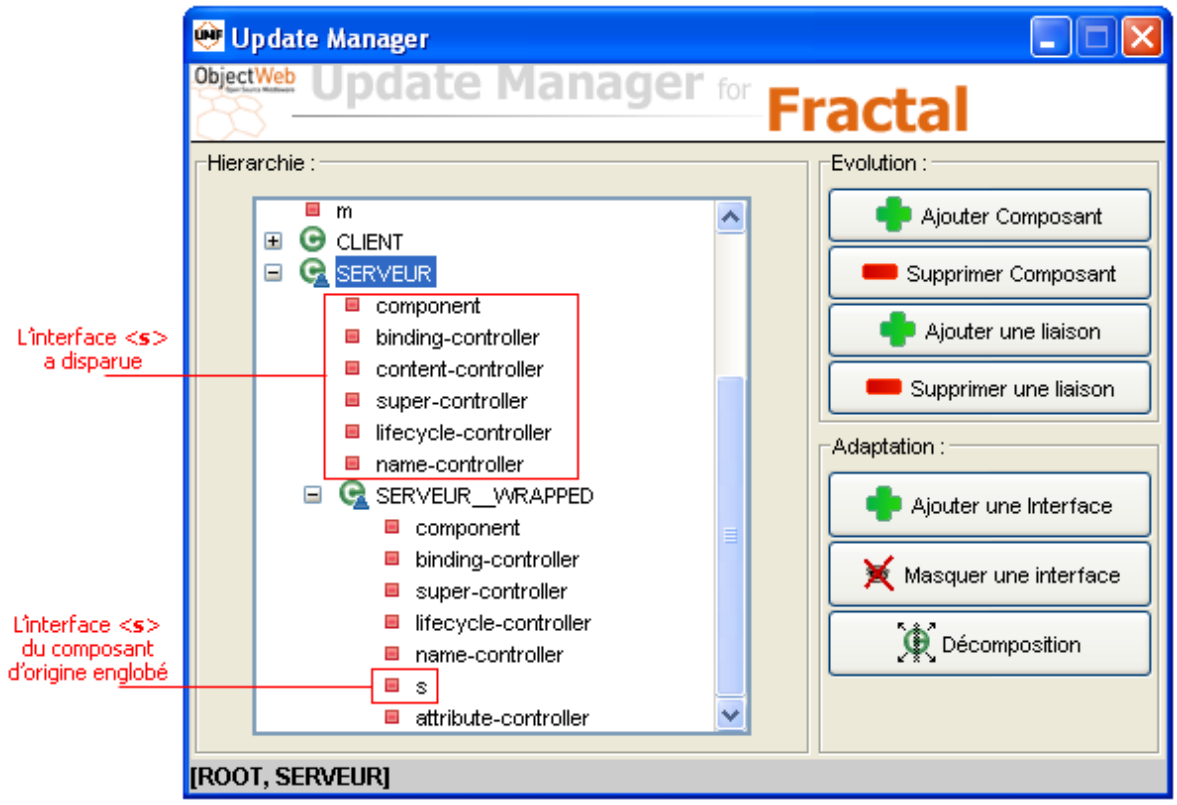

FIG. 9 – UMF : Exemple d'opération d'adaptation (Masquer une interface 2/2)

Références

[BCS04] E. Bruneton, T. Coupaye, and J.B Stefani. *The Fractal Composition Framework, Interface specification 1.0.* France Telecom R&D ed., 2004.

[BMBvZ04] Kathrin Berg, John Müller, Judith Bishop, and Jay van Zyl. The use of feature modelling in component evolution. *Technical Report. University of Pretoria*, May 2004.

[Bos97] Jan Bosch. Superimposition : A component adaptation technique. *Dept. of Computer Science and Business Administration/Blekinge Institute of Technology*, 1997.

[Bru04] E. Bruneton. *The Julia Tutorial.* France Telecom R&D ed., 2004.

[BSO05] G. BASTIDE, A. SERIAI, and M. OUSSALAH. Commode : Un modèle d'adaptation structurelle pour les composants logiciels. *OCM 2004, Berne*, 9-11 mars 2005.

[CC04] H. Chang and Philippe Colet. Vers la négociation de contrats dans les composants logiciels hiérarchiques. Technical report, Projet OCL, Rapport de Recherche, ISRN I3S/RR-2004-33-FR, Octobre 2004.

[DL03] P.C. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003, volume 2893 of LNCS, pages 1-14, Springer-Verlag, Paris*, November 2003.

[DL04] Pierre-Charles David and Thomas Ledoux. Pour un aspect d'adaptation dans le développement d'applications à base de composants. *In Actes Journée de l'AS 150, Systèmes répartis et réseaux adaptatifs au contexte, Paris, France*, April 2004.

[Gau04] BASTIDE Gautier. *Adaptation des composants logiciels appliquée aux environnements ubiquitaires et mobiles.* PhD thesis, Ecole des Mines de Douai, 2004.

[Göb04] Steffen Göbel. Encapsulation of structural adaptation by composite components. *WOSS'04 Newport Beach, CA10/31*, 11 january 2004.

[GKL04] Paul Grünbacher, Johannes Kepler, and Yves Ledru. Automated software enineering. *ERCIM News No.58*, July 2004.

[GMW68] D. Garlan, R. Monroe, and D. Wile. *ACME : Architectural Description Of Components-based systems.* Leavens Gary and Sitaraman Murali, Foun-

dations of component-based systems, Cambridge University Press, 2002, pp. 47-68.

[JR01] M. Sutton Jr. and I. Rouvellou. Advanced separation of concerns for component evolution. *Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) (OOPSLA)*, October 2001.

[KBC02] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Automatic adaptation of component-based software. *ICSSEA*, 2002.

[Kea97] G. Kiczales and John Lamping et al. Aspect oriented programming. *In Proceedings of ECOOP'97*, 1997.

[KH98a] Ralph Keller and Urs Holzle. Binary component adaptation. *Lecture Notes in Computer Science, Vol. 1445, p. 307- ? ?*, 1998.

[KH98b] Ralph Keller and Urs Holzle. Implementing binary component adaptation for java. *Technical Report, University of California, Santa Barbara, Computer Science*, August 6 1998.

[Kic96] Gregor Kiczales. Beyond the black box : open implementation. *IEEE Software, vol. 13, no 1, pp. 8-11*, janvier 1996.

[KLM97] Gregor Kiczales, John Lamping, and Gail Murphy. Open implementation design guidelines. *in 19th International Conference on Software Engineering*, May 1997.

[KSO04] T. Khammaci, A. Smeda, and M. Oussalah. A Multi-Paradigm Approach to Describe Software Systems. In *Proceedings of the 3rd WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'04)*, Salzburg, Autriche, 2004.

[MEY92] MEYER. *Applying Design by Contract*. IEEE Computer, p. 40-51, 1992.

[ML85] LEHMAN M. and BELADY L. Program evolution : Process of software change. *London : Academic Press*, 1985.

[MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description langages. *IEEE Transactions on Software Engineering, Vol. 26*, 2000.

[OBFDR02] Audrey Occello, Mireille Blay-Fornarino, Anne-Marie Dery, and Michel Riveill. Vers une adaptation dynamique cohérente des composants. *In Journée composants : Systèmes à composants adaptables et extensibles, Grenoble, France*, 17 et 18 octobre 2002.

[OMG97] OMG. Model driven architecture, 1997. http ://www.omg.org/mda/.

[OTS05] Mourad Oussalah, Dalila Tamzalit, and Nassima Sadou. Software architecture evolution : Description and management. *In proceding of the 2005 International Conference on Software Engineering Research and Practice (SERP'05) Las Vegas, Nevada*, 27-30 June, 2005.

[Ous02] Mourad Oussalah. Towards a user-customizable graph class evolution model. *In The 2002 International Conference on Artificial Intelligence (IC-AI'02), Las Vegas, NV, USA*, June 2002.

[RvdHMRM04] Roshanak Roshandel, André van der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae : A system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology, 13(2), pp. 240-276*, April 2004.

[SOK03] Smeda, M. Oussalah, and T. Khamaci. A multi-paradigm approach to describe complex software system. *WSEAS Transactions on Computers, Issue 4, Volume 3, pp. 936-941*, October 2003.

[Szy96] Clemens Szyperski. Independently extensible system - software engineering potential and challenges. *In proceeding of the 19th Australian Computer Science Conference, Melbourne, Australia*, 1996.

[Szy98] Szyperski. *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, 1998.

[Szy04] Clemens Szyperski. Transat : Maîtriser l'evolution d'une architecture logicielle. *Conférence LMO'04*, 2004.

[Tam00] Dalila Tamzalit. *GENOME :un Modèle pour la Simulation d'Émergence de Structures d'Objets.* PhD thesis, Université de Nantes – UFR Sciences et Techniques, 2000.

[TFS04] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Préservation de choix architecturaux lors de l'evolution d'un composant. *atelier OCM-SI (Objets Composants et Modèles dans l'ingénierie des Systèmes d'Information), INFORSID'04, Biarritz, France*, 25 may 2004.

[WSSJ98] Welch, I. S., Stroud, and R. J. Using metaobject protocols to adapt third-party components. *Work in Progress Paper at Middleware'98*, 1998.

Résumé

Une importante problématique s'impose de plus en plus : celle d'assurer le cycle de vie des composants, principalement en assurant leur mise à jour. Cette mise à jour se décline en deux techniques majeures : l'évolution et l'adaptation. Nous proposons de lever l'ambiguïté entre ces deux notions par un état de l'art sur les techniques d'évolution et les techniques d'adaptation. Nous cherchons ensuite à mettre en évidence les relations entre l'évolution et l'adaptation. Nous souhaitons ainsi définir un cadre générique d'étude de la mise à jour dans les architectures à base de composants.

Mots-clés : Composants logiciels, Évolution, Adaptation

Abstract

A significant problematic is more and more essential : to ensure the cycle of life of the components, mainly by ensuring their update. This update is declined in two major techniques : the evolution and the adaptation. We propose to raise ambiguity between these two concepts by a state of the art on the techniques of evolution and the techniques of adaptation. Then, we seek to highlight the relations between the evolution and the adaptation. We wish to define a generic framework of study of the update in components based architectures.

Keywords : Software components, Evolution, Adaptation