

PROJET RNTL ARCAD\*  
D1.2 - Etude de l'environnement  
d'intégration  
D1.3 - Document d'architecture  
D1.2 - T0+6 : interne - T0+12 : public  
D1.3 - T0+12 : interne - T0+36 : public

Coordonateur : Michel RIVEILL  
Université de Nice / ESSI  
930 route des Colles  
06903 Sophia Antipolis Cedex

avec la participation de M. Blay-Fornarino, D. Caromel  
T. Coupaye et T. Ledoux

12 décembre 2001

### Résumé

Le projet RNTL ARCAD (<http://arcad.essi.fr>) a pour principal objectif de définir un modèle de composants adaptables et poursuit à cette fin deux buts complémentaires :

- définir un modèle permettant l'intégration à des composants métiers de propriétés non fonctionnelles.
- mettre en oeuvre ce modèle en respectant les normes (ou propositions communément acceptées)

Ce document présente une description de la démarche prise par le projet ARCAD et de l'environnement utilisé par les différents membres du projet. Cette démarche commune et la présence d'une même base d'intégration devrait permettre aux membres du projet d'évaluer les différentes propositions afin d'émettre dans un second temps quelques propositions communes.

---

\*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](mailto:Thierry.Coupaye@francetelecom.com)), INRIA (projet Oasis - [Denis.Caromel@inria.fr](mailto:Denis.Caromel@inria.fr)), projet Sardes - [Daniel.Hagimont@inria.fr](mailto:Daniel.Hagimont@inria.fr)), Ecole des Mines de Nantes (équipe OCM - [Thomas.Ledoux@emn.fr](mailto: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](mailto:Anne-Marie.Pinna@unice.fr)). Le coordonateur du projet est [Michel.Riveill@unice.fr](mailto:Michel.Riveill@unice.fr)

# 1 Introduction

Les technologies de la répartition sont parvenues récemment à un bon degré de maturité, notamment avec l'apparition de plates-formes d'exécution réparties conformes aux spécifications COM+[Mica], .Net[Micb, ECM], CORBA[OMGa] ou EJB[VMb]. Néanmoins, ces plates-formes ne satisfont que de manière très partielle les besoins de nombreux applicatifs. Tout au plus, on assiste à des tentatives ponctuelles et morcelées pour étendre ces plates-formes à chaque domaine applicatif rencontré.

On peut certes considérer que des infrastructures logicielles particulières doivent être pensées en relation avec des domaines d'applications précis. Mais, d'une part, demeurent des principes d'organisation communs des infrastructures logicielles exploitables dans ces différents domaines, et, d'autre part, une tendance lourde des recherches actuelles porte justement sur la conception et la construction d'infrastructures adaptables, c'est-à-dire susceptibles d'être adaptées à des conditions opératoires et à des domaines d'application différents.

La technologie à composants est en passe de s'imposer pour la construction et la mise en oeuvre de grandes applications réparties. Aujourd'hui, les environnements d'exécution pour composants comportent des "structures d'accueil" (conteneur [VMb], [OMGb]) qui fournissent à un ensemble de modules logiciels des services communs permettant le déploiement et l'exécution d'une application.

L'objectif principal du projet ARCAD est de proposer un environnement d'exécution pour composants qui soit extensible afin de pouvoir être complété pour prendre en compte des propriétés non-fonctionnelles non prévues lors de son activation. Les propriétés non-fonctionnelles seront mises en oeuvre sous la forme de composants techniques. L'environnement d'exécution doit en particulier permettre le déploiement des composants d'une application répartie, puis la modification dynamique des configurations par l'ajout aux composants métiers en cours d'exécution de nouveaux services techniques ajoutés ultérieurement. Notre objectif est de pouvoir installer une application y compris sur des sous-systèmes pour lesquels une telle installation n'avait pas été prévue à l'origine.

Les différents composants de l'application rendus adaptables doivent pouvoir modifier leur comportement en fonction des modifications de leur environnement (modification des caractéristiques physiques de l'environnement : ajout de station, modification du débit réseau, etc. mais aussi modification logique : ajout/suppression de services, ajout/suppression de composants logiciels, modification de propriétés non fonctionnelles).

Ce domaine de l'adaptabilité de la plate-forme, des composants métiers par l'ajout et l'intégration des propriétés non-fonctionnelles est vaste et peu formalisé. De nombreuses propositions existent mais aucune évaluation précise n'a été réalisée. Le projet ARCAD qui regroupe des équipes travaillant

depuis de nombreuses années sur ces domaines, avec des approches différentes, souhaite avant tout évaluer de manière précise les différentes solutions qu'ils ont envisagées.

Pour cela à partir d'une base d'expérimentation externe au projet, respectant les standards normatifs actuels notre proposition est de réaliser et mettre en oeuvre pour le même scénario applicatif différents prototypes permettant de mettre en évidence les principaux atouts mais aussi les manques des différentes approches.

La base d'expérimentation choisie est JOnAS[Evi] disponible au sein du consortium ObjectWeb[Obj]. Cette base respecte la spécification EJB[VMb] qui met en évidence un modèle de composants simple, impose différents propriétés non-fonctionnelles (persistance, transaction et sécurité) et propose un modèle simple de composants. En complément à ce modèle et cette base de code, nous souhaitons utiliser une application pilote à spécifier mettant en oeuvre : des composants d'interfaces, des composants applicatifs et pouvant utiliser de nouvelles propriétés non-fonctionnelles.

A partir de cette base nous souhaitons étendre (en respectant les éléments déjà définis) le modèle de composants, définir sous la forme de composants techniques de nouvelles propriétés non-fonctionnelles pouvant être associées aux composants métiers et expérimenter plusieurs modèles d'associations entre composants métiers et techniques. Notre objectif est de proposer ultérieurement un modèle de composants adaptables et d'en valider une ou plusieurs implémentations en fonction des résultats obtenus par comparaison.

Le plan adopté dans ce document permet de présenter successivement le modèle d'application utilisé par le projet pour l'intégration des composants ; les principales propriétés non-fonctionnelles étudiées ; la démarche d'expérimentation mise en oeuvre dans le cadre du projet et les différentes expérimentations projetées.

## 2 Modèle de composants

Après avoir présenté le modèle EJB qui sera la base de nos expérimentations, cette section présente le modèle de composants qui sera utilisé dans une première étape par le projet ARCAD.

### 2.0.1 Spécification EJB : status et perspectives

La spécification EJB a été définie par Sun dans le cadre du processus de spécification de la technologie Java pour les applications critiques de l'entreprise. Elle évolue maintenant dans le cadre du JCP ("Java Community Process"). Ce processus a comme objectif de trouver un consensus autour des spécifications et de les faire évoluer. Sun réalise aussi des suites de tests

de compatibilité et une version de référence, pour les spécifications J2EE (Java 2 Enterprise Edition [Micc]).

Le but principal de la spécification EJB est de faciliter et normaliser le développement, l'assemblage et le déploiement des composants EJB sur des plates-formes conformes à la spécification. Elle fournit deux types d'informations :

1. L'architecture de l'environnement pour l'exécution des EJB (la définition du serveur EJB et du conteneur EJB ainsi que des services fournis par l'environnement d'exécution aux composants EJB. Les principaux services sont : les transactions, le support pour la désignation et la distribution des objets, le support pour la persistance et la sécurité. Ces services se présentent aux composants EJB comme des ressources fournies par le serveur EJB. Le conteneur est la matérialisation d'une couche architecturale qui s'interpose entre chaque composant et le serveur afin d'assurer l'indépendance du composant vis-à-vis de l'implémentation particulière du serveur. Il s'interpose aussi entre le client et le composant, ce qui permet l'accès transparent aux ressources, lorsque leur utilisation n'est pas programmée mais déclarative.
2. Un guide de programmation et d'utilisation des EJB. La programmation des composants EJB est indépendante du système, et même de la plate-forme EJB sur laquelle le composant est déployé. Un composant EJB comporte :
  - une interface définissant les fonctions du composant,
  - une interface contenant les opérations de création, suppression et recherche,
  - une classe qui réalise les fonctions du composant ainsi que des méthodes spécifiques imposées par la spécification EJB, et qui sont nécessaires à la gestion du cycle de vie du composant au sein d'un conteneur EJB
  - un descripteur de déploiement qui décrit le composant et définit ses besoins en terme de ressources nécessaires à son exécution.

## 2.1 Architecture du modèle EJB

Selon le modèle EJB, une application distribuée est constituée de composants (ou Beans), qui sont des objets respectant une interface. A l'exécution, ces composants sont encapsulés dans des conteneurs, eux-mêmes inclus dans des serveurs EJB. Conteneurs et serveurs font partie de la plate-forme intergicielle :

- un serveur est un environnement d'exécution pour ses conteneurs. Un serveur se présente généralement sous la forme d'un processus, et il fournit des services systèmes pour ses conteneurs, comme par exemple la persistance, le mode transactionnel, ou l'authentification ;
- un conteneur encapsule un ou plusieurs composants, et gère les proprié-

tés non-fonctionnelles de ces composants. Par exemple, les propriétés non-fonctionnelles ci-dessus.

Plus précisément, un conteneur est constitué, au minimum, des objets suivants, accessibles à distance :

- l’usine à composants, ou EJB Home, permet de créer de nouveaux composants à l’intérieur du conteneur, de retrouver des composants existants, et de détruire des composants. Il y a une usine à composants par conteneur et par type de composant ;
- les objets d’interposition, ou EJB Objects, permettent d’accéder aux composants encapsulés à l’intérieur du conteneur. En effet, ces composants ne sont jamais accédés directement depuis l’extérieur, mais toujours en passant par un objet d’interposition (d’où son nom). Il y a un objet d’interposition par composant encapsulé (sauf éventuellement pour les composants dit “sans état”, ou stateless bean).

Un objet d’interposition a avant tout un rôle équivalent à celui d’un squelette dans CORBA : il dépaquette les messages d’invocation à distance reçus sur le réseau, invoque la méthode correspondante du composant encapsulé, empaquette le résultat de cet appel, puis le retourne dans la de réponse. Cependant, et c’est là qu’il se différencie d’un squelette CORBA, un objet d’interposition peut effectuer certaines opérations juste avant et juste après avoir invoqué la méthode sur le composant à la manière des intercepteurs introduit dans la nouvelle proposition CORBA 3. Cette possibilité permet de l’assimiler à un encapsulateur (wrapper) du Bean associé.

Le modèle EJB prévoit que le code des objets d’interposition, ainsi que celui des usines à composants, soit généré automatiquement à partir d’un fichier de déploiement. Ce fichier de déploiement, écrit par l’utilisateur en fonction de ses besoins, avant de déployer son application, indique par exemple que pour la méthode `transfer` des composants de type `Account` il faut être dans le cadre d’une transaction, alors que pour la méthode `getBalance` ce n’est pas nécessaire. Ce fichier est utilisé lors du déploiement par un générateur de code, pour produire la classe des objets d’interposition. Dans l’exemple précédent, le code produit pour la méthode `getBalance` de l’objet d’interposition se contenterait d’appeler la méthode `getBalance` sur le composant, alors que le code produit pour la méthode `transfer` ferait des traitements supplémentaires pour gérer les transactions.

Le modèle EJB est actuellement utilisé pour prendre en chare en dehors du code fonctionnel le code de trois propriétés non-fonctionnelles : persistance, transactions et protection. Comme nous l’avons dit, ce sont les objets d’interposition qui permettent cette séparation. L’algorithme qu’ils utilisent pour traiter un appel de méthode à distance est approximativement le suivant :

- vérifier si l’appelant a bien le droit d’appeler cette méthode, pour un composant protégé ;
- démarrer une transaction si la méthode doit s’exécuter dans un contexte

- transactionnel et qu'aucune transaction n'est en cours ;
- charger le composant appelé en mémoire, s'il ne s'y trouve pas déjà (pour un composant persistant) ;
- appeler la méthode sur le composant ;
- enregistrer le composant sur support persistant (pour un composant persistant) ;
- terminer la transaction en cours si celle-ci a été démarrée par l'objet d'interposition lui-même.

## 2.2 Modèle de composants du projet ARCAD

L'objectif du projet ARCAD n'est pas de proposer un nouveau modèle de composants, ni de limiter l'impact des résultats du projet au seul EJB. En particulier, les plates-formes EJB actuelles utilisent des objets d'interposition (i.e des méta-objets) performants mais monolithiques, ce qui limite beaucoup l'extensibilité des plates-formes EJB (pour gérer une propriété technique nouvelle, il faut réécrire l'objet d'interposition, lui-même généralement généré à partir d'un compilateur qu'il faudrait alors modifier). Cette action éventuelle doit impliquer une bonne compréhension de la combinaison des services existants pour les combiner avec celui à intégrer. Aujourd'hui, avec le modèle EJB, le concepteur d'application, ne connaît que les composants métiers auxquels, il peut ajouter des services techniques précédemment existants :

- les composants techniques n'ont pas d'existence propre, seul trois services sont proposés : persistance, sécurité et transaction ;
- les mécanismes de fusion ne sont pas ouverts et accessibles aux concepteurs d'applications.

Nous prenons comme base de travail un modèle de calcul très général, inspiré du modèle calculatoire d'ODP [ODP95b, ODP95a], qui englobe tous les autres et en particulier le modèle EJB et le modèle CCM (Corba Component Model[OMGb]) de l'OMG. Ce modèle de programmation utilise les concepts de composants, d'interfaces et de connecteurs pour structurer les applications. Ces concepts sont définis ci-dessous.

Un *composant* encapsule du code et des données, comme un objet. Cependant, contrairement à un objet, un composant peut avoir plusieurs interfaces d'entrées et de sorties. Cette définition du terme "composant" est très minimaliste, et peut donc facilement être raffinée, si besoin est, pour correspondre à des définitions plus précises, comme celle de la norme CORBA 3.0 [CCM99].

Une *interface* est un point d'accès à un composant. Une interface a un type, défini par un ensemble de signatures de méthodes, ainsi qu'un sens : une interface d'entrée permet d'appeler des méthodes dans un composant, alors qu'une interface de sortie décrit les appels de méthodes vers d'autres composants.

Le rôle d'un *connecteur* est de permettre aux composants de communiquer entre eux par l'intermédiaire de leurs interfaces. Un connecteur peut représenter n'importe quel type d'interaction entre composants : client-serveur, publication-abonnement...

En complément à ce modèle, le projet ARCAD, fait a priori la supposition que chaque service technique est un composant au même titre que les autres composants métiers de l'application ou de l'intergiciel sous-jacent. De plus, les mécanismes de fusion permettant de lier les composants métiers et techniques doivent pouvoir être accessibles aux concepteurs d'application.

Cette fusion peut servir à combiner un composant applicatif avec les différents composants représentant les services non-fonctionnels utilisés. Il est par exemple possible d'utiliser les techniques de composition de méta-objets, c'est à dire remplacer les objets d'interposition par des listes d'objets plus simples, ne gérant qu'une seule propriété à la fois, et générés chacun par leur propre générateur de code.

### **3 Composants techniques**

Cette section décrit les différents composants techniques qui seront abordés dans le cadre du projet ARCAD.

#### **3.1 Les composants techniques de la proposition EJB**

Nous souhaitons reprendre sous la forme de composants techniques indépendants, les propriétés non-fonctionnelles présentes dans la proposition EJB 2.0 (cycle de vie, persistance, transaction et sécurité). L'objectif de cet effort est de pouvoir construire à terme un intergiciel étendant les plateformes EJB. Cet intergiciel supportera au moins les mêmes services et sera étendu par les services décrits dans les sous-sections suivantes.

Pour ces services, notre objectif est de définir la ou les interfaces du composant technique, d'en proposer une ou plusieurs implémentations et de fournir un outil permettant la composition du composant métier avec les composants techniques nécessaires à son exécution.

#### **3.2 Communication asynchrone, synchronisation par futur**

Les infrastructures habituelles offrent pour la plupart une communication synchrone. Une communication asynchrone permet bien souvent une plus grande indépendance entre les composants, avec en particulier une meilleure résistance aux inter-blocages. Nous souhaitons donner la possibilité de désynchroniser facilement les communications entre deux composants, et donc il nous semble nécessaire d'avoir une telle propriété. Par contre, les communications asynchrones souffrent d'une manière évidente d'un manque de synchronisation, en particulier en cas de dépendances fonctionnelles entre com-

posants. Pour cette raison, la propriété de communication asynchrone pourra être couplée avec la possibilité d'avoir une synchronisation par futurs entre composants.

### **3.3 Communication en mode déconnecté**

Cette propriété prolonge d'une certaine manière la précédente. Dans le cas d'une communication asynchrone, on peut demander aux deux composants d'être tous deux activés et connectés au moment de la communication ; cela permet entre autre de garantir certaines propriétés liées par exemple à la cohérence des données. Mais on peut, et dans certains cas c'est une contrainte forte de l'application, vouloir permettre une communication dite en "Mode déconnecté". Il n'est plus alors nécessaire que les deux composants soient connectés au même instant. Notons que dans ce cas, les synchronisations par futur sont également un aspect pertinent et fort utile. Cette propriété est par exemple nécessaire dès que l'on envisage qu'une application puisse s'exécuter sur un équipement nomade : carte à puce, assistant personnel ou station de travail.

### **3.4 Communication de groupe**

La capacité à participer à une communication de groupe est une propriété importante dans le cadre de composants devant servir à la construction d'applications collaboratives, de commerce électronique, ou encore de services télécoms. Il est donc important de définir les modalités d'une telle communication comme un aspect non-fonctionnel configurable, et d'en implémenter les primitives nécessaires. Cette propriété servira de support pour la gestion de données dupliquées (réplication).

### **3.5 Migration**

Les techniques de migration et de mobilité sont une des bases permettant aux applications de s'adapter aux changements de localisation de ses différents composants et de se reconfigurer. La migration peut se faire au moyen de la mobilité de code, de calculs, ou de données. La mobilité des données, associée à des techniques de gestion de caches répartis, permet à la fois de diminuer la latence d'accès aux informations et de modifier dynamiquement l'environnement d'exécution d'une application pour répondre à des besoins changeants. La mobilité du code et de calculs permet par exemple de déplacer dynamiquement l'exécution d'un processus client vers un serveur de données pour remédier à la variabilité des performances d'un réseau, ou à une déconnexion. L'aspect migration de composants est donc une propriété non-fonctionnelle fondamentale. Il faudra donc définir les caractéristiques et contraintes nécessaires pour permettre la migration d'un composant, et définir une ou plusieurs sémantiques de migration (faible, forte, etc.).

### **3.6 Création, placement dynamique, et accès à distance**

Un composant est habituellement créé et placé de façon statique dans un conteneur. Avec les nouvelles techniques de programmation d'architecture répartie mises en oeuvre dans ce projet, et en particulier les aspects réflexifs, il devrait être possible de créer dynamiquement des composants à partir d'objets Java standard, de les rendre accessibles à distance par d'autres composants pré-existants, voire même de les placer dynamiquement dans d'autres conteneurs.

### **3.7 Régulation de charge**

Il nous semble nécessaire de pouvoir offrir à terme différents mécanismes permettant de réguler la charge entre différentes stations. La construction de ces outils repose généralement sur d'autres propriétés non fonctionnelles comme le placement dynamique, la migration, la duplication. L'étude de cette propriété non fonctionnelle construite pour partie à l'aide d'autres propriétés plus élémentaires permettra d'étudier plus précisément la composition de composants techniques.

### **3.8 Réplication, Disponibilité**

On peut répliquer des objets et des traitements soit pour répartir la charge entre différentes stations, soit pour assurer la disponibilité du service. Inévitablement de la mise en oeuvre de composants techniques pour la réplication, l'étude de cette propriété non fonctionnelle nous semble en elle-même intéressante dans le cadre des EJB par l'adhérence qui existe entre les composants métiers et les différents supports de la persistance.

### **3.9 Sécurité**

Les techniques utilisant la mobilité du code et des calculs impliquent l'exécution sur un serveur de programmes provenant d'une autre machine. En l'absence de mesures de protection appropriées, une telle exécution présente un danger potentiel. Des aspects sécurité doivent donc être pris en compte. Définir la sécurité comme une propriété non-fonctionnelle devrait permettre une configuration dynamique de cet aspect qui est absolument nécessaire en présence de migration. En effet, lorsque des composants se déplacent d'un domaine administratif à un autre, les sécurités à utiliser lors des communications changent. Il est donc impératif d'étendre le modèle de sécurité aujourd'hui disponible dans les EJB afin de pouvoir l'adapter dynamiquement aux caractéristiques de la communication point à point entre chaque paire de composants (authentification, cryptage, etc.).

## 4 Expérimentations

Notre soucis d'expérimentation en commun, nous amène à posséder une plate-forme d'expérimentation capable d'évoluer et d'être facilement portable dans les différentes équipes ce qui nous a conduit à choisir une plate-forme dont nous pourrions avoir les sources en Java. Les sources sont nécessaires pour pouvoir faire évoluer en profondeur la plate-forme et Java offre une portabilité accrue par rapport à d'autres langages de programmation.

Parmi les différents plates-formes offrant ces deux caractéristiques, deux modèles restent en concurrence : les implémentations OpenSource de la spécification EJB et celles de la spécification CCM[OMGb]. Pour des raisons liées à la taille de la plate-forme mais aussi par la disponibilité de différents services techniques déjà intégrés, il nous a semblé intéressant de pouvoir travailler sur une implémentation de la spécification EJB. Notre choix s'est porté sur JOnAS[Evi], plate-forme réalisée par Evidian (filiale de Bull) et disponible par l'intermédiaire du consortium ObjectWeb[Obj].

La petite taille relative de la plate-forme initiale, nous permet de pouvoir véritablement l'utiliser comme plate-forme d'expérimentation et proposer différentes extensions matérialisant les différentes voies d'approche afin de pouvoir ultérieurement en faire une étude comparative. Les services techniques disponibles nous permettent de travailler directement sur la fusion de services, fusion qui n'est pas spécifiée dans la spécification EJB.

Une fois cette première étude menée dans le cadre des EJB, nous souhaiterions l'étendre afin de l'intégrer au modèle des CCM, modèle de composants plus complexes mais aussi plus riche, à travers la plate-forme OpenCCM[RM] réalisée par le LIFL (Laboratoire d'Informatique Fondamentale de Lille) elle aussi disponible dans le consortium ObjectWeb.

A partir des différentes contraintes précédentes, communes pour chacun des membres du consortium, nous souhaitons expérimenter les différentes approches prises dans chacune des équipes. Cette expérimentation doit conduire aux résultats suivants :

- a) *Compléter le modèle de composants initial* et proposer une description de composants techniques. Il nous semble souhaitable que les composants techniques possèdent une description permettant non seulement de connaître les différents fonctionnalités qu'ils offrent (description absente dans le modèle EJB mais présente dans le modèle CORBA, COM+ ou .Net sous la forme d'une description dans un langage d'interface à la "IDL") mais aussi de décrire dans un formalisme approprié la manière de l'utiliser. Aujourd'hui la lecture d'un composant ne permet que d'en connaître son interface, il nous semble souhaitable d'y inclure une description comportementale.
- b) *Définir différents composants techniques*. Une première étape consistera à reprendre les services disponibles dans la plate-forme JOnAS : cycle de vie, persistance, transaction et sécurité. Une seconde étape

- consistera à intégrer au sein de cette plate-forme des services définis par les différents membres du consortium : composants mobiles, connecteurs permettant la communication par futur, connecteurs gérant le mode deconnecté, composants répliqués, composants disponibles, etc.
- c) *Fusionner un composant métier avec différents composants techniques.*  
Le véritable enjeu du projet se situe à ce niveau. En effet, la description de composants techniques aussi sophistiqués fussent-ils est incomplète si l'on ne connaît pas la manière dont ils fonctionnent (aspect détaillé dans a) et la manière dont ils doivent coopérer avec le composant métier et les autres composants applicatifs.
  - d) Evaluer de manière comparative les différentes propositions. L'objectif de cette évaluation est multiple car nous souhaitons que les techniques utilisées puissent s'adapter à de multiples niveaux : au niveau de l'intergiciel, des applications mais aussi des outils de monitoring en considérant que chacun de ces niveaux peut être décrit sous la forme d'un assemblage de composants métiers qui peuvent être adaptés par fusion de composants techniques.

La suite de cette section présente rapidement la plate-forme JOnAS et les différentes approches qui seront prises dans le cadre du projet.

#### 4.1 JOnAS

JOnAS fournit une implémentation 100 % Java de la spécification EJB et offre le support des deux catégories de composants EJB définis par la spécification, les composants de type session et ceux de type entité.

JOnAS est construit au dessus d'un moniteur transactionnel Java qui permet la gestion des transactions distribuées. La sémantique transactionnelle associée au composant entité (EB : Entity Beans) peut être définie de manière déclarative via des attributs transactionnels définis au niveau du descripteurs de déploiement. Toutes les valeurs possibles définies par la spécification EJB pour ces attributs transactionnels sont supportées par JOnAS.

Le support des composants entité est basé sur l'utilisation de l'interface JDBC[Kon98], conformément à la spécification EJB : le support de la persistance automatique (i.e. assurée par le conteneur de manière transparente) est fourni avec certaines restrictions, imposées par les drivers JDBC existants et par des difficultés liées à la correspondance entre le modèle objet de Java et le modèle relationnel.

La distribution est basée sur le mécanisme de distribution RMI[Sun] et sur JNDI[MH98], l'interface standard de désignation et d'accès aux répertoires.

JOnAS fournit en plus du moniteur transactionnel Java et d'un ensemble de classes implémentant la spécification EJB, des outils permettant la génération de classes d'interpositions qui constituent le conteneur EJB ainsi que des classes qui représentent la matérialisation des informations fournies dans

le descripteur de déploiement.

JOnAS a déjà été téléchargé par plus de 40 000 personnes et est utilisé par des centaines d'entre elles, y compris pour des applications actuellement opérationnelles.

## 4.2 Approche commune

L'approche prise par le projet est de considérer la spécification EJB et son implémentation JOnAS comme une base d'expérimentation sur laquelle nous pouvons intervenir à différents niveaux.

Le projet ARCAD souhaite expérimenter en parallèle, plusieurs approches permettant d'ouvrir le mécanisme de fusion et de considérer les propriétés non-fonctionnelles comme des composants.

Aujourd'hui l'intégration dans JOnAS des propriétés non-fonctionnelle est faite par le compilateur GenIC à partir de la description du bean et des propriétés souhaitées décrites dans un fichier de déploiement. Mais plusieurs voies d'approche alternatives seront adoptée par les partenaires du projet :

- proposer un générateur capable de réifier l'objet d'interposition afin qu'il fasse appel à des méta-objets et encapsuler dans ces méta-objets les différents services techniques souhaités (OCM-EMN)
- construire statiquement un objet d'interposition capable de réifier les appels de méthodes (OASIS-INRIA)
- insérer dans l'objet d'interposition le mécanisme d'interaction et décrire l'appel aux nouveaux services sous la forme d'interaction (RAINBOW-I3S)
- utiliser des techniques de compilation pour adapter de manière efficace un composant métier (SARDES-INRIA)
- composer statiquement et dynamiquement des comportements décrit sous la forme d'automates (ASR-France Télécom R&D)

Pour expérimenter ces différentes voies, devant permettra l'extensibilité et l'adaptabilité statique ou dynamique de la plate-forme et des composants métiers, les différents partenaires du projet procéderont par étapes :

1. Exprimer sous la forme de composants les différents services techniques qu'il serait souhaitable d'intégrer :
  - mobilité, communication par futur (OASIS-INRIA)
  - réplication en vue d'obtenir la disponibilité (RAINBOW-I3S)
  - réplication en vue de permettre le mode déconnecté, protection (SARDES-INRIA)
2. Associer aux différents composants techniques sa spécification d'intégration afin de permettre ultérieurement les associer aux composants métiers :
  - interaction, automate (RAINBOW-I3S)
  - langage de spécification d'intégration (SARDES-INRIA)

- automate (ASR-France Télécom R&D)
- 3. Adapter par extension le composant métier en intégrant statiquement ou dynamiquement, un ou plusieurs composants techniques, éventuellement par reconstruction d'un objet d'interposition (conteneur) :
  - Mécanismes de fusion ouverts basés sur la réflexion et AOP (OCM-EMN)
  - M.O.P (OASIS-INRIA)
  - fusion d'interactions (RAINBOW-I3S)
  - compilation et chargeur adaptable (SARDES-INRIA)
  - composition d'automates (ASR-France Télécom R&D)

La suite de ce document présente de manière plus détaillée les différentes expérimentation qui seront menées.

### 4.3 Infrastructure pour la décision et la réalisation de l'adaptation - EMN / OCM

L'objectif du projet ARCAD est de proposer un environnement extensible pour composants adaptables. L'environnement doit en particulier permettre l'*adaptation* dynamique des applications par ajout de nouveaux services techniques aux composants métiers en cours d'exécution.

Une adaptation peut se décomposer en trois étapes successives :

- *Notification*. L'étape de notification implique tout ce qui caractérise l'origine de l'adaptation. Cette notification peut être effectuée par différents acteurs (logiciel ou administrateur humain) et à différents moments (déploiement, exécution).
- *Décision*. L'étape de décision a trait à tous les choix d'adaptation qui vont être pris. Ces choix peuvent être effectués par différents acteurs, concerner différents sujets à adapter (composants métiers, composants techniques, leur association) 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 l'adaptation. La réalisation peut être effectuée par différents acteurs et selon différents mécanismes (objets d'interposition, fusion de code). La réalisation s'opère sur les sujets à adapter qui ont été définis dans l'étape de décision.

Dans ce cadre, les travaux de recherche de l'équipe OCM de l'EMN porteront principalement sur ces deux derniers points, à savoir les étapes de décision et de réalisation de l'adaptation.

#### 4.3.1 Décision : proposition d'une infrastructure pour l'adaptabilité dynamique et application à JOnAS.

**Infrastructure pour l'adaptabilité.** Le processus de décision va consister à décider quels sont les changements significatifs du point de vue de

l'application, et quelles modifications du programme doivent être effectuées en réaction à ces changements. Notre objectif ici est de dégager (par la réflexion et l'expérimentation) les différents concepts nécessaires et suffisants pour programmer n'importe quel type de décisions d'adaptation, puis d'encapsuler ces concepts dans un ou plusieurs langages spécialisés (DSLs) fournissant un haut degré d'abstraction et une approche déclarative pour faciliter leur utilisation par les différents acteurs. Ces stratégies d'adaptation pourront par exemple être utilisées par le déployeur ou par l'intergiciel lui-même pendant l'exécution.

**Une première expérimentation.** Un premier prototype nous a permis de valider une approche basée sur des *politiques d'adaptation* exprimées dans un langage spécialisé rudimentaire écrit en XML [Dav01]. Ces politiques sont ensuite interprétées dynamiquement par un moteur d'adaptation (réalisé en Java) qui modifie les associations composants métiers/composants techniques de l'application. La solution proposée est basée sur le protocole à métaobjets RAM [BSLS01] de type *run-time* permettant de séparer le code métier des services techniques et de modifier dynamiquement leurs associations.

**Application à JOnAS.** Dans notre premier prototype, la séparation des préoccupations (*separation of concerns*) est réalisée par le MOP RAM où chaque méta-objet implémente un service technique particulier. Le couplage lâche entre le moteur d'adaptation, les composants métiers et les services techniques permet d'envisager la réutilisation des politiques d'adaptation avec un MOP différent de RAM ou même avec un serveur EJB [VMb] où le conteneur jouerait le rôle d'assembleur dynamique.

Nous menons actuellement une expérimentation dans ce sens avec le serveur EJB JOnAS [Evi]. Les politiques sont utilisées pour configurer les associations entre les différents composants métiers et les services en indiquant à JOnAS d'une part quelles sont les modifications de l'environnement à prendre en compte, et d'autre part comment réagir à ces modifications de façon appropriée.

Dans un premier temps, nous avons défini nos propres services « jouets »<sup>1</sup> pour nous concentrer sur les mécanismes d'adaptation. L'idée de base est articulée autour de deux étapes distinctes. La première consiste à introduire une indirection au niveau de l'objet d'interposition généré par GenIC ; alors que la seconde, a pour but d'intégrer l'infrastructure d'observation et le moteur d'adaptation des politiques réalisé au cours du travail cité en [Dav01].

---

<sup>1</sup>La réutilisation des services préfinis de JOnAS fera l'objet de nos futurs travaux.

### 4.3.2 Réalisation : étude des mécanismes d'assemblage

L'objectif de cette thématique de recherche est de travailler sur les problématiques de fusion/assemblage des composants métiers et des composants techniques. Notre première tâche consiste à concevoir un modèle de composants suffisamment expressif pour envisager plusieurs types d'assemblage (objets d'interposition, fusion de code) à des moments différents (développement, déploiement, exécution). Entre autres, si la priorité n'est plus l'adaptabilité mais la performance, il devra être possible de modifier dynamiquement la granularité des composants par le biais d'un mécanisme de fusion de composants (*inlining* du code technique avec le code métier).

Plusieurs sources d'inspiration (non exclusives) sont possibles concernant le type de modules manipulés à ce niveau : éléments architecturaux inspirés des ADLs (composants et connecteurs), objets et métaobjets (réflexion), aspects (AOP), ...

**Proposition d'un tissage retardé.** Une étude des relations entre AOP et la réflexion et plus particulièrement entre AspectJ [KHH<sup>+</sup>01] et notre MOP RAM [BSLS01], nous a permis de faire une proposition de tissage retardé (*late weaving*) [DLBS01].

L'idée a été de prendre « le meilleur » des deux mondes, à savoir la réification des aspects/composants techniques à l'exécution (RAM) et le modèle puissant des points de jonction (AspectJ) pour réaliser un tissage assez fin des composants techniques avec les composants métiers et ce à l'exécution.

## 4.4 Objet d'interposition - INRIA/OASIS

L'utilisation d'objets d'interposition est une technique utilisée fréquemment pour l'implémentation de protocole à méta objets. En effet, elle offre un excellent compromis entre souplesse et expressivité d'une part, facilité d'implémentation et portabilité d'autre part.

En partant du principe que, dans un langage à objets, l'aspect dynamique d'un programme est entièrement exprimé par l'échange de messages entre objets, la technique des objets d'interposition propose de placer, entre l'objet appelant et l'objet appelé, un objet d'interception qui a pour tâche d'intercepter les messages envoyés. Cet objet a un comportement préemptif : les messages interceptés ne sont effectivement transmis à l'objet appelé que si l'objet d'interception le désire, et il n'est pas possible à l'objet appelant de passer outre l'interception des messages.

Par ailleurs, l'objet d'interception est conçu pour avoir un type compatible avec celui de l'objet destinataire des messages, de manière à ce que sa présence soit invisible à l'objet appelant. L'utilisation d'objets d'interposition est donc une technique de base pour effectuer l'interception d'appels de méthodes de manière transparente. Dans la littérature de l'ingénierie logi-

cielle, les objets d'interception apparaissent sous la forme du Design Pattern *Proxy*. Les objets d'interposition sont également très présents dans l'implémentation des middlewares à objets, ils sont connus dans ce contexte sous le nom de *stubs*.

Par rapport à d'autres techniques d'interception des appels de méthodes, la technique des objets d'interposition présente plusieurs avantages. Tout d'abord, elle est non-intrusive puisqu'elle ne fait appel qu'à un objet standard et ne repose que sur les mécanismes existants, en particulier sur le polymorphisme et la liaison dynamique. Dans le cas de Java, cette technique est portable sur toutes les implémentations de la machine virtuelle Java, ce qui n'est pas le cas de la plupart des autres techniques d'interception d'appels de méthodes. Ensuite, ces techniques sont relativement faciles à implémenter, soit par génération de code (génération statique, au moment de la compilation) ou de bytecode (à l'exécution, au moyen d'un chargeur de classes spécialisé), ou même, dans certains cas, depuis le JDK 1.3, en utilisant la technique des *dynamic proxies* qui fait désormais partie de la distribution standard de Java. Enfin, cette approche permet d'établir une séparation claire, au niveau de granularité des objets et même des classes, entre l'appelant, l'appelé, le mécanisme d'interception de l'appel et le traitement appliqué à l'appel de méthode intercepté.

#### 4.5 Composition de comportements - FT/ASR

Un composant, outre les interfaces qu'il propose et donc les fonctionnalités qu'il exhibe affiche un *comportement* propre. Dans les modèles de composants existants, et plus généralement dans les différentes techniques de composition logicielle, ce comportement est implicite, i.e. supposé connu par les composants qui utilisent un composant donné. Le comportement des composants étant arbitrairement complexe – c'est le cas des composants techniques tels que des services de persistance ou de réplication par exemple – il est impossible de raisonner sur la validité d'une composition, et même plus simplement, il est très difficile de composer correctement des composants au comportement complexe puisque ce comportement n'est pas exprimé. C'est notamment là que les techniques de "séparation de problèmes" (separation of concerns) tel que la programmation par aspects[Kic96] (Xerox), la programmation par sujets[OHBS94] (IBM), les filtres de composition[ABV92] (U. Twente), etc., atteignent leurs limites car elle ne permettent pas de modéliser assez finement les interactions entre des composants multiples interagissant à des mêmes interfaces.

Le travail proposé ici, que nous appelons *composition comportementale*, consiste à décrire explicitement le comportement des composants, à adjoindre cette description à la définition des composants (interfaces fournies et requises, etc.), et finalement à composer des composants en composant leur comportement. Ce comportement peut être représenté, par exemple, par des

automates à états finis. Le problème de la composition de composants peut donc être envisagé comme un problème de composition d'automates. Les langages synchrones [BB] et/ou langages réactifs [Hal98] possèdent des propriétés intéressantes pour une gestion fine de la composition comportementale : déterminisme, réactivité, cohérence temporelle des signaux et réactions. A plus long terme, des techniques de validation du comportement global d'une composition de composants s'appuyant sur la vérification formelle et la simulation pourraient être appliquées pour certifier que la composition résultante est *fiable*.

Les problèmes soulevés par la composition comportementale concernent 1) la modélisation du comportement des composants, 2) la composition de comportements et 3) l'intégration de l'approche proposée dans un modèle de programmation impératif classique. Plus spécifiquement, les problèmes soulevés par la modélisation et l'expression du comportement des composants concernent la généralité de l'approche (est-on capable de trouver un formalisme suffisamment général qui permette de représenter le comportement de tous les composants ?) ; et son utilisabilité (quel formalisme offrir à l'utilisateur final ?). Une modélisation sous forme d'automates est caractérisée par des états (initiaux et finals) et des transitions. Une transition est de la forme  $\langle \text{événement} \rangle \langle \text{condition} \rangle \langle \text{action} \rangle$ . De nombreux types d'événements peuvent être intéressants pour la composition comportementale : les interactions avec des composants, les événements internes aux composants, en particulier ceux liés à leur cycle de vie, les événements provenant de l'environnement système (mémoire insuffisante par exemple), les exceptions, etc. Les conditions portent sur l'état interne des composants et/ou leur contexte d'exécution. Les actions peuvent être des exécutions d'opérations, des modifications des variables de contexte, etc. Il est difficile de déterminer le comportement global du système. Notre objectif est de générer, à partir des spécifications comportementales de chaque composant et des propriétés liées à la composition, le comportement global et déterministe de ce dernier. Pour atteindre cet objectif, une algèbre de composition et une grammaire d'expression des propriétés sont en cours de définition. Ces propriétés vont influencer sur la construction du comportement global dans le sens où elles vont soit ajouter du comportement soit modifier le comportement initial dans les différentes spécifications des composants.

Dans le cadre du projet ARCAD, nous nous intéressons plus particulièrement à la composition de propriétés non fonctionnelles au sein d'une infrastructure logicielle. Nous voyons une application comme un *système* constitué d'un ensemble de couples (*domaine*, *conteneur*). Un *domaine* est un ensemble déterminé de composants applicatifs (un sous-ensemble de l'ensemble des composants applicatifs d'une l'application) auquel on fournit un ensemble déterminé de services techniques, réalisés par des composants techniques. Un *conteneur* est la composition d'un ensemble déterminé de composants techniques. Un conteneur est attaché à un domaine, il lui fournit un en-

semble de services techniques. Un conteneur peut être vu comme un système réactif aux opérations effectuées par les composants applicatifs. Pour cela, on modélise, pour chaque composant technique, d'une part, le comportement propre de ce composant technique et, d'autre part, le comportement des composants applicatifs vis-à-vis de ce composant technique. L'assemblage de composants techniques au sein d'un conteneur correspond alors à la composition des différents automates qui modélisent le comportement de ce composants techniques et des automates qui modélisent le comportement des composants applicatifs par rapport à chacun des composants techniques.

La composition comportementale peut se décliner de manière statique (configuration) et dynamique (configuration et reconfiguration). Il existe deux types de reconfiguration dynamique : 1) une reconfiguration de conteneur, i.e., une modification du comportement des composants techniques, l'ajout ou le retrait de composants techniques d'un conteneur, etc. ; et 2) une reconfiguration de domaine, i.e., l'ajout, le retrait d'un composant applicatif dans un domaine, etc. Les travaux en cours concernent ces deux types de reconfiguration. Une approche hybride (génération en Esterel [BG92] pour une partie statique, génération en Junior [INR] pour une partie dynamique) est envisagée.

#### 4.6 Interactions - UNSA/Rainbow

Le langage ISL[LB98] (Interaction Specification Language) permet de décrire les interactions entre composants de manière externe. Ce langage, relativement simple, permet de modifier à l'aide de règles de réécriture le comportements métiers des composants. Il est ainsi possible de connecter plusieurs composants à l'aide d'interactions qui sont des comportements réactifs construits à partir d'une dizaine d'opérateurs : opérateur séquentiel, opérateur non ordonné, opérateur conditionnel, opérateur d'exception, etc.

Si un comportement est réécrit dans plusieurs interactions, le nouveau comportement est le résultat de la fusion des comportements réactifs associés. Cette fusion est établie à partir de règles de fusion se basant sur les opérateurs d'ISL. La description d'ISL et la sémantique de la fusion sont données dans [BER01]. ISL est indépendant de tout langage de programmation et de toute technique de communication. Un service d'interactions basé sur ISL est mis en oeuvre dans DICO\*[DIC] pour les langages cibles Java et C++, avec une communication Corba. Dico\* se base sur un dépôt d'interactions qui permet la définition, la pose et le retrait dynamiques d'interactions.

Voici quelques exemples de description ISL :

```
Interaction object&Memoire(ObjectRecoverable O, Memoire M)
O.* -> M.memoriser(O); O._call
O.restore() -> M.restore(O)
```

```

Interaction object&Memoire&Histo(ObjectRecoverable Or, Memoire M, Historique h)
    extends object&Memoire(Or, M)
    Or.* -> M. memoriser(Or); (Or._call //h.note(_call))

Interaction memoriser(Object O, Memoire M, File F)
boolean connected = true;
O.* -> if (connected)
    then try{ m.memoriser(a,_call) ;}
        catch ObjectNonExistent
            { f.open(..); f.write(_call); connected=false} ;
    else { f.write(_call); a._call();
        try {m.memoriser(a,null) ;
            f.close(); m.memoriser(serializer(f)); connected=true;}
        catch ObjectNonExistent {}
    }
}

```

La proposition du projet RAINBOW part de la constatation suivante : ajouter une propriété non fonctionnelle à un composant métier se résume à l'expression et la prise en charge des interactions entre les composants techniques mettant en oeuvre les propriétés non-fonctionnelles souhaitées et le composant métier. Les objectifs de l'équipe sont donc de :

- vérifier si ISL est adapté pour exprimer l'intégration d'un service technique.
- vérifier si les règles de fusion donnent une réponse satisfaisante à la combinaison de services.

Pour cela, les différentes étapes du travail sont les suivantes :

1. intégrer le service d'interactions à Jonas en modifiant le générateur GenIC.
2. reconcevoir l'intégration des trois services standards (persistance, sécurité et transaction) pour en proposer une implémentation sous la forme de composants techniques et proposer leur intégration à l'aide d'ISL et du service d'interactions.
3. intégrer de nouveaux composants techniques (interaction, migration, asynchronisme, gestion des réplicats...) à l'aide des interactions.

Les deux derniers points peuvent mettre en évidence des lacunes dans l'expressivité d'ISL qu'il sera alors nécessaire de compléter en proposant pour chaque introduction d'opérateur les nouvelles règles de fusion qui devront être appliquées. Les règles de fusion actuelles formalisent la combinaison des opérateurs existants dans le cadre de l'interconnexion de composants métiers mais peuvent ne pas être adaptées aux types de combinaisons attendues pour les composants techniques étudiés. Aussi une part essentielle de notre travail est-elle de déterminer quelles sont les extensions indispensables à ISL et de les exprimer en terme d'opérateurs dont la sémantique est claire. La

fusion doit alors être revue pour prendre en charge ces nouveaux opérateurs. Il va peut être également falloir ouvrir le mécanisme de fusion actuel afin de laisser aux utilisateurs de la latitude quant à la façon de combiner les comportements.

Si cette étude aboutie, ISL et les interactions permettront non seulement d'interconnecter des composants métiers mais aussi d'ajouter ou de supprimer de manière dynamique des propriétés non fonctionnelles à des composants métiers. Cet ajout de propriété pourra se faire de manière interne par l'application elle-même qui décide de modifier son comportement en fonction de l'évolution de son environnement ou de manière externe, par l'environnement ou un administrateur qui souhaiterons faire évoluer la partie non-fonctionnelle d'un composant ou d'un ensemble de composants en fonction d'une évolution observée ou attendue.

En complément à cette étude de faisabilité, nous souhaitons en mesurer son impact en terme de confort pour les programmeurs d'applications et de performances (temps CPU et place mémoire) afin d'évaluer le gain d'une solution dynamique à l'aide du service d'interactions par rapport aux besoins réels d'une application devant utiliser des composants adaptables.

#### **4.7 Noyau de système adaptable - INRIA/SARDES**

- Le projet ARCAD étudie principalement les trois approches suivantes :
- Prise en compte des aspects non-fonctionnels. Il est proposé de les mettre en oeuvre par adaptation de l'objet d'interposition dans l'infrastructure EJB.
  - Exprimer les aspects non-fonctionnels qu'il est souhaitable de traiter et les intégrer dans les EJB. Notamment la mobilité, la duplication pour la disponibilité et le mode déconnecté sont visés.
  - Permettre la fusion entre aspects ou la fusion entre aspect et code métier.

La contribution du projet Sardes portera essentiellement sur les deux premiers axes de travail.

##### **4.7.1 Prise en compte des aspects non-fonctionnels**

La solution classique pour prendre en compte des aspects non-fonctionnels est de les insérer dans une structure d'exécution au niveau d'objets intermédiaires. Ces objets sont souvent appelés objets d'interposition (EJB), intercepteurs ou encore proxy. Dans le projet Sardes, nous avons travaillé de longue date sur l'intégration (transparente) d'aspects non-fonctionnels dans une infrastructure d'objets répartis, en utilisant des objets d'interposition. Ces travaux ont concerné notamment le contrôle d'accès avec comme résultat le modèle à capacités cachées [eDL] implanté dans l'environnement Java, qui permet d'intégrer une politique de contrôle d'accès exprimée sépa-

rément du code de l'application. Ils ont également concerné la duplication avec comme résultat le système Javanaise [DHb, eDL] implanté dans l'environnement Java, qui permet d'intégrer la duplication d'objets Java sans modifier l'application sur laquelle elle est appliquée. Ces deux expérimentations utilisent des objets d'interposition afin d'intégrer la gestion de ces aspects non-fonctionnels. Cependant, les objets d'interposition ne constituent pas la seule solution pour intégrer des aspects non-fonctionnels. En effet, il est possible de prendre en compte des aspects non-fonctionnels lors de la compilation du composant métier, ou alors lors de son chargement, mais dans les deux cas par modification du code du composant métier. Cette modification vise à insérer le code traitant les aspects non fonctionnels dans le code métier de l'application, faisant ainsi économie de l'utilisation d'objets d'interposition. Nous proposons d'utiliser des techniques de compilateurs adaptables (ou meta-compilateurs) afin de prendre en compte des aspects non-fonctionnelles. Plusieurs projets ont exploré les techniques de meta-compilation (par exemple OpenJava [MT] ou Javassist [TSCI]). Notre objectif est en premier lieu d'étudier l'intérêt de cette approche pour intégrer les aspects non-fonctionnels que nous avons déjà traités, à savoir le contrôle d'accès et la duplication. Un objectif plus lointain serait de fournir un langage permettant de programmer l'intégration d'aspects non-fonctionnels en réifiant à la compilation le code appelant et appelé dans les applications.

#### **4.7.2 Exprimer des aspects non-fonctionnels**

De nombreux aspects non-fonctionnels doivent pouvoir être exprimés, puis intégrés dans les applications réparties. Ceux qui sont traditionnellement abordés dans les plate-formes à composants répartis sont les aspects liés à la persistance, aux transactions et à la sécurité. Dans le projet Sardes, nous avons mené plusieurs expérimentations qui visent à gérer de façon non-fonctionnelle les aspects liés à la duplication (pour la mise en cache) [DHb, eDL, VMa] et au contrôle d'accès [DHa]. Plus récemment, nous avons étudié l'adaptation non-fonctionnelle d'une application à base de composants afin de dupliquer des composants pour tolérer des déconnexions des usagers. Dans cet axe de travail, nous proposons d'utiliser les mécanismes d'intégration d'aspects non-fonctionnels développés dans le premier axe pour traiter des aspects liés à la duplication (avec différentes conditions de duplication et de gestion de la cohérence) et au contrôle d'accès.

En complément à ces travaux, la contribution de Sardes portera également sur un autre axe de travail qui concerne les modèles à composants.

#### **4.7.3 Modèles à composant**

Deux modèles à composants ont émergés ces dernières années, le modèle à composants EJB et le modèle à composants CCM. Ces deux modèles

ne sont pas satisfaisants pour différentes raisons. Les EJB ne fournissent pas réellement une encapsulation de composants, puisqu'il n'est notamment pas possible d'observer les services requis des composants ou les interconnexions entre les composants. Le modèle CCM fournit de telles fonctions, mais il ne permet pas la construction de composants composites, ce qui nous semble essentiel pour qu'un modèle à composant soit réellement utilisable. Enfin, aucun des deux modèles ne prévoit des règles de reconfiguration ou d'adaptation des architectures des applications à base de composants. Notre objectif est de définir un modèle à composants qui possède toutes les propriétés souhaitables. Ce travail consiste en premier lieu à identifier les propriétés souhaitables, puis à définir rigoureusement un modèle possédant ces propriétés. La rigueur de cette définition passe par une description formelle du modèle. La contribution du projet Sardes dans cet axe de travail sera la définition formelle d'un modèle à composants. Ce modèle à composants pourra être implanté, par exemple comme un environnement middleware sur Java, ou comme un noyau de système d'exploitation. De telles expérimentations sont en cours [JS].

## 5 Conclusion

Nous avons fusionné la version finale du document D1.2 (Etude de l'environnement d'intégration) et la première version du document D1.3 (Document d'architecture) parce qu'il nous était impossible en l'état actuel du projet de proposer un modèle d'architecture raisonnable.

Après une justification des besoins d'adaptabilité, ce document a présenté la démarche prise par le projet ARCAD. Pour cela ce document contient :

- une présentation détaillée du modèle de composants qui sera utilisé lors des différentes expérimentations (section 2),
- une description des différents composants techniques qui pourraient être étudiés (section 3),
- une présentation commune des différentes expérimentations que nous souhaitons mener (section 4).

L'utilisation d'un environnement d'expérimentation commun aux différents membres du projet doit nous permettre :

- d'évaluer de manière comparatives différentes approches possibles,
- de proposer dans un second temps le modèle de composants ARCAD et l'architecture nécessaire à leurs supports.

Pour mener ces expérimentations autour des intergiciels adaptables et afin d'obtenir rapidement des résultats significatifs et visible, le projet ARCAD a choisi de proposer dans un premier temps plusieurs mises en oeuvre 'extensibles' de la plate-forme EJB. Notre base de travail est l'intergiciel JOnAS réalisé par Bull-Evidian et diffusé par le consortium ObjectWeb. A partir de cette source de code le projet ARCAD doit :

- proposer différents composants techniques reprenant les propriétés non-fonctionnelles aujourd’hui disponibles dans une proposition EJB (sécurité, persistance et transaction) et proposer une spécification et une implémentation d’autres composants techniques (mobilité, communication asynchrone, mode déconnecté, diffusion, etc) ;
- comparer différents mécanismes permettant la fusion de code métier et fonctionnel ;
- utiliser ces composants et mécanismes pour étendre d’autres modèles de composants existants (CCM) afin de les rendre plus extensibles ;
- et en dernier lieu, proposer un modèle de composants extensibles permettant de décrire les différentes facettes qui auront été jugées pertinentes.

## Références

- [ABV92] Mehmet Askit, Lodewijk M.J. Bergmans, and Sinan Vural. An Object-Oriented Language-Database Integration Model : The Composition-Filters Approach. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP’92)*, volume 615, pages 372–395. Springer-Verlag, June 1992.
- [BB] G. Berry and A. Benveniste. The Synchronous Approach to Reactive and Real-Time Systems , Proc. of the IEEE, vol. 79, n° 9, September 1991.
- [BER01] Laurent BERGER. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO*. thèse, Université de Nice - Sophia Antipolis, discipline Informatique, octobre 2001.
- [BG92] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.
- [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. also published as TR 01/7/INFO.
- [Dav01] Pierre-Charles David. Une infrastructure pour middleware adaptable. Rapport de DEA, École des Mines de Nantes, Université de Nantes, September 2001.

- [DHa] et L. Ismail D. Hagimont. A protection scheme for mobile agents on java, third acm/ieee international conference on mobile computing and networking (mobicom), budapest, septembre 1997.
- [DHb] F. Boyer D. Hagimont. A configurable rmi mechanism for sharing distributed java objects, *ieee internet computing*, volume 5, number 1, january 2001.
- [DIC] DICO. <http://www.essi.fr/rainbow>.
- [DLBS01] Pierre-Charles David, Thomas Ledoux, and M. N. Bouraqadi-Saâdani. Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.
- [ECM] ECMA. Submission of the common language infrastructure to ecma (ecma tc39/tg3 - <http://msdn.microsoft.com/net/ecma/>).
- [eDL] D. Hagimont et D. Louvegnies. Javanaise : Distributed shared objects for internet cooperative applications, ifip international conference on distributed systems platforms and open distributed processing (middleware'98), the lake district, septembre 1998.
- [Evi] Bull Evidian. Java open application server (jonas) for ejb.
- [Ha198] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.
- [INR] INRIA. Junior : A kernel for reactive programming in java, site du projet Mimosa – <http://www-sop.inria.fr/mimosa/rp/junior>.
- [JS] E. Najm J.B. Stefani, F. Germain. Elements of an object-based model for distributed and mobile computation, 4th international conference on formal methods for open object-based distributed systems, stanford, ca, usa, september 2000.
- [KHH<sup>+</sup>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.
- [Kic96] Gregor Kiczales. Aspect-oriented programming : A position paper from the xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming.*?, 1996.
- [Kon98] Manu Konchady. An introduction to JDBC. *Linux Journal*, 55 :34–37, November 1998.
- [LB98] A.-M. Dery et M. Fornarino L. Berger. Interactions between objects : an aspect of object-oriented languages. In *ICSE'98 Workshop on Aspect-Oriented Programming (Kyoto, Japan)*, April 1998.

- [MH98] Richard Monson-Haefel. The Java Naming and Directory Interface (JNDI) : A more open and flexible model. *Java Report : The Source for Java Development*, 3(2) : ? ?-? ?, February 1998.
- [Mica] Microsoft. Com+.
- [Micb] Microsoft. .net.
- [Micc] Sun Microsystems. J2ee : The platform for enterprise solutions (<http://java.sun.com/j2ee/overview.html>).
- [MT] S. Chiba M. Tatsubori. Programming support of design patterns with compile-time reflection, oopsla'98 workshop on reflective programming in c++ and java, vancouver, october 1998.
- [Obj] ObjectWeb. <http://www.objectweb.org>.
- [OHBS94] Harold Ossher, William Harrison, Franck Budinsky, and Ian Simmonds. Subject-Oriented Programming : Supporting Decentralized Development of Objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.
- [OMGa] OMG. Corba.
- [OMGb] OMG. Corba 3.0 ccm ftf draft ptc/99-10-04.
- [RM] P. Merle R. Marvie. Corba component model : Discussion and use with openccm.
- [Sun] Sun Microsystems. Java RMI.
- [TSCI] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of legacy java software, european conference on object-oriented programming (ecoop), 2001.
- [VMa] D. Hagimont V. Marangozova. An infrastructure for corba component replication, rapport technique sirac, novembre 2001.
- [VMb] M. Hapner V. Matena. Enterprise java beans : specification of the ejb 1.0 architecture.