

# Adaptation Dynamique

## Concepts et Expérimentations

Abdelmadjid Ketfi, Nouredine Belkhatir, Pierre-Yves Cunin

LSR-IMAG 220, rue de la chimie  
Domaine Universitaire, BP 53  
38041 Grenoble Cedex 9 France

Tel: +33 04 76 63 55 48 - Télécopie : +33 04 76 63 55 50 - Mél: [Abdelmadjid.Ketfi@imag.fr](mailto:Abdelmadjid.Ketfi@imag.fr)

### Résumé

Le travail présenté dans ce papier traite le problème de l'adaptation dynamique dans le cadre des applications à base de composants. Dynamique signifie la possibilité d'introduire des modifications dans une application en exécution sans l'arrêter. Le génie logiciel à base de composants a pour objectif le développement de gros logiciels par l'intégration de composants existants. L'ancienne notion de développement de logiciels en écrivant du code a été remplacée par l'assemblage de composants existants. Certains systèmes sont critiques et à haute disponibilité. Ils ne peuvent pas être arrêtés et doivent être maintenus en exécution. Dans la première partie de ce papier, nous présentons quelques concepts de l'adaptation dynamique. Notre approche, appliquée au modèle à composants JavaBeans est présentée dans le reste du papier. Plusieurs approches d'adaptation définissent un modèle à composants où chaque composant est constitué d'une partie fonctionnelle et d'une partie de contrôle. Cette dernière partie regroupe la liste des opérations permettant de gérer le cycle de vie du composant. Une telle approche est convenable lorsque l'on définit de nouveaux modèles, par contre elle ne peut pas être appliquée aux modèles existants qui n'ont pas été conçus pour supporter l'adaptation dynamique. Dans notre approche, nous ne faisons aucune hypothèse sur le modèle à composants de l'application à adapter. Ainsi, notre objectif est de fournir des solutions génériques indépendantes d'un modèle spécifique.

### Abstract

*This paper deals with the dynamic adaptation problem in the context of component-based applications. By dynamic we mean the ability to change an application at run-time without stopping it. Component-based software engineering focuses on building large software systems by integrating existing software components. The old notion of developing a system by writing code has been replaced by assembling existing components. Many systems are critical and non-stop. They must be highly available and cannot be stopped to be maintained. Such systems must be adapted on the fly. In the first part of this paper, we provide a survey of the main concepts of the dynamic adaptation. The remainder of the paper presents our approach applied on the JavaBeans component model. Many existing adaptation approaches are based on specific component models. These models associate in advance to each component a functional part and a control part. This last part defines a set of operations to manage the component life cycle. Such approaches are suitable for new models that have to be defined but cannot be applicable for existing models that have not been designed to support the dynamic adaptation. In our approach we do not make any supposition on the component model of the application to be adapted. Thus we try to provide generic solutions independent of a specific model.*

**Keywords:** *Dynamic adaptation, component, architecture, deployment, JavaBeans.*

## 1. Introduction

La programmation orientée composant est l'assemblage de logiciels à partir de composants existants. C'est une méthodologie qui suppose la décomposition d'une application en un ensemble d'entités bien définies qui collaborent, adressant entre autres l'extensibilité, l'interopérabilité et la réutilisation.

Adapter une application à base de composants consiste à adapter un ou plusieurs de ses composants. Traditionnellement, pour adapter un composant, l'application doit être arrêtée puis redémarrée après l'adaptation. Cette approche ne convient pas aux applications à haute disponibilité comme les applications bancaires, Internet, et les services de télécommunication où l'adaptation doit être réalisée en perturbant l'application le moins possible. En général, adapter un composant en exécution revient entre autres à le déconnecter de l'application et à connecter une autre version. Cette tâche nécessite plusieurs activités en parallèle pour assurer la cohérence de l'application.

L'un des travaux autour de l'adaptation dynamique des applications date de 1976, dans lequel R. Fabry [1] décrit comment il est possible de développer une application dont les modules peuvent être changés dynamiquement. Ce papier est organisé en deux grandes parties. La section 2 présente les concepts fondamentaux liés à l'adaptation dynamique. Les raisons pour lesquelles une adaptation peut être réalisée sont présentées en premier. Les deux sous-sections suivantes présentent respectivement comment une adaptation peut être réalisée dans le cadre d'une application à base de composants et les conditions qui doivent être remplies par une bonne approche d'adaptation. Nous discutons dans la section 3 notre approche et son application au modèle à composants JavaBeans [2] avant de conclure dans la dernière section.

## 2. Concepts liés à l'adaptation dynamique

Dans cette section, nous commençons par les raisons qui conduisent à adapter une application d'une façon générale, nous présentons ensuite comment l'adaptation dynamique peut être réalisée pour une application à base de composants. La dernière sous-section présente les conditions et les critères qui permettent d'évaluer une approche d'adaptation.

### 2.1 Raisons d'adaptation

Après avoir mis en œuvre une application, plusieurs raisons peuvent conduire à l'adapter (statiquement ou dynamiquement). Ces raisons peuvent être classées en quatre catégories :

- *Adaptation correctionnelle*: dans certains cas, on remarque que l'application en cours d'exécution ne se comporte pas correctement ou comme prévu. La solution est d'identifier le composant de l'application qui pose le problème et de le remplacer par une nouvelle version supposée correcte. Cette nouvelle version fournit la même fonctionnalité que l'ancienne, elle se contente simplement de corriger ses défauts.
- *Adaptation adaptative* : même si l'application s'exécute correctement, parfois son environnement d'exécution comme l'OS, les composants matériels ou d'autres applications ou données dont elle dépend changent. Dans ce cas, l'application est adaptée en réponse aux changements affectant son environnement d'exécution.
- *Adaptation évolutive* : au moment du développement de l'application, certaines fonctionnalités ne sont pas prises en compte. Avec l'évolution des besoins de l'utilisateur, l'application doit être étendue avec de nouvelles fonctionnalités. Cette extension peut être réalisée en ajoutant un ou plusieurs composants pour assurer les nouvelles fonctionnalités ou même en gardant la même architecture de l'application et étendre simplement les composants existants.
- *Adaptation perfective* : l'objectif de ce type d'adaptation est d'améliorer les performances de l'application. A titre d'exemple, on se rend compte que l'implémentation d'un composant n'est pas optimisée. On décide alors de remplacer l'implémentation du composant en question. Un autre exemple peut être un composant qui reçoit beaucoup de requêtes et qui n'arrive pas à les satisfaire. Pour éviter la dégradation des

performances du système, on diminue la charge de ce composant en installant un autre composant qui lui partage sa tâche.

Ces différentes raisons concernent les applications d'une façon générale et ne sont pas spécifiques aux modèles à composants. Dans la section suivante, nous présentons comment l'adaptation dynamique peut être réalisée pour une application à base de composants.

## 2.2 Types d'adaptation

Plusieurs types d'adaptation peuvent être identifiés. Certaines adaptations affectent les clients de l'application et sont plus compliquées à réaliser. D'autres affectent l'architecture de conception ou l'architecture de déploiement de l'application.

### Adapter l'architecture conceptuelle de l'application

- *Ajouter un nouveau composant.* Par exemple, ajouter un composant dans une application pour l'étendre avec un nouveau service. Cette adaptation implique l'adaptation des interconnexions entre les composants. Ce type d'adaptation peut être classé en deux catégories :
  - Instancier le nouveau composant à ajouter à partir d'un type déjà chargé par l'application.
  - Créer le nouveau composant à partir d'un nouveau type.Quand il est ajouté à une application en cours d'exécution, en général, un composant doit prendre en compte l'état de l'application, il doit donc récupérer l'état à partir duquel il doit démarrer et se personnaliser en fonction de cet état.
- *Supprimer un composant existant.* Le composant supprimé ne doit pas affecter l'exécution des autres composants. Il doit être aussi dans un état stable avant qu'il soit supprimé. Par exemple, si un composant est en cours d'écrire des données dans un fichier, il ne doit pas être supprimé avant la terminaison de sa tâche. Un autre défi concerne les données et les messages échangés entre le composant en question et les autres composants, ces données ou messages ne doivent pas être perdus.
- *Modifier les interconnexions entre composants.* Par exemple, quand un nouveau composant est ajouté, il doit être connecté aux composants existants. En général, lorsque deux composants sont connectés, les types de leurs ports connectés doivent être compatibles. Dans une application répartie, la connexion entre deux composants spécifie le type de communication entre les deux et dépend de la localisation des deux composants. En d'autres termes, la communication entre deux composants dans le même espace d'exécution est différente de la communication entre deux composants dans deux espaces d'exécution différents sur la même machine. Elle est également différente de la communication entre deux composants s'exécutant sur deux machines différentes. L'adaptation des interconnexions doit préserver les messages et les données en transit.

### Adapter l'implémentation d'un composant

Ce type d'adaptation est généralement considéré comme une adaptation perfective, adaptative ou correctionnelle. Dans le premier cas, l'adaptation est motivée par des raisons de performance, par exemple, on décide de remplacer un composant de stockage utilisant un *vecteur* par un autre composant utilisant une *Hashtable*. Un autre exemple peut être un administrateur d'une application qui souhaite expérimenter plusieurs algorithmes implémentant la même fonctionnalité, il doit pouvoir passer d'une implémentation à une autre sans arrêter l'application. Du point de vue adaptative, on peut donner l'exemple d'un utilisateur qui change d'imprimante ou d'un autre matériel, le *pilote* de ce matériel doit changer. La vision correctionnelle de l'adaptation est simplement la modification de l'implémentation pour résoudre des bugs ou des problèmes constatés dans l'implémentation courante.

### Adapter l'interface d'un composant

Ceci signifie la modification de la liste des services fournis par le composant. Ceci peut être réalisé en ajoutant/supprimant une interface à/de la liste des interfaces fournies par le composant.

- *Ajouter une interface* : l'objectif est d'étendre les fonctionnalités du composant. Les fonctionnalités exposées à travers les autres interfaces restent inchangées.
- *Supprimer une interface* : lorsqu'un composant expose une interface, il doit l'implémenter. Si le service en question n'est pas et ne sera pas utilisé, il est important pour des raisons de performance de supprimer cette interface et par conséquent son implémentation.

### Adapter l'architecture de déploiement de l'application

Ceci correspond à la migration d'un composant d'un site à un autre pour la répartition des charges par exemple. Ce type d'adaptation n'affecte pas la structure de l'application, cependant, la communication entre le composant déplacé et les autres composants doit être adaptée selon la nouvelle localisation du composant. L'état interne du composant doit être sauvegardé et réinjecté dans le composant dans sa nouvelle localisation. Les messages reçus et non encore traités doivent être pris en compte.

## 2.3 Performance de l'adaptation

Plusieurs conditions doivent être vérifiées par une opération d'adaptation. Dans cette section, nous présentons les paramètres qui permettent d'évaluer une approche d'adaptation.

### La cohérence

L'application d'administration qui se charge de lancer l'opération d'adaptation est plus prioritaire que l'application administrée. Ceci ne lui donne pas le droit de tout faire avec cette application. L'opération d'adaptation doit préserver la cohérence de l'application adaptée. Elle ne doit pas être lancée à n'importe quel moment, sinon, il faut prévoir des mécanismes pour que cette opération ne force pas l'application à s'adapter brusquement, mais doit lui laisser le temps de passer dans un état cohérent.

La cohérence de l'application peut être locale ou globale. Dans le premier cas, elle concerne un composant indépendamment des autres, par exemple, lorsqu'un composant est altéré au moment où il modifie l'une de ses ressources, il ne peut pas revenir dans un état cohérent. La cohérence globale concerne l'application dans sa globalité, par exemple, les messages transitant dans l'application ne doivent pas être perdus.

La cohérence de l'application peut être résumée par les points suivants :

- *Sécurité* : une opération d'adaptation mal faite ne doit pas conduire l'application adaptée à un crash.
- *Complétude* : au bout d'un certain temps, l'adaptation doit se terminer, et doit au moins introduire les changements nécessaires à l'ancienne version de l'application.
- *Well-timedness* : il faut lancer l'adaptation aux bons moments. Le programmeur doit spécifier à l'avance les points d'adaptation.
- *Possibilité de rollback* : même si par un moyen ou un autre on démontre la correction (*correctness*) de l'adaptation, certaines erreurs peuvent échapper aux preuves. Il faut disposer de moyens permettant d'annuler l'adaptation et faire retourner l'application à son état initial.

### Degré d'automatisation

Le degré d'automatisation représente la capacité de l'application de réaliser sa propre adaptation. Ceci est possible car en exécution, l'application a toutes les informations pour décider qu'une adaptation est nécessaire, et également, toutes les capacités pour réaliser une telle adaptation sans l'intervention d'un administrateur humain. Cette automatisation n'est pas toujours possible car certains changements dans l'environnement d'exécution ne peuvent pas être captés par l'application, et il n'est pas possible de tout prévoir au moment de la conception de l'application.

### 3. Notre Approche

Plusieurs approches d'adaptation (DCUP [3], OLAN [4],...) définissent un modèle à composants où chaque composant est constitué d'une partie fonctionnelle et d'une partie de contrôle. Cette dernière partie regroupe la liste des opérations permettant de gérer le cycle de vie du composant (arrêt, passivation, mise à jour, ...). Une telle approche est convenable lorsque l'on définit de nouveaux modèles, par contre elle ne peut pas être appliquée aux modèles existants qui n'ont pas été conçus pour supporter l'adaptation dynamique.

Notre approche consiste à séparer la partie contrôle de la partie applicative. L'objectif est de fournir des solutions génériques indépendantes des modèles et de leurs capacités de supporter l'adaptation.

#### 3.1 Processus d'adaptation

Le processus d'adaptation peut être constitué de plusieurs étapes :

- Définition des règles de correspondance entre l'ancien et le nouveau composant.
- Passivation des deux composants.
- Le transfert de l'état de l'ancien composant vers le nouveau selon les règles de correspondance définies dans la première étape.
- Déconnexion de l'ancien composant et connexion du nouveau.
- Activation du nouveau composant.

Ces activités seront introduites en détail dans le cadre de notre expérimentation sur le modèle JavaBeans. Un Bean est un composant logiciel écrit en Java. Il peut être manipulé par programmation ou visuellement pour construire des applications.

#### 3.2 Communication entre les composants

Afin de pouvoir adapter en temps réel un composant, il est nécessaire de pouvoir contrôler toutes ses communications entrantes et sortantes (appels du et vers le composant). Dans le cas où le composant définit une partie de contrôle, cette communication peut être facilement contrôlée car le composant participe lui-même dans sa propre adaptation. Dans le cas contraire, il faut éviter que la communication soit directe entre les objets et donc intercepter et rediriger les appels. Cette redirection peut coûter cher en termes de performances. Pour cette raison, la redirection n'est appliquée qu'aux composants susceptibles d'être adaptés. Trois types de composants peuvent être alors dégagés :

- Composants adaptables : leur adaptation est gérée par la machine ou l'environnement de contrôle. Il suffit que le concepteur les déclare explicitement adaptables. Les services rendus par un composant de ce type sont fournis d'une manière indirecte.
- Composants non-adaptables : ce sont des composants installés définitivement et ne peuvent pas être adaptés. La communication avec ce type de composants est directe.
- Composants auto-adaptables : ce sont des composants adaptables et qui ont la capacité de réaliser et de participer dans leur propre adaptation. Ils se contentent de fournir une interface qui regroupe la liste des opérations d'administration assurées par le composant.

La figure 1 présente l'adaptateur utilisé pour communiquer avec un composant adaptable. Plusieurs points peuvent être constatés :

- Le composant cible peut être dynamiquement adapté.
- La méthode cible peut être dynamiquement adaptée. Ceci est nécessaire si le nouveau composant ne définit pas la même méthode.
- Un composant peut être soit passif ou actif.
- Si un composant est passif, les requêtes reçues sont stockées dans une liste.
- Les événements stockés sont estampillés. Quand le composant est activé, l'estampille permet de traiter les requêtes dans le bon ordre.

```

public class ____Hookup_17bcab15c1 implements ModelListener,
    java.io.Serializable {

    public void setTarget(Object t) {
        target = t;
    }

    public void setTargetMethod(Method m) {
        targetMethod = m;
    }

    public void passivate() {
        active = false;
    }

    public long getCurrentStamp() {
        if(waitingList.isEmpty()) { active = true; return -1; }
        return (long) ((StampedEvent)waitingList.get(0)).stamp;
    }

    public void sendTheCurrentEvent() {
        try {
            targetMethod.invoke(target, new Object[]
                {((StampedEvent)waitingList.get(0)).argument});
            waitingList.remove(0);
            if(waitingList.isEmpty()) active = true;
        }
        catch(Exception e) { e.printStackTrace(); }
    }

    public void draw(ModelEvent arg0) {
        try{
            if(active)
                targetMethod.invoke(target, new Object[] {arg0});
            else
                waitingList.add(new StampedEvent(arg0));
        }
        catch(Exception ex) { ex.printStackTrace(); }
    }

    private Object target;
    Method targetMethod;
    boolean active = true;
    List waitingList = new ArrayList();
}

```

Figure 1 : Adaptateur d'un composant adaptable

### 3.3 Règles de correspondance

Un Bean contient potentiellement un ensemble de classes. Cet ensemble peut complètement différer entre l'ancien et le nouveau Bean d'où la nécessité de définir des règles de correspondance entre les deux Beans. La correspondance se fait entre les propriétés et d'autre part entre les méthodes.

**Propriétés :** une propriété du nouveau Bean peut correspondre à plusieurs propriétés de l'ancien. La correspondance de propriétés est nécessaire pour le transfert d'état.

**Méthodes :** une méthode fournie par le nouveau composant peut correspondre à plusieurs méthodes fournies par l'ancien. La correspondance des méthodes est indispensable pour la reconnexion dynamique.

### 3.4 Passivation et activation d'un composant

Pour des raisons de cohérence, un Bean doit être mis à l'état *passif* avant d'être adapté. Passif signifie qu'aucune requête ne peut être servie par le Bean. Ces requêtes sont sauvegardées ou non selon la stratégie appliquée. La sauvegarde des requêtes dépend aussi du type de communication entre les Beans. Une communication asynchrone (par événements) est plus facile à gérer qu'une communication synchrone (appel de méthodes).

### 3.5 Transfert d'état

Le nouveau Bean doit démarrer à partir de l'état de l'ancien. Les valeurs des propriétés sont transférées selon les règles de correspondance précédemment définies. L'état d'un Bean est représenté par l'ensemble des valeurs des propriétés de tous les objets constituant le Bean, et ayant des correspondances dans le nouveau Bean. Dans l'état actuel, le transfert ne concerne que les propriétés publiques, une limitation qui sera levée dans des travaux futurs.

### 3.6 Reconnexion dynamique

La reconnexion dynamique peut être vue à deux niveaux de granularité : Bean ou méthode. La figure 2 montre la déconnexion du composant Bean1 et la connexion du composant Bean2 qui prend sa place. La reconnexion est possible car la communication entre les Beans connectés est indirecte. La reconnexion peut passer par deux étapes :

- Tous les adaptateurs qui pointent sur l'ancien Bean doivent pointer sur le nouveau. Autrement dit, la référence de l'ancien Bean dans ces adaptateurs doit être remplacée par la référence du nouveau. Si la méthode fournie par le nouveau Bean est différente de l'ancienne, la référence de la méthode appelée dans l'adaptateur doit être adaptée selon la nouvelle méthode.
- Tous les adaptateurs vers lesquels le Bean adapté pointe doivent se désabonner de ce Bean et s'abonner auprès du nouveau.

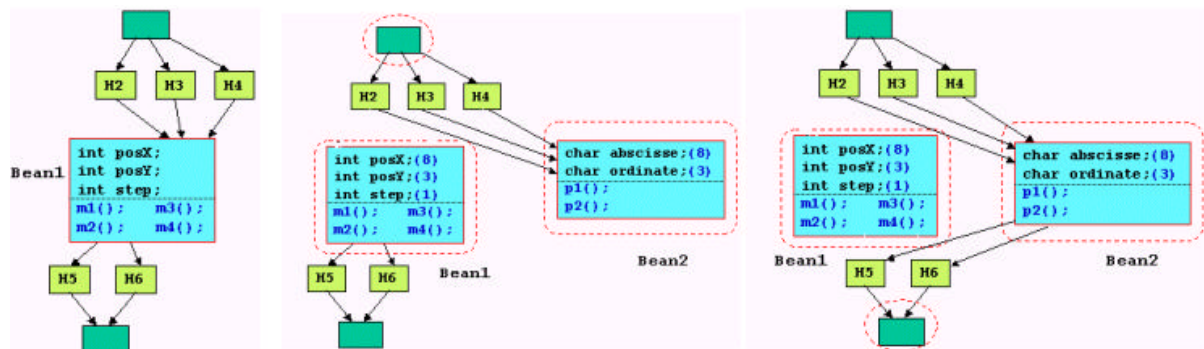


Figure 2 : Reconnexion dynamique des Beans

## 4. Conclusion

Dans ce papier, nous avons discuté certains concepts fondamentaux liés à l'adaptation dynamique et spécifiquement dans le cadre des applications à base de composants. Certaines applications sont critiques et doivent être adaptées en exécution avec le minimum de perturbation. Les approches d'adaptation diffèrent dans la granularité adressée (procédure, module, objet, composant), dans les types d'adaptation supportés (implémentation, architecture de conception, interface, architecture de déploiement) et dans leurs performances. Plusieurs approches partent du principe que les opérations qui permettent d'adapter un composant doivent être prévues au moment de sa conception. Elles sont ainsi basées sur des modèles à composants spécifiques qui incluent en plus de la partie fonctionnelle une partie de contrôle. De telles approches ne peuvent pas être appliquées pour adapter des applications basées sur des modèles existants. Dans notre approche, notre objectif est de séparer les opérations d'administration du composant afin de fournir des solutions génériques. Cet objectif reste difficile à atteindre tant que les composants ne participent pas dans leur administration. Il devient nécessaire d'éviter les connexions directes entre les composants afin de pouvoir contrôler leur communication. Il est également difficile de réaliser le transfert d'état de l'ancien composant vers le nouveau car la spécification de l'état lui-même dépend de la sémantique de l'application. Par conséquent, nous pensons que certaines opérations d'adaptation peuvent être automatisées et d'autres opérations nécessitent la participation du composant et doivent être prévues à la conception de l'application.

## Références

- [1] R.S. Fabry : "How to design a system in which modules can be changed on the fly", *Proc. 2nd Int. Conf. on Soft. Eng.*, pp. 470-476 (1976).

- [2] *JavaBeans Architecture*, Sun Microsystems.  
<http://java.sun.com/docs/books/tutorial/javabeans/>
- [3] F. Plasil, D. Balek, R. Janecek : "DCUP: Dynamic Component Updating in Java/CORBA Environment", Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague, 1997.
- [4] N. De Palma, L. Bellissard, M. Riveill : "Dynamic Reconfiguration of Agent-based Applications", European Research Seminar on Advances in Distributed systems (ERSADS'99), Madeira, Portugal, April 1999.