PROJET RNTL ARCAD*
D1.1 - État de l'art sur l'adaptabilité

T0+6 : interne - T0+12 : public

Coordonnateur : Thomas LEDOUX
Ecole des Mines de Nantes
4, rue Alfred Kastler
44307 Nantes Cedex 3

avec la participation de M. Blay, E. Bruneton, D. Caromel
T. Coupaye, D. Hagimont, J-M. Menaud, J. Noyé et M. Riveill

12 décembre 2001

Résumé

Ce document a pour objectif d'étudier l'adaptabilité dans les systèmes logiciels. Dans un premier temps, nous présentons les principes généraux de l'adaptabilité. Ensuite, nous proposons une synthèse des différentes approches technologiques permettant de supporter l'adaptabilité. Puis, nous passons en revue les différentes propositions prenant en compte l'adaptabilité dans le domaine des intergiciels. En guise de conclusion, nous exposons une synthèse qui compare les travaux des différents partenaires concernant l'adaptabilité.

1 Introduction

1.1 Besoins d'adaptabilité

Les nombreux progrès réalisés dans les domaines des systèmes d'informations grande échelle et de l'informatique mobile ont introduit de nouvelles problématiques et créé de nouveaux besoins en terme d'adaptabilité pour la construction de systèmes logiciels.

---

*Le projet ARCAD (Architecture Extensible pour Composants Adaptables) a été labellisé en décembre 2000. Les différents partenaires sont France Télécom R&D (équipe ASR - Thierry.Coupaye@francetelecom.com), INRIA (projet Oasis - Denis.Caromel@inria.fr, projet Sardes - Daniel.Hagimont@inria.fr), Ecole des Mines de Nantes (equipe OCM - Thomas.Ledoux@emn.fr), le laboratoire I3S commun à l'Université de Nice - Sophia Antipolis et au CNRS (projet Rainbow - Anne-Marie.Pinna@unice.fr). Le coordonateur du projet est Michel.Riveill@unice.fr

Ces environnements présentent, en effet, une hétérogénéité importante, une grande variabilité et de nombreuses possibilités d'évolution, aussi bien au niveau des moyens d'exécution que des moyens de communication. Ainsi, les ressources offertes peuvent être extrêmement disparates selon que l'on utilise un assistant personnel, un ordinateur portable, une station de travail ou un serveur d'exécution dédié. Les réseaux de communication présentent des infrastructures et des performances bien différentes comme les réseaux filaires ou les réseaux sans-fil. De plus, ces éléments sont soumis à d'importantes variations au cours du temps. La disponibilité des ressources n'est pas figée et peut varier en fonction d'ajout ou de suppression à la volée de périphériques, ou de leur atteignabilité dans le cas de l'utilisation d'une ressource distante. Les performances du réseau peuvent varier en fonction de l'environnement distant (la qualité de service de bout en bout) ou de l'environnement proche du terminal (déplacements de l'utilisateur, interférences avec des éléments physiques ou changements de cellules). Ces environnements bénéficient enfin d'une grande dynamique d'évolution que l'on peut aisément constater avec l'apparition des services Web [Web], des *smart phones* [CeB] et des réseaux sans-fil de troisième génération [IMT].

Les caractéristiques de ces environnements nous obligent à reconsidérer le développement et l'exécution d'applications dans un contexte d'un genre nouveau. En effet, dans les environnements traditionnels, le développement et l'exécution d'application s'effectuent en supposant que le support d'exécution est connu à l'avance. Lorsqu'une application alloue une ressource, une non-disponibilité de celle-ci en cours d'exécution conduit à une terminaison non prévue de l'application. Dans les environnements que nous considérons, du fait de l'hétérogénéité, de la variabilité et de l'évolutivité, de nombreux changements peuvent intervenir et ceux-ci ne doivent pas être considérés comme des erreurs fatales, mais doivent au contraire être pris en compte pour justement adapter l'application. Il s'avère donc indispensable de répondre à la problématique de l'adaptabilité pour la construction de systèmes logiciels.

1.2 Scénario d'adaptabilité

Pour illustrer ces nouveaux besoins en adaptabilité, nous allons étudier l'utilisation d'une application de flux vidéo sur un assistant personnel avec une connexion sans-fil.

À la conception de cette application, le programmeur décide de prévoir et d'implanter deux adaptations possibles dans le cas où le service ne peut être rendu de manière optimale. La première consiste à diminuer le nombre d'images transmises par seconde, et la seconde consiste à dégrader la qualité de l'arrière plan de la vidéo.

Ensuite, tout en se déplaçant, notre utilisateur exécute son application et plusieurs adaptations sont alors envisageables. Lors d'une diminution de la bande passante due aux interférences de la connexion sans-fil, une possibilité

d'adaptation est d'adopter le mode prévu de dégradation de l'arrière plan de la vidéo. Lors d'une congestion entre le serveur vidéo et l'assistant, une autre possibilité est de diminuer le nombre d'images transmises par seconde. Des adaptations non prévues peuvent également avoir un sens. La localisation de l'utilisateur peut être utilisée pour choisir le serveur vidéo le plus proche.

Enfin, l'application peut également évoluer en adaptant ses fonctionnalités ou en intégrant de nouvelles. Si le client utilise un décodeur MPEG-2, lors de la connexion à un serveur vidéo utilisant un encodage MPEG-4, celui-ci doit pouvoir charger et installer le décodeur correspondant. Cette intégration de nouvelles fonctionnalités peut se faire à l'initiative de l'utilisateur, par exemple si celui-ci veut intégrer un module effectuant des ralentis et des retours en arrière, ou à l'initiative d'un tiers, par exemple si le fournisseur de flux vidéo souhaite ajouter un module de télépaiement.

Ce scénario montre qu'il existe plusieurs types d'adaptations, que celles-ci peuvent intervenir à différents moments et à l'initiative de différents acteurs.

1.3 Organisation du document

Ce document se propose de faire un état de l'art et une synthèse sur l'adaptabilité dans les systèmes logiciels. Il est organisé en trois parties. Dans une première partie, nous dégageons un certain nombre de principes et de critères généraux autour de l'adaptabilité. Dans une deuxième partie, nous proposons une synthèse des différentes technologies actuelles offrant un support à l'adaptabilité. Enfin, dans une troisième partie, nous considérons la place de l'adaptabilité dans le domaine des intergiciels. Une synthèse axée autour des différents travaux des équipes du projet et s'appuyant sur les critères proposés conclut ce document.

2 Principes généraux

Le problème de l'adaptabilité dans les systèmes logiciels est très vaste, mais il en émerge un certain nombre de questions récurrentes auxquelles chaque solution doit répondre. Le but de cette section est d'identifier ces questions et d'apporter pour chacune d'elles un certain nombre de réponses.

2.1 Introduction

2.1.1 Préambule

« adapter v. : rendre (un dispositif, des mesures, etc.) apte à assurer ses fonctions dans des conditions particulières ou nouvelles » [Hac]

Qu'entend-on par *adaptation* dans les systèmes logiciels ? En reprenant la définition sémantique du terme, une adaptation correspond au processus de modification du système, nécessaire pour permettre un fonctionnement

adéquat dans un contexte donné. Le terme adéquat signifie que le système correspond parfaitement à ce que l'on attend dans ce contexte précis. Le processus de modification du système peut s'opérer de différentes manières que nous allons détailler par la suite[1].

2.1.2 Propriétés pour l'adaptabilité

« adaptabilité n. f. : qualité de ce qui peut être adapté, de ce qui peut s'adapter. » [Hac]
Cette notion fait référence à la potentialité, aux qualités inhérentes d'un système. La capacité à supporter l'adaptabilité sous-tend donc un certain nombre de propriétés.

Modularité et composabilité. L'adaptabilité suppose implicitement la capacité d'un système logiciel à être modulaire et composable. Un système monolithique ne peut être modifié qu'en le remplaçant intégralement ; dans un système construit sous forme de modules interconnectés, il est possible de modifier ou de remplacer certaines parties du système avec un minimum d'interférences avec le reste du système. Par opposition au terme monolithique, la *modularité* est une mesure de la séparation du système en ses éléments constituants. La *composabilité* est une mesure des capacités d'assemblage entre les éléments constituants du système. La combinaison de ces deux propriétés autorise alors l'adaptabilité.

En ce qui concerne la modularité, sa granularité peut être plus ou moins fine et le couplage entre les différents modules plus ou moins lâche. Les modules peuvent prendre différentes formes selon le contexte technique de construction du logiciel. Une bibliothèque de code en langage C, une classe Smalltalk, un composant EJB [DmYK01], un aspect AspectJ [KHH+01] sont des exemples de modules. Les modules sont plus ou moins réifiés selon le cycle de vie du logiciel : certains sont présents à l'exécution (classe Smalltalk) alors que d'autres disparaissent à la compilation (aspect AspectJ) autorisant ainsi ou non l'adaptabilité à l'exécution. Enfin, lors de l'adaptation, un couplage lâche entre les modules favorise le minimum d'interférences avec le reste du système.

En ce qui concerne la composabilité, de nombreux travaux restent à faire pour la construction de systèmes réellement *plug-and-play*. Récemment, les approches à base de composants ou d'aspects ont mis l'accent sur cette problématique, en réifiant par exemple les connections entre modules (composants et connecteurs des ADLs [MT00]) ou en définissant des langages de tissage de modules (aspects [KLM+97]).

[1] Les problématiques connexes telles que l'évolution du logiciel (*software evolution*) [IEE01], plus éloignées du cadre de notre étude, ne sont pas abordées.

Introspection. L'introspection est la capacité d'un système à s'observer et donc à répondre à des questions sur son état. Cette propriété est essentielle pour l'adaptation de systèmes complexes. Signalons que les langages récents ayant un grand pouvoir d'expression (Java, C♯) proposent de base des mécanismes d'introspection et que les plates-formes intergicielle du marché ont recours à l'introspection pour leur fonctionnement (EJB [DmYK01], .NET [Mic]).

Un candidat idéal à l'introspection : la réflexion. Transposée du domaine philosophique au domaine informatique dans le début des années 80 [Smi82], la *réflexion* a trouvé un grand écho auprès des concepteurs de systèmes complexes. Elle propose la *réification*, processus permettant d'encoder, à l'exécution, les données propres au système ; l'*introspection*, mécanisme permettant au système de se regarder et donc de raisonner sur son propre état ; l'*intercession*, mécanisme permettant au système d'agir sur lui-même, et donc d'altérer sa propre interprétation. Contrairement à l'introspection, l'intercession n'est pas un prérequis à l'adaptabilité. En effet, le processus d'adaptation peut être mis en œuvre par un administrateur externe.

2.1.3 Étapes de l'adaptation

Une adaptation se décompose en trois étapes successives :

Déclenchement. L'étape de déclenchement consiste à détecter et à notifier d'un changement. Ce déclenchement peut être effectué par différents acteurs (par ex. utilisateur, sonde logicielle) et à différents moments (par ex. déploiement, exécution).

Décision. L'étape de décision consiste à déterminer les modifications qui doivent être effectuées pour réagir aux changements détectés dans la première étape. Ces choix peuvent être effectués par différents acteurs (par ex. programmeur, déployeur), concerner différents sujets à adapter (par ex. bibliothèque C, ensemble de classes Java) et suivre différentes règles ou stratégies d'adaptation.

Réalisation. L'étape de réalisation concerne tous les moyens qui vont être mis en œuvre pour appliquer la décision prise à l'étape précédente. La réalisation peut être effectuée par différents acteurs (par ex. déployeur, intergiciel) et selon différents mécanismes. La réalisation s'opère sur les sujets à adapter qui ont été définis dans l'étape de décision.

Ce découpage permet de mettre en évidence les différents éléments intervenant dans une adaptation : les acteurs, les moments, les sujets à adapter

et les mécanismes. Certains de ces éléments sont dépendants, le mécanisme d'adaptation dépend par exemple du sujet de l'adaptation, alors que certains autres peuvent être indépendants, l'acteur déclenchant l'adaptation peut être différent de celui qui va décider de la stratégie à suivre. Ces éléments sont détaillés dans les paragraphes 2.2 à 2.5.

2.2 Sujet de l'adaptation (Quoi ?)

La première chose à définir est le sujet de l'adaptation. Pour pouvoir y répondre, nous allons tout d'abord supposer qu'un système logiciel est une architecture composée d'un ensemble de *boîtes*[2] reliées entre elles. La cible de l'adaptation peut être soit une boîte, soit une liaison (entre deux boîtes), soit l'architecture elle-même (c-à-d la composition de n boîtes)[3].

Boîte. Sa granularité peut être plus ou moins fine et sa durée de vie, dans le cycle de vie du système, plus ou moins longue (cf. §2.1.2). L'adaptation d'une boîte consiste à appliquer un processus de modification sur cette dernière. Cela peut aller d'un simple paramétrage à une transformation complète (cf. §2.5).

Liaison. Une liaison est une relation entre deux boîtes. Un lien d'héritage, un bus logiciel sont des exemples de liaison de granularité différente. La modification, le changement de nature de la liaison entre deux boîtes sont d'autres cibles potentielles de l'adaptation. Par exemple, qui n'a jamais fait un travail de rétro-conception dans le cadre de la technologie objet pour utiliser un lien de délégation entre deux classes plutôt qu'un lien d'héritage ! Aujourd'hui, différents travaux de recherche [BS97] [Ber01] proposent cette voie pour supporter l'adaptabilité dans un contexte d'interopérabilité entre composants.

Architecture. La cible de l'adaptation peut être l'architecture elle-même. Il s'agit alors de réaliser un nouvel assemblage des boîtes. Il peut être mis en œuvre de différentes manières : soit par retrait, remplacement, reconfiguration de certaines boîtes et/ou liaisons, soit par ajout de nouvelles. Par exemple, le schéma de conception Strategy [GHJV95] fournit un cadre privilégié pour ce type d'adaptation : il correspond à une reconfiguration de liaisons entre boîtes selon certaines conditions.

[2]Nous avons délibérément choisi ce terme, *neutre*, plutôt que module ou composant pour ne pas présupposer de sa représentation et de son implémentation.

[3]Le cas où l'architecture peut être perçue comme une boîte elle-même, en respectant un modèle hiérarchique de boîtes, nous ramène à deux sujets pour l'adaptation.

2.3 Acteur de l'adaptation (Qui ?)

Après avoir étudié ce que nous pouvons adapter, nous allons essayer de déterminer quels sont les acteurs impliqués dans cette adaptation. Un acteur intervient à chacune des étapes de l'adaptation (cf. §2.1.3), mais les acteurs des différentes étapes ne sont pas forcément les mêmes. Ainsi, l'acteur déclenchant l'adaptation peut être différent de celui qui va décider de la stratégie à suivre. Nous distinguons trois types d'acteurs :

Programmeur. Lorsque l'adaptation est prévue à l'avance, le programmeur peut décider de figer une ou plusieurs des étapes d'adaptation. Les choix sur les sujets à adapter, les règles d'adaptation et les mécanismes peuvent être fixés à l'implémentation. Éventuellement, les règles d'adaptation peuvent être fusionnées avec le sujet de l'adaptation (par ex. conditionnelle, le schéma de conception Strategy [GHJV95]). L'adaptation est automatique, mais ses étapes sont fusionnées et donc non modifiables.

Administrateur. Si l'on désire ne pas figer une des étapes de l'adaptation à l'implémentation, il est possible d'effectuer les choix au lancement du système ou pendant son exécution. Ces choix peuvent être faits par un administrateur - nom générique représentant un déployeur ou un utilisateur - qui peut alors décider du déclenchement de l'adaptation, des sujets à adapter, de la stratégie ou des mécanismes. L'adaptation (ses étapes) est réifiée, mais elle est non automatique.

Logiciel. Si l'on se place dans la même hypothèse que ci-dessus (c-à-d étapes non figées à l'implémentation et choix retardés), les choix d'adaptation peuvent également être pris par une entité logicielle. Par exemple, le déclenchement de l'adaptation peut s'opérer grâce à des sondes pour l'observation de ressources et des notifications de changement. La décision d'adaptation peut, par exemple, être assurée par un système expert. La réalisation de l'adaptation peut être laissée à un intergiciel spécialisé. D'un point de vue systémique, on considère que ce dernier cas propose la notion d'auto-adaptation puisque le logiciel peut modifier son propre comportement en fonction de son contexte d'exécution. L'adaptation est automatique et réifiée.

2.4 Moments de l'adaptation (Quand ?)

Les moments de l'adaptation dépendent des choix faits précédemment, à savoir quoi adapter et qui adapte. Il existe trois temps pour l'adaptation : avant l'exécution du système, au lancement du système et pendant l'exécution du système. Les moments des différentes étapes (cf. §2.1.3) ne sont pas forcément les mêmes.

Avant l'exécution du système. Pour autoriser des adaptations ultérieures, il est nécessaire en phase de conception et développement de prévoir les possibilités d'adaptabilité du système et de faire des choix en conséquence. Nous avons vu en §2.1.2 que la modularité et la composabilité font partie des prérequis pour l'adaptabilité. Certaines approches fournissent des moyens d'expression intéressants pour favoriser celles-ci (ADL [MT00], aspects [KLM+97]). De plus, si l'on s'intéresse à l'adaptabilité à l'exécution, il faut choisir une approche permettant aux boîtes et/ou à leurs liaisons d'exister explicitement à l'exécution, et pas seulement au moment du codage. Par exemple, les intergiciels actuels proposent de séparer les services métiers des services d'infrastructure sans toutefois réifier ces derniers à l'exécution.

Ensuite, une adaptation statique peut être réalisée à la compilation, à l'édition des liens en fonction de la plate-forme cible.

Au lancement du système. Au moment du déploiement, l'administrateur (cf. §2.3) connaît l'état de l'environnement d'exécution et sa topologie. À cet instant ponctuel, il est donc amené à faire des choix pertinents d'adaptation. Il s'agit d'une adaptation statique en amont de l'exécution du programme.

Pendant l'exécution du système. Dans certains cas, il est possible de modifier à l'exécution le sujet d'adaptation (cf. §2.2). On parle alors d'adaptation ou de reconfiguration dynamique[4]. L'installation du décodeur MPEG à la volée, présenté dans le scénario de l'introduction, constitue un exemple d'adaptation dynamique de la liaison. Elle peut être effectuée par un tiers ou être réalisée automatiquement (cf. §2.3).

2.5 Mécanismes pour l'adaptation (Comment ?)

Dans cette section, nous allons identifier les mécanismes pour réaliser l'adaptation d'un système logiciel. Le sujet de l'adaptation sera le point de départ de cette réflexion. Cherche-t-on à adapter la boîte, la liaison entre deux boîtes ou l'architecture globale ? À chaque cible de l'adaptation est associé un mécanisme différent d'adaptation.

2.5.1 Préambule

Le choix d'une technique parmi les mécanismes pour réaliser l'adaptation est non seulement lié au sujet de l'adaptation et à ses caractéristiques mais aussi à certains critères de choix sur l'adaptation.

[4]Le fait qu'il faille désactiver (arrêter) le système pour faire une mise à jour nous ramène au cas précédent avec une configuration statique.

Caractéristiques du sujet de l'adaptation. Le sujet de l'adaptation (boîte, liaison, architecture) présente un certain nombre de caractéristiques qu'il est essentiel de détailler à ce niveau de discours pour élaborer une classification de techniques d'adaptation.

- *Sujet de l'adaptation.* Nous allons considérer qu'il est composé d'un état, d'une interface qui expose ses fonctionnalités et d'un code qui implémente ses fonctionnalités. L'adaptation concerne-t-elle l'état du sujet, le code du sujet ou son interface seulement ?
- *Type du sujet.* Le sujet suit-il une philosophie de type *white box* (c-à-d il expose son implémentation) ou de type *black box* (c-à-d il expose son interface seulement) [Kic94] ? Pour plus de clarté, nous parlerons par la suite de sujet d'adaptation « noir » ou « blanc ».

Critères de choix. Certains critères connexes aux caractéristiques du sujet de l'adaptation doivent aussi être pris en compte.

- *But de l'adaptation.* L'adaptation a-t-elle pour but de modifier la sémantique du sujet ou bien au contraire de la préserver ?
- *Identité.* Après adaptation, désire-t-on garder une référence sur le sujet de l'adaptation lui-même ou peut-on accepter de désigner un mandataire de ce sujet ?

2.5.2 Quelques techniques de base pour l'adaptation

Nous allons maintenant présenter quelques techniques de base pour réaliser l'adaptation. Elles seront présentées selon le sujet de l'adaptation. Nous allons insister principalement sur le type du sujet (« noir » ou « blanc ») qui nous a paru être le critère le plus important. Notons que les techniques qui fonctionnent sur les sujets d'adaptation « noir » marchent aussi pour les sujets d'adaptation « blanc ».

Techniques pour adapter la boîte. Si l'on s'intéresse à l'adaptation de la boîte *seulement*, il est possible de changer son état ou son code[5].

Paramétrisation. La paramétrisation est un cas notable de configuration où le système est construit avec un ensemble de paramètres prédéfinis dont les valeurs peuvent être spécifiées après construction du système. La paramétrisation de la boîte est possible quand celle-ci est noire.

Transformation. Cette technique consiste à modifier directement le code de la boîte. La modification de la boîte est seulement possible quand celle-ci est blanche. La transformation peut être faite « à la main », mais peut aussi être réalisée par des technologies comme la spécialisation de programme [JGS93] ou le tissage d'aspects [KLM+97]) (cf. §3).

[5]Le changement d'interface sans changement de code concerne ses relations avec l'extérieur et non pas l'intérieur de la boîte.

Techniques pour adapter la liaison. L'adaptation de la liaison peut être réalisée par deux voies radicalement différentes. Le type de la liaison et celui des boîtes de la liaison constituent le critère de classification.

Interposition. Cette technique est utilisée dans le cas où la liaison est blanche. La communauté système propose depuis longtemps des techniques d'interposition comme mécanisme de liaison avec la volonté d'adapter la liaison elle-même [BS97]. Cette liaison peut prendre des formes très diverses comme par exemple l'association talon + squelette + protocoles de communication + réseau. Le concept de *fabrique de liaisons* permet d'adapter aisément la liaison [DHDTJB98].
Parallèlement, la communauté langage s'est intéressée aux liaisons blanches pour contourner le problème d'adaptation des boîtes noires. En effet, dans ce cas, l'interposition peut jouer le rôle d'interception. L'idée est alors de placer une boîte `B'` dans la liaison devant la boîte noire `B` à adapter pour réaliser une interception et utiliser cette indirection pour adapter le comportement de `B`. La communauté génie logiciel évoque aussi les encapsulateurs (*wrappers*) qui jouent le rôle d'intercepteurs. La principale différence réside dans le fait que la boîte `B` est incluse dans la boîte `B'` et donc considéré comme invisible du monde extérieur. On retrouve la même idée avec les conteneurs EJB.

Délégation. Cette technique est utilisée dans le cas où la liaison est noire, mais les boîtes de la liaison, blanches. La délégation consiste à délèguer un travail d'une entité vers une autre entité. La mise en place d'une délégation dans la boîte blanche, génère une autre liaison, mais celle-ci peut être perçue comme une extension et donc une adaptation de la liaison noire initiale. La délégation peut être soit introduite explicitement par le concepteur, soit de façon transparente par des techniques de compilation ou de chargement (*hooks* [BSLS01]).

Techniques pour adapter l'architecture. Si l'on s'intéresse à l'adaptation de l'architecture complète, il est possible d'engager une procédure de *configuration/reconfiguration.* La configuration (resp. reconfiguration[6]) de l'architecture consiste en une composition (resp. recomposition) des boîtes à l'intérieur de l'architecture. Cette configuration/reconfiguration peut donc être réalisée soit par retrait, remplacement, paramètrage de certaines boîtes et/ou liaisons, soit par ajout de nouvelles. L'ensemble des techniques de base présenté ci-dessus peut être utilisé pour réaliser la configuration/reconfiguration.

[6] Une configuration issue d'une première configuration.

2.5.3 Autres considérations

Jusqu'à présent, nous nous sommes intéressés à l'adaptation du sujet de l'adaptation par *modification* de ce dernier. Il est possible d'adapter ce dernier non pas en le modifiant, mais en changeant le regard que l'on pose sur lui. Alors, quelque soit le sujet de l'adaptation (boîte, liaison, architecture), il peut être interprété différemment et donc adapté par *interprétation*. Cela revient à prendre en compte l'interpréteur lui-même comme un nouveau sujet d'adaptation. C'est ce que propose la philosophie d'implémentation ouverte (*open implementation*) de G. Kiczales [Kic96].

3 Technologies pour l'adaptabilité

Dans cette partie, nous proposons une synthèse des différentes technologies actuelles offrant un support à l'adaptabilité.

3.1 Technologie objet

Dans le cadre du paradigme objet, il existe un type d'adaptation basé directement sur les techniques objets : composition, héritage, polymorphisme, liaison dynamique, délégation, généricité.

Ce type d'adaptation, qui est apparu dès le début des langages à objets et a très certainement largement contribué à leur succès, se situe comme nous allons le voir à un niveau de granularité assez faible. C'est donc une adaptation plutôt "in the small", contrairement à celle qui focalise nos réflexions. Cependant, ces techniques sont particulièrement pertinentes pour nos investigations car elles constituent très souvent des mécanismes de base utilisés pour l'adaptation de plus haut niveau portant sur les architectures logicielles. Nous allons donc évoquer rapidement ces techniques, sous le regard de l'adaptation.

La *composition* d'éléments logiciels existe depuis le début de la programmation structurée. Composition de modules, puis composition de classes, on peut ainsi assembler statiquement, par une relation dite *de clientèle*, un ensemble de codes. Cette composition de classes permet donc une certaine adaptation, puisqu'il est possible de changer la composition, par exemple remplacer une classe par une autre. La nature statique et fine de cette adaptation ne saurait nous échapper : cette technique repose sur l'utilisation de classes (pour accéder à des routines statiques par exemple) ou la déclaration d'instances dans le source du programme.

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'évolutivité des logiciels objets[7]. 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. Si cette technique est par construction statique (elle passe par l'écriture de code source), les environnements modernes offrant tous le chargement dynamique, on peut ainsi adapter des applications sans nécessairement arrêter leur exécution.

La *délégation* permet à un objet qui offre une méthode publique de réaliser l'implémentation de cette dernière par un simple et unique appel à un autre objet ; l'objet *délègue* le travail à une autre entité [Lie86]. Si le choix de déléguer est statique (lors de l'écriture du code de la classe), nous abordons là tout de même un mécanisme plus dynamique que la composition car on peut adapter une telle délégation à l'exécution : il suffit par exemple d'affecter un attribut cible de délégations avec une nouvelle occurrence. Notons que dans les dernières années, la délégation a été érigée en technique standard d'adaptation par le biais d'une utilisation généralisée et intensive des *interfaces*. Elles apportent aux architectures logicielles actuelles une amélioration indéniable de leur flexibilité, et contribuent à leur adaptabilité statique et dynamique.

Enfin, la *généricité* des langages à objets permet d'écrire une classe où certains des types utilisés sont des paramètres. Contrainte ou non, la généricité apporte essentiellement une détection statique d'erreurs de typage (à l'utilisation ou à l'édition de liens), sans imposer l'utilisation de transtypages explicites. Par exemple, un compilateur pourra vérifier qu'une `liste de figures` ne contient que des figures, et lors de l'extraction d'un élément de la liste, un transtypage ne sera pas nécessaire pour affecter celui-ci dans une variable de type `figure`. On peut résumer très sommairement la généricité comme la capacité à générer une classe à partir d'une classe préexistante par substitution de types. En regard de l'adaptation, il convient tout d'abord de noter que cette généricité doit être planifiée à l'avance : lors de la définition d'une classe, le concepteur doit préciser les types qui sont effectivement génériques. Ceci est à comparer à la redéfinition où les méthodes sont en général virtuelles par défaut. De par son caractère planifiée, la généricité ne semble donc pas être un mécanisme de base pour l'adaptation. L'exemple bien connu de C++ où le mécanisme des *template* est "Turing complete" à la compilation ne semble pas changer cet aspect : il faudra toujours planifier les générations de code pouvant intervenir. En perspective, et pour moduler

[7]Nous verrons plus loin qu'elle joue un rôle important dans certaines techniques réflexives, cf. section 3.3.

notre jugement, notons que la notion de "class substitution", un nouveau mécanisme de généricité proposé par J. Palsberg and M. Schwartzbach [PS94], pourrait faire disparaître cette inadéquation. En effet, il est alors possible de générer une substitution de type *a posteriori* sur une classe, et donc de l'adapter par généricité sans que cela ne soit prévu à l'avance.

3.2 Technologies de compilation et support d'exécution

Il existe deux manières primitives d'adapter un logiciel, la première consiste simplement à changer son code, la deuxième à changer son interprétation (voir section 2.5.3) en modifiant le support d'exécution sous-jacent.

Changer du code est spécialement simple dans le cas d'un langage interprété, il suffit que l'interprète relise les morceaux de programme à modifier (par exemple, par l'intermédiaire des métaprédicats `assert/retract` en Prolog). Dans le cas d'un langage compilé, cette lecture de code source devient un chargement de code compilé. Cette opération est simple s'il est possible de déterminer que le code modifié n'est pas en cours d'utilisation. Les choses se compliquent si ce n'est pas le cas. Ainsi, lorsqu'un langage à objets est utilisé, un scénario typique est la modification d'une classe dont il existe déjà au moins une instance. Dans un cadre qui garantit statiquement certaines propriétés (typage, sécurité), il s'agit aussi de garantir que la nouvelle version est compatible avec la version précédente. Traiter ces questions requiert un support d'exécution adapté, par exemple un certain nombre de modifications de la machine virtuelle Java dans le cas du chargement dynamique de classes proposé par [MPG+00].

Un point intéressant est que, plutôt que de charger un code précompilé, il est possible d'effectuer une recompilation prenant en compte l'état de l'exécution ayant déclenché l'adaptation, et donc de faire de la génération de code dynamique [KEH91]. Le problème de la génération de code dynamique est son coût potentiel dans le cas où il y a des contraintes de réaction à la demande d'adaptation. Une possibilité est de compiler les « briques » nécessaires à l'adaptation à l'avance pour juste les assembler de manière adéquate à l'exécution. La génération de code dynamique peut alors s'accompagner d'un processus de spécialisation, par exemple, en utilisant des techniques d'évaluation partielle, comme proposé dans [CN96].

En effet, la demande d'adaptation correspond habituellement à de nouvelles valeurs de paramètres significatifs du contexte d'exécution. Il est alors fondamental d'utiliser au mieux ces données. Bien évidemment, ces techniques de spécialisation peuvent aussi être utilisées statiquement, pour des besoins de configuration. Tempo, évaluateur partiel pour C [CHN+96], ainsi que son adaptation à Java, JSpec [SLCM99], permettent de combiner ces deux aspects.

Ces idées de génération dynamique de code et de spécialisation sont utilisées de manière très ciblée dans des supports d'exécution comme Vortex

[DDG+96] et HotSpot [SUN01] qui adaptent en continu la qualité du code en générant dynamiquement du code plus efficace pour les points chauds repérés par profilage.

D'un certain point de vue, la combinaison d'un compilateur et d'une machine virtuelle n'est rien d'autre qu'un interprète. On voit donc que recompiler un programme (en supposant que le source ne change pas) peut être considéré, en référence au programme source, comme un premier type de modification du support d'exécution. Une deuxième point d'attaque est la machine virtuelle. Le projet VVM (*Virtual Virtual Machine*) [FPR98] propose ainsi de « virtualiser » la machine virtuelle, qui devient programmable par chargement d'une spécification de machine virtuelle, appelée *VMlet*. On peut alors envisager de changer l'interprétation d'un programme en changeant de VMlet. C'est ce même genre d'idée que l'on va retrouver, à un autre niveau, dans la section suivante sur les techniques réflexives, où les VMlets seront remplacées par des métaobjets !

3.3 Techniques réflexives

La réflexion caractérise la propriété d'un système capable de raisonner et d'agir sur lui-même (cf. section 2.1). Cette section se propose de faire une classification des techniques de réflexion. Nous pouvons, par exemple, faire la distinction entre réflexion structurelle et comportementale, comme le fait Ferber [Fer89], ou alors même entre réflexion implicite ou explicite, suivant que le niveau de base a connaissance ou non de l'existence du niveau meta.

Afin de bien mettre en avant les particularités de chacune des techniques d'implémentation, nous avons choisi d'utiliser une classification basée sur le moment auquel les méta-objets existent et sont utilisés (cf. figure1). Dans le cadre du modèle d'exécution de Java, nous aboutissons ainsi à trois catégories qui correspondent aux trois grandes phases de la vie d'un programme Java : la compilation, le chargement des classes dans la machine virtuelle, et l'exécution. Nous utiliserons par la suite les trois termes anglais *compile-time*, *load-time* et *run-time* pour designer ces trois phases. Comme un MOP peut utiliser des méta-objets à différents moments de la vie d'un programme, nous aboutissons au total à six cas, qui se réduisent au nombre de quatre si l'on ne considère que l'étape la plus statique.

1. *Pure compile-time MOPs.* Ils effectuent une traduction source a source. La traduction est décrite par le méta-programme, les méta-objets représentant les classes, les méthodes, les déclarations et autres instructions du programme de base. OpenJava [CT98], le successeur d'Open C++, est un excellent exemple de cette catégorie de MOPs.
2. *Pure load-time MOPs.* Ils effectuent sur la version compilée d'une classe (*bytecode*) la même chose que les MOPs compile-time sur le code source. Les MOPs *load-time* sont souvent implementés à travers des chargeurs

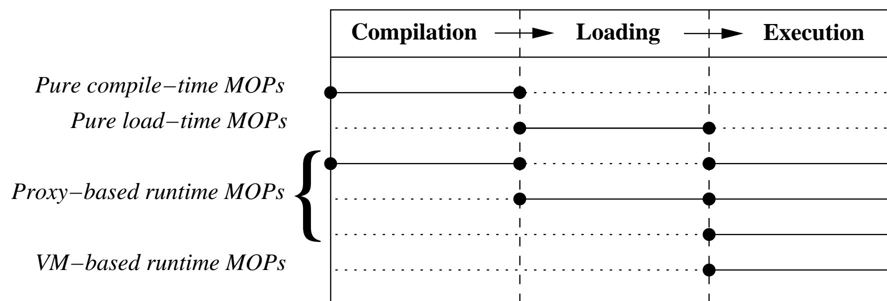

FIG. 1 – Une classification des MOPs basée sur le moment auquel les méta-objets sont utilisés.

de classe (*classloaders*) spécialisés. Kava [WS99] et Javassist [Chi00] appartiennent à cette catégorie.

3. *Proxy-based MOPs.* Les MOPs de ce type introduisent des objets intermédiaires (*proxies* ou *wrappers*) afin de dérouter les appels de méthodes vers le niveau méta. L'introduction de ces *proxies* peut se faire à la compilation (OpenJava), au chargement des classes (Kava) ou encore à l'exécution, plus précisément lors de la création des objets. Reflective Java [WS97], ProActive [CKV98], RAM[BSLS01] et Reflex [TBSN01] rentrent tous dans cette catégorie.
4. *VM-based MOPs.* Ces MOPs utilisent une machine virtuelle dont le fonctionnement a été modifié afin de permettre l'interception de certains événements (appels de méthodes, accès à des champs, prise ou libération des verrous, ...). Ces solutions font bien évidemment perdre à Java ses qualités de portabilité. Dans cette catégorie de MOPs, on trouve entre autres MetaXa [KG96], Guarana [OB99], Correlate [TVJ+01] et Iguana/J [RC00].

3.4 Aspects et tissage

La programmation dite par aspects vise à permettre une séparation des différentes préoccupations des programmeurs face à la réalisation d'un programme. Elle est certainement une des voies les plus prometteuses pour l'adaptabilité des composants. La richesse de la littérature ces dernières années traitant de cette nouvelle approche de la programmation témoigne de combien il paraît utile de prendre en compte le caractère hétéroclite des différents savoirs pour permettre leur intégration de façon plus fiable et plus facile. Dans le paradigme de la *programmation par aspects* [KLM+97], il devient possible d'implémenter des comportements s'entrelaçant avec le reste du système d'une manière modulaire. La programmation par aspects s'attache tout particulièrement à permettre l'expression séparée des parties techniques et des parties fonctionnelles des applications, bien que de récents tra-

vaux se soient intéressés à décomposer également les parties fonctionnelles. Un aspect est une abstraction d'un comportement ou d'un rôle, dont la particularité est de s'appliquer à un ensemble de classes, voire d'objets. Ainsi un *aspect* correspond à la définition de structures et/ou comportements qui se superposent à la définition de l'application métier dite aussi aspect de base [BSL01]. Les *points de jointures* sont les endroits où les aspects interviennent sur le reste du système. Comme les aspects sont des modules à part entière, il faut pouvoir les composer à nouveau avec le reste du système. L'entrelacement entre les aspects, aussi nommé tissage (*weaving*), est alors pris en charge par un « outil » dédié (le *weaver*). C'est ce dernier point qui nous permet de séparer les techniques de méta-programmation de celles de la programmation par aspects. Celle-ci s'appuie très souvent sur des techniques de méta-programmation par transformation de programmes, modification de l'interprète ou manipulation dynamique de méta-objets pour atteindre son but. Cependant, elle vise à le faire non pas par manipulation directe des méta-comportements mais par une voie déclarative, laissant les outils gérer les entrelacements. La composition des aspects est probablement le défi non encore relevé de cette nouvelle approche de la programmation. À l'instar des techniques de méta-programmation sur lesquelles il s'appuie, le paradigme de la programmation par aspects, cherche à prendre en compte la décomposition en aspects à toutes les étapes du développement du logiciel.

AspectJ [KHH+01] est une extension orientée aspect du langage Java. Elle ajoute quelques nouvelles constructions syntaxiques pour déclarer les aspects et les points de jointures ainsi qu'un tisseur pour composer les aspects et les classes. Ce tisseur est basé sur la transformation de programme produisant ainsi du code Java. Les programmes utilisant AspectJ sont donc compatibles avec toute machine virtuelle Java. La composition des aspects est alors à la fois liée aux constructions syntaxiques (*before*, *after*, *handler*, ...) et à l'ordre des déclarations. Appliquer cette technologie pour l'adaptation des composants au déploiement est assurément une voie intéressante de définition des outils de déploiement.

Alors que cette approche met l'accent sur l'intérêt d'exprimer des préoccupations transversales aux classes et n'offrent pas de solutions déclaratives aux utilisateurs sur la composition d'aspects, les travaux autour de la programmation orientée sujets [HO93] ont privilégié la possibilité d'exprimer différents points de vue sur les objets dans des sujets totalement indépendants qui peuvent ensuite être combinés à partir de règles de combinaisons. La programmation orientée sujets est une extension du paradigme de la programmation par objets. Un sujet est un programme ou un fragment de programme exprimé dans le paradigme objet. La programmation orientée sujets propose de composer un ensemble de sujets pour produire l'application finale. Ce processus de compilation est nommé intégration. Les sujets sont tous vus au même niveau, aucun sujet n'est a priori dominant sur un autre. Les sujets peuvent être composés entre eux pour produire des sujets plus im-

portants, qui peuvent être à leur tour composés entre eux. L'intégration des sujets se fait en définissant des règles. Une règle implique deux ou plusieurs sujets, elle permet de définir par des opérations simples (fusion, redéfinition et séquence) la sémantique de la composition.

Ces différentes mises en œuvre ne permettent pas d'intégrer dynamiquement de nouvelles préoccupations à un programme existant. Pour pallier ce manque, très limitatif en terme d'applications évolutives adaptables, différentes propositions ont été faites.

Certains s'intéressent aux aspects sous la forme de wrappers ou extensions qui permettent un contrôle adaptable dynamiquement (JavaPod [BR01], AD-Tos [PDF$^+$02] et JAC [PSDF01]). Ainsi, ces travaux se présentent comme des « framework ». À la différence de AspectJ, ils sont basés sur la définition de *wrappers* de façon programmée offrant de ce fait une plus grande puissance d'expression des points de jonctions. Ces approches présentent la particularité de permettre la manipulation (ajout, retrait ou re-ordonnancement) des aspects de façon dynamique, autorisant ainsi une adaptation en fonction de l'environnement.

D'autres travaux visent également à appréhender les aspects comme des entités de l'application. Ainsi les filtres de composition [BA00] ajoutent aux objets des filtres qui interceptent l'envoi et la réception de messages. Les messages entrants sont soumis à l'ensemble des filtres d'entrée, de même pour les sortants. Un filtre peut accepter ou rejeter un message tout en déclenchant un certain nombre d'actions. Les filtres sont ordonnés dans un ensemble (de filtres d'entrée ou de sortie), c'est leur seule capacité de composition immédiate. Cette approche laisse percevoir la nécessité d'une évolution du concept de composant pour lui intégrer une dimension composants d'aspects. Dans la même veine, se situe le travail sur les interactions [DBFM$^+$01, DBFM02, Ber01]. Il entre dans la famille de la programmation par aspects dans la mesure où une interaction peut être intégrée dynamiquement et a posteriori aux fonctionnalités de base des objets. La composition des nouvelles interactions avec les interactions existantes est basée sur des règles de fusion établies à partir de la définition d'opérateurs dont la sémantique naturelle a été établie indépendamment des langages cibles. Ce mécanisme de fusion automatique permet par exemple de détecter des incohérence entre aspects, et d'ordonner automatiquement certains aspects. De plus, l'approche proposée n'est pas dépendante d'un langage cible (des mises en œuvres en Smalltalk, Java et C++ sur des plates-formes RMI, CORBA et EJB ont été expérimentées) ce qui permet l'intégration de composants hétérogènes. Ce travail ouvre des perspectives à la fois en matière d'adaptabilité dynamique des composants et également de gestion automatique des compositions.

3.5 Architectures logicielles

Les travaux en architecture visent à maîtriser la construction et le déploiement (installation, évolution, etc.) de systèmes logiciels dans des environnements hétérogènes et évoluants. Ils sont donc concernés au premier chef par l'adaptabilité. Pour cela, ils cherchent à distinguer des *abstractions* dans les systèmes – des *modules* – et les *interactions* entre ces abstractions.

Il existe des *principes de conception généraux*, qui guident la spécification des modules et des assemblages afin de garantir une bonne adaptabilité. L'un des plus importants est le Principe d'Ouverture-Fermeture (B. Meyer), qui stipule que tout module doit être à la fois *ouvert* (il est possible de l'étendre) et *fermé* (sa description est stable et bien définie, donc il est disponible pour être utilisé dans un système) ; le module est alors dit *ouvert-fermé*.

Les *schémas de conception* (*design patterns*) [GHJV95] peuvent être considérés comme des "principes de conception locaux" (aux classes). Ils permettent d'ouvrir-fermer un système pour des objectifs couramment recherchés. Par exemple, le schéma de conception Strategy permet d'ouvrir-fermer un système à de nouvelles implantations d'une famille d'algorithmes.

Un *framework objet* est un ensemble de classes qui capture une conception abstraite résolvant une famille de problèmes, et permet donc d'ouvrir-fermer un système à une échelle souvent supérieure à celle d'un schéma de conception. Différentes techniques sont proposées pour spécialiser et étendre les *frameworks* (héritage de classes abstraites, délégation, etc.). Un problème lié aux *frameworks* est que les dépendances entre modules (des classes ici) sont implicites.

Dans les *langages de description d'architecture* (ADLs) [MT00], les modules sont appelés *composants* et leurs liaisons sont matérialisées par des *connecteurs* modélisant les interactions entre composants. Composants et connecteurs sont généralement accessibles uniquement au travers d'*interfaces* (*ports* pour les composants et *rôles* pour les connecteurs). Les ADLs sont utilisés pour raisonner (formalisation, validation) sur les assemblages (UniCon, Wright, Rapide, SADL), ou pour générer des squelettes de systèmes (Weaves, MetaH, Olan), ou les deux (Darwin, C2).

Cependant, un ADL ne permet de décrire un système que dans un seul point de vue, parce qu'il se base sur un méta-modèle figé. L'*architecture orientée-modèles* (MDA) de l'OMG [StOSSG00] propose d'étendre le méta-modèle d'UML, par des *profils* qui le spécialisent. Un modèle est spécifié dans différents profils, chaque profil pouvant être projeté vers différentes infrastructures techniques (CCM ou EJB, SGBDR ou SGBDOO, etc.).

4 Utilisation dans les Intergiciels

Dans cette partie, nous examinons la place de l'adaptabilité dans le domaine des intergiciels.

4.1 Modèles de composants industriels

Les Enterprise Java Beans (EJB) de Sun [DmYK01], le modèle de composants CORBA (CCM) [BEA99], San Francisco d'IBM (SF) [And96] et COM de Microsoft [Mic95] sont les quatre modèles de composants et *frameworks* industriels les plus répandus. Ils sont bâtis sur les mêmes principes généraux, qui sont exposés ci-après, et ils convergent rapidement : intégration SF-EJB, EJB-CCM, prise en compte des communications asynchrones par CCM puis EJB puis COM, etc.

SF, EJB, CCM et COM sont des modèles de *composants applicatifs ou métiers*. Les composants représentent des processus (transferts bancaires, gestion d'ordres) ou des entités (comptes bancaires, employés, clients, etc.) utilisés couramment dans différents domaines d'activité d'une entreprise (finance, paie, gestion de stocks, de commandes, etc.). Les composants sont déployés dans des structures d'accueil : les *conteneurs*[8]. Les conteneurs ont deux rôles principaux : (1) ils gèrent le cycle de vie (création, accès, activation, passivation, destruction, etc.) des composants qu'ils contiennent et (2) leur fournissent automatiquement et de manière transparente un certain nombre de services techniques[9] (persistence, comportement transactionnel, concurrence, sécurité, etc.) par interception des interactions avec ces composants. Le paramétrage de ces services techniques est effectué de manière déclarative, au moment du déploiement, au travers de *descripteurs de déploiement*[10]. Les différents modèles décrivent principalement des *modèles de programmation* qui spécifient en réalité des *contrats* entre les composants et les conteneurs. Il existe un certain nombre de *catégories de composants* prédéfinis dans chaque modèle (*sessions, entités, procédés, messages*, etc.). Les contrats dépendent de ces types des composants. Ils spécifient notamment quels services techniques sont associés à chaque type de composants. Les différents modèles concernent principalement les points abordés ci-dessus : 1) catégories de composants considérés (sessions sans états dans COM - sessions, entités, messages dans EJB - entités, dépendants, commandes dans SF - sessions, entités, processus, services dans CCM) ; et 2) services techniques associés à chacun de ces types (nommage, cycle de vie, transactions, sécurité dans COM - cycle de vie, nommage, persistance, transactions, concurrence, sécurité dans EJB - idem + notification d'événements dans CCM - idem + requêtes dans SF), modèles de programmation associés. Enfin, parmi les différences notables, signalons que SF et COM, à la différence de EJB et CCM, ne sont pas des spécifications mais des produits (SF par exemple propose plus de 1100 composants directement utilisables).

Le support de l'adaptabilité offert par les modèles industriels peut être

[8]On peut considérer l'infrastructure transactionnelle MTS (Microsoft Transaction Server) comme le conteneur des composants COM.

[9]On parle également d'*aspects non fonctionnels*.

[10]Pour COM, le registre de Windows est utilisé comme descripteur de déploiement.

perçue à deux niveaux. D'une part, ce sont effectivement des modèles de composants, ils fournissent donc, de fait, une adaptabilité au niveau applicatif. D'autre part, et surtout, l'adaptabilité offerte par ces plate-formes est incarnée par le paramétrage déclaratif des services techniques au déploiement.

4.2 Intergiciels réflexifs

Certains intergiciels adaptables adoptent une approche réflexive. Ces systèmes offrent une représentation de leurs structures et autorisent, à des degrés divers, certaines adaptations. Selon les cas, ces adaptations peuvent être effectuées statiquement ou dynamiquement. Les architectures réflexives permettent, en outre, de séparer les propriétés non-fonctionnelles du code applicatif.

DART.
Le projet DART [RL98] vise à fournir un environnement adaptable pour les applications réparties, dans lequel des propriétés (persistance, duplication, monitoring, etc.) peuvent être dynamiquement attachées à des objets de base. DART utilise un espace plat de méta-objets (appelé méta-espace) qui peut être associé à un groupe. Un groupe d'objets de base est un ensemble d'objets qui partagent le même méta-espace. Lorsque le contrôle est transféré depuis un objet de base vers son méta-espace, tous les méta-objets composant le méta-espace sont invoqués séquentiellement par un gestionnaire du méta-espace.

FlexiNet.
Hayton et al. proposent d'utiliser la réflexivité pour construire une plate-forme modulaire et extensible, appelée FlexiNet [HHD98]. Ils proposent d'utiliser la réflexivité de façon limitée, uniquement dans les talons et squelettes. Ainsi, ils permettent d'obtenir des performances presque équivalentes à des plate-formes non réflexives. La composition des méta-objets est sous forme de listes et non de méta-espaces. Cela permet de séparer au niveau méta le code non-fonctionnel et de composer plus facilement le code de chaque propriété non fonctionnelle.

JavaPod.
JavaPod [BR01] est une plate-forme à composants permettant d'associer de façon transparente aux composants une sélection de propriétés non-fonctionnelles requises parmi une liste de propriétés ouverte et extensible.

L'architecture de JavaPod s'inspire fortement de celles des EJBs à qui elle emprunte notamment la notion de conteneur. Un conteneur JavaPod se construit par composition de composants appelés extensions. Il représente la partie système d'un composant, de la même façon qu'il existe une structure

système associée à un processus dans un système d'exploitation, ou à un objet dans une machine virtuelle Java. Un composant est un groupe d'objets co-localisés et fortement liés d'un point de vue fonctionnel. Le conteneur encapsule le composant, au sens où toutes les interactions du composant avec l'extérieur doivent normalement passer par le conteneur. Grâce à cette interposition, le conteneur peut gérer les propriétés non-fonctionnelles du composant : modèle d'exécution, de persistance, de synchronisation ou autre. Le conteneur joue donc le rôle de méta-objet. Il fournit un MOP afin d'adapter dynamiquement le comportement des objets qu'il gère.

Jonathan.
Jonathan est un environnement de programmation répartie écrit entièrement en Java. Jonathan a été développé à l'origine par le CNET (France Telecom R&D) dans le contexte du projet européen ReTINA, dont le but était de définir une architecture de systèmes répartis pour le monde des télécommunications. Les applications de télécommunications comme, par exemple, les services multimédia, ont de fortes contraintes en termes de passage à l'échelle, de flexibilité, et de temps-réel : Jonathan est une plate-forme ouverte, dans le sens où, contrairement aux plates-formes standard (et en particulier, à la plupart des ORBs CORBA), les éléments qui la constituent peuvent être utilisés par les développeurs d'applications pour satisfaires des contraintes spécifiques.

Jonathan est organisé autour d'un noyau extrêmement réduit, dont le seul rôle est de permettre la communication des composants de l'infrastructure. Actuellement, ces composants consistent essentiellement en un certain nombre de protocoles de communication ou de présentation, de gestionnaires de ressources, etc.

Différentes personnalités peuvent être construites à l'aide de ces composants. Une personnalité est un ensemble d'interfaces de programmation : Java RMI est une personnalité, CORBA une autre, COM une troisième... Jonathan propose actuellement deux personnalités. La personnalité nommée David implémente un ORB CORBA, tandis que Jérémie permet la programmation d'applications dans le style RMI.

OpenCorba.
OpenCorba [Led98] est basé sur le langage réflexif NeoClasstalk [Riv97], une évolution réflexive du langage Smalltalk fournissant un protocole de changement dynamique de classes et un contrôle de l'envoi des messages. Il utilise le paradigme de la séparation des aspects pour décorréler les mécanismes de répartition du programme fonctionnel. OpenCorba est un ORB réflexif implémentant la plate-forme CORBA de l'OMG dans NeoClassTalk. La séparation des aspects et l'association des méta-objets (représentant les aspects réifiés) aux objets du programme se fait à la compilation des spéci-

fications IDL des interfaces du programme. Les aspects de CORBA réifiés par OpenCorba concernent principalement les mécanismes d'invocation via un proxy et le contrôle de type sur la classe serveur.

OpenOrb.
La plate-forme OpenORB [SC99] est une architecture expérimentale d'intergiciel réflexif dont l'objectif est d'offrir plus de configurabilité, de reconfigurabilité et de suppporter des reconfigurations à long terme. L'approche choisie est un compromis entre une simple technologie à composants et la réflexion. Le modèle de composants proposé s'inspire largement de RM-ODP [Ray95]. Un méta-espace, chargé d'implémenter les fonctions réflexives, est associé à chaque composant de la plate-forme. La réflexion est à la fois structurelle (on peut accéder à la représentation d'un composant) et comportementale (on peut modifier l'exécution d'un composant à l'aide d'intercepteurs). L'utilisation de contraintes architecturales permet de maintenir l'intégrité de la plate-forme. Les concepts de l'architecture d'OpenORB sont appliqués à tous les niveaux du système.

5 Synthèse

5.1 Vers une comparaison

Dans la section 2, nous avons proposé un certain nombre de principes et critères généraux autour de l'adaptabilité. Les sections 3 et 4 ont présenté différents travaux menés autour de cette problématique. Nous nous proposons à présent de rediscuter plus précisémment, dans ce contexte, des intergiciels et des technologies développées par les différents partenaires du projet. Cette étude nous permettra d'établir une grille de comparaison des différents supports à l'adaptabilité proposés dans le projet (cf. tableau 1). Dans un premier temps, nous allons repréciser les différents critères à prendre en compte :

- *Sujet de l'adaptation.* Pour déterminer la cible de l'adaptation, la première chose à faire est de réaliser la projection des différents modules manipulés vers les notions de boîte, liaison et architecture.
- *Acteur de l'adaptation.* Un acteur intervient à chacune des étapes de l'adaptation (cf. §2.1.3). Nous avons distingué trois types d'acteurs : le programmeur, l'administrateur et le logiciel. Ces derniers ne sont pas forcément les mêmes selon les différentes étapes. Par exemple, l'acteur notifiant l'adaptation peut être différent de celui qui va décider de la stratégie à suivre lui-même différent du réalisateur de l'adaptation.
- *Moments de l'adaptation.* Rappelons qu'il existe trois temps pour l'adaptation : avant l'exécution du système, au lancement du système et pendant l'exécution du système. Dans le processus d'adaptation, il faut

distinguer alors le moment de définition des règles d'adaptation, du moment d'application des règles d'adaptation.

– *Mécanismes.* Il faut identifier enfin les mécanismes de base pour réaliser l'adaptation (par ex. paramétrisation, transformation, interposition,...). Ces derniers dépendent du sujet de l'adaptation et de sa nature (sujet « noir » ou « blanc »).

OpenCorba (EMN). OpenCorba [Led98] est un bus logiciel réflexif implémentant les APIs CORBA. Il y est clairement établi une correspondance entre aspects non fonctionnels et métaclasses, de façon à mettre en œuvre la séparation et l'adaptabilité dynamique des aspects grâce à la réflexion. Plus précisement, un changement dynamique de métaclasse permet l'adaptabilité à l'exécution. Le sujet de l'adaptation est le lien méta (entre classe et métaclasse) ; l'acteur de l'adaptation est le programmeur lui-même (car la décison d'adaptation est « auto-contenue » dans l'application) ; le moment de définition des règles d'adaptation se situe avant l'exécution du système alors que le moment d'application de ces règles d'adaptation est pendant son exécution ; les mécanismes utilisés sont à la fois une délégation transparente vers les métaclasses et un protocole de changement dynamique de classe.

Interactions (Rainbow). Les interactions sont supportées par un service d'interactions dont une des mise en œuvre a été réalisée sur la plate-forme JONAS. Les sujets de l'adaptation pour les interactions sont les liaisons entre les composants et l'architecture de l'application par établissement ou destruction de ces liaisons. L'acteur de l'adaptation est le programmeur qui définit les interactions qui pourront assujettir son application, l'administrateur qui impose de nouvelles interactions en fonction du contexte de déploiement et enfin le logiciel qui peut décider de la mise en place de nouvelles interactions. Le moment de définition des règles d'adaptation est donc situé avant l'exécution du système (la boîte est préparée à interagir), les adaptations sont définies et s'appliquent pendant l'exécution. Les mécanismes utilisés sont de l'interposition et de l'interprétation de règles.

Jonathan (France Télécom R&D). L'adaptation dans Jonathan revêt deux aspects distincts.

Configuration Cet aspect est purement architectural. Jonathan est organisé comme un ensemble de canevas (*frameworks*). Chaque canevas représente une fonction "à gros grain" : gestion des liaisons, des communications, des ressources, de la configuration. Chaque canevas est constitué d'un ensemble de composants (qui sont accessibles uniquement au travers d'interfaces) dont Jonathan fournit une ou plusieurs implantations. Les composants sont spécialisables à volonté. Le canevas de configuration permet de

configurer Jonathan au déploiement (au lancement de l'application), c-à-d de sélectionner les composants utilisés dans chaque canevas (dans Jonathan 3, des descripteurs de configurations peuvent être exprimés en XML). Les *personnalités* David (CORBA) et Jeremie (RMI) de Jonathan, par exemple, peuvent être vues comme des configurations particulières de Jonathan.

Liaisons flexibles Comme dans RM-ODP [Ray95], une liaison dans Jonathan est un objet qui encapsule des ressources de communication et qui supporte donc une sémantique particulière de communication entre des interfaces liées. Les liaisons sont flexibles car elles sont construites par des entités spéciales appelées "usines à laisons". Les usines à liaisons permettent de créer et gérer des identificateurs d'interfaces ; et d'établir et gérer des liaisons d'un type particulier. Jonathan fournit plusieurs usines à liaisons du type client-serveur et associées aux personnalités David et Jeremie, une autre permet de créer des liaisons RTP multicast, etc. Le choix d'une usine à liaisons et les informations qui lui sont passées en paramètres permettent de créer des liaisons flexibles.

JavaPod (Sirac). Dans la plate-forme JavaPod, le sujet de l'adaptation est l'assemblage d'extensions qui constitue les conteneurs : un changement d'extension permet l'adaptation des aspects non fonctionnels des composants (une extension correspond approximativement à un aspect). Le mécanisme de base pour l'adaptation est donc le modèle de composition d'extensions utilisé par cette plate-forme, qui est similaire à la délégation à la Self [SU95]. En ce qui concerne les acteurs et les moments de l'adaptation, le prototype actuel ne permet qu'une adaptation au lancement du système, par le programmeur ou par le déployeur de l'application. Mais le mécanisme de composition d'extensions étant dynamique, on pourrait parfaitement compléter ce prototype en ajoutant un gestionnaire de politiques d'adaptation, qui réaliserait dynamiquement les adaptations nécessaires pour réagir aux notifications d'un service de surveillance (qu'il faudrait également ajouter).

ProActive (Oasis). ProActive est une librairie qui implémente un modèle à objets actifs distribués. ProActive repose sur un MOP générique, et tous les aspects non-fonctionnels du modèle sont configurables au travers de méta-objets qui modifient la sémantique de l'appel de méthode sur une référence donnée. Il est possible de changer le lien entre un objet de base et son méta-objet à l'exécution, et ceci objet par objet et non pas classe par classe, ce qui permet d'obtenir une adaptation à grain fin à l'exécution. Cette faculté d'adaptation dynamique est pleinement exploitée, par exemple, pour l'implémentation de la migration des objets actifs entre machines virtuelles. Les règles d'adaptation sont définies avant l'exécution du programme, mais le choix des règles et leur application se produit à l'exécution. Le principal

mécanisme utilisé est l'interception transparente d'appels de méthode grâce à des objets d'interposition. Le code des objets d'interposition est généré à l'exécution, ce qui permet de retarder jusqu'au dernier moment le choix des objets à adapter.

	Sujets	Acteurs	Moments	Mécanismes
OpenCorba	lien méta	programmeur	déf. avant	délégation
			app. pendant	prot. chgt classe
Interactions	liaison	tous	déf. avant+pendant	interposition
	architecture		app. pendant	
JavaPod	extensions	programmeur	lancement	délégation
		administrateur		(à la SELF)
Jonathan	liaisons	programmeur	avant, lancement	interposition
	architecture	administrateur		
ProActive	Références	programmeur	déf. avant	interposition
	appel de méthodes	administrateur	app. pendant	

TAB. 1 – Comparaison entre des travaux développés par les partenaires

5.2 Conclusion

Ce document avait pour but de faire un état de l'art sur l'adaptabilité. Aussi, dans un premier temps, nous avons défini un certain nombre de principes généraux et critères de l'adaptabilité. Puis, les sections technologies pour l'adaptabilité et intergiciels ont présenté différents travaux menés autour de cette problématique et proposant un support pour l'adaptabilité. Enfin, nous avons ébauché une synthèse axée autour des différents travaux des équipes du projet et s'appuyant sur les différents critères proposés.

Références

[And96] D. H. Andrews. IBM's San Francisco Project : Java Frameworks for Application Developers, December 1996.

[BA00] L. Bergmans and M. Aksit. *Software Architectures and Component Technology : The State of the Art in Research and Practice*, chapter Constructing Reusable Components with Multiple Concerns Using Composition Filters. Kluwer Academic Publishers, m. aksit edition, 2000.

[BEA99] BEA Systems et al. CORBA Component Model Joint Revised Submission. Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.

[Ber01] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO.* PhD thesis, Université de Nice, octobre 2001.

[BR01] Eric Bruneton and Michel Riveill. Experiments with javapod, a platform designed for the adaptation of non-functional properties. In *Reflection 2001, LNCS 2192*, September 2001.

[BS97] Gordon S. Blair and Jean-Bernard Stefani. *Open Distributed Processing and Multimedia.* Addison-Wesley, 1997.

[BSL01] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et science informatiques.*, 20(4) :271–311, 2001.

[BSLS01] N. M. N. Bouraqadi-Saâdani, T. Ledoux, and M. Südholt. A reflective infrastructure for coarse-grained strong mobility and its tool-based implementation. Invited presentation at the *International Workshop on "Experiences with reflective systems"* (held in conjunction with Reflection 2001, the *"3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns"*), September 2001.

[CeB] Cebit - world's biggest information & technology and telecommunication trade fair. http ://www.cebit.de.

[Chi00] S. Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP'2000*, LNCS. Springer Verlag, 2000.

[CHN+96] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.-N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, volume 1110 of *Lecture Notes in Computer Science*, pages 54–72. Springer-Verlag, February 1996.

[CKV98] D. Caromel, W. Klauser, and J. Vayssière. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, pages 1043–1061, September–November 1998.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL'96 : The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, FL, USA, January 1996. ACM Press.

[CT98] Shigeru Chiba and Michiaki Tatsubori. Yet another java.lang.class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[DBFM+01] A.M. Dery, M. Blay-Fornarino, S. Moisan, B. Arcier, and L. Mulle. Distributed access knowledge-based system :Reified Interaction Service for Trace and Control. In *3nd International Symposium on Distributed Object Applications (DOA 2001)*, Rome, Italy, September 17-20 2001.

[DBFM02] A.M. Dery, M. Blay-Fornarino, and S. Moisan. Apport des interactions pour la distribution des connaissances. *Numéro spécial de L'OBJET* :, x, 2002. to appear.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex : An optimizing compiler for object-oriented languages. In *OOPSLA'96, Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, CA, USA, October 1996. ACM Press. ACM SIGPLAN Notices, 31(10).

[DHDTJB98] Bruno Dumant, François Horn, Frédéric Dang Tran, and Stefani Jean-Bernard. Jonathan : an open distributed processing in Java. In *Middleware'98 : IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, U.K., September 1998.

[DmYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. Enterprise Java Beans Specification Version 2.0 Proposed Final Draft 2. Sun Microsystems Inc., April 24, 2001.

[Fer89] J. Ferber. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices*, 24(10) :317–326, October 1989.

[FPR98] B. Folliot, I. Piumarta, and F. Riccardi. A dynamically-configurable, multi-language execution platform. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Hac] Hachette. Le dictionnaire universel francophone en ligne. http ://www.francophonie.hachette-livre.fr.

[HHD98] R. Hayton, A. Herbert, and D. Donaldson. Flexinet : A flexible component oriented middleware system. In *Proceedings of ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA'93.* ACM SIGPLAN Notices, 1993.

[IEE01] IEEE, editor. *International Symposium on Principles of Software Evolution*, Kanazawa, JAPAN, February 2001. IEEE Computer Society.

[IMT] International telecommunication union - imt 2000 home page. http ://www.imt-2000.org.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[KEH91] D. Keppel, S.J. Eggers, and R.R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[KG96] J. Kleinoeder and M. Golm. Metajava : An efficient run-time meta architecture for java. Techn. Report TR-I4-96-03, Univ. of Erlangen-Nuernberg, IMMD IV, 1996.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, 2001.

[Kic94] G. Kiczales. Why are black boxes so hard to reuse ? Invited talk at OOPSLA'94 (Portland, Oregon), october 1994.

[Kic96] Gregor Kiczales. Beyond the black box : Open implementation. *IEEE Software*, 13(1), January 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer-Verlag, June 1997.

[Led98] T. Ledoux. *Réflection dans les systèmes répartis : application à CORBA et Smalltalk*. PhD thesis, Université de Nantes, Ecole des mines de Nantes, France, 1998.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, USA*, September 1986.

[Mic] Microsoft. .net. http ://www.microsoft.com/net.

[Mic95] Microsoft Corporation. The Component Object Model Specification, March 1995.

[MPG+00] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes. Runtine support for type-safe dynamic java classes. In

E. Bertino, editor, *Proceedings of the European Conference on Object-oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 337–361, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, Vol. 26(1) :70–93, january 2000.

[OB99] A. Oliva and L. E. Buzato. The design and implementation of Guaraná. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 203–216. The USENIX Association, 1999.

[PDF+02] R. Pawlak, L. Duchien, G. Florin, L. Martelli, and L. Seinturier. Une approche pour la programmation répartie : les composants d'aspect. *Numéro spécial de L'OBJET : Coopération dans les systèmes à objets*, 7, 2002. à paraître.

[PS94] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. *Science of Computer Programming*, 23(1) :19–53, 1994.

[PSDF01] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac : A flexible and efficient solution for aspect-oriented programming in java. In *Reflection 2001, LNCS 2192*, pages 1–24, September 2001.

[Ray95] Kerry Raymond. Reference Model of Open Distributed Processing (RM-ODP) : Introduction. Technical report, CRC for Distributed Systems Technology. Centre for Information Technology Research, University of Queensland, Brisbane 4072 Australia, 1995.

[RC00] B. Redmond and V. Cahill. Iguana/J : Towards a dynamic and efficient reflective architecture for Java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, June 2000.

[Riv97] Fred Rivard. *Evolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Ecole des Mines de Nantes, France, June 1997.

[RL98] P. Raverdy and R. Lea. Dart : A distributed adaptive runtime. In *IFIP 23 International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, UK, September 1998.

[SC99] Katia Barbosa Saikoski and Geoff Coulson. Adaptive group in openorb. In *Proceedings of the 6th Doctoral Consortium on*

Advanced Information System Engineering (CAiSE'99 DC), juin 1999.

[SLCM99] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In R. Guerraoui, editor, *ECOOP'99 - Object-Oriented Programming - 13th European Conference*, volume 1648 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999. Springer-Verlag.

[Smi82] Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[StOSSG00] Richard Soley and the OMG Staff Strategy Group. Model driven architecture. White Paper, November 2000. Object Management Group.

[SU95] Randall B. Smith and David Ungar. Programming as an experience : The inspiration for self. In Walter Olthoff, editor, *ECOOP '95, Aarhus, Denmark*, number 952, pages 303–330. Springer-Verlag, 1995.

[SUN01] The Java HotSpot virtual machine. Technical white paper, Sun Microsystems, 2001.

[TBSN01] E. Tanter, N. M. N. Bouraqadi-Saâdani, and J. Noyé. Reflex - towards an open reflective extension of java. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.

[TVJ+01] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001)*, May 2001.

[Web] Web services community - home page. http ://www.webservices.org.

[WS97] Z. Wu and S. Schwiderski. Reflective Java : Making java even more flexible. Technical report, ANSA, Feb 1997. http ://www.ansa.co.uk/ANSATech/97/Primary/193602.pdf.

[WS99] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Pierre Cointe, editor, *Proceedings of the second international conference Reflection'99*, number 1616 in LNCS, pages 2 – 21. Springer, July 1999.