

---

# L'essayer et l'adapter : une approche pour améliorer la confiance dans l'usage d'un composant

Pham Thi Xuan Loc<sup>(\*)</sup> — Philippe Mauran — Gérard Padiou

*Institut de Recherche en Informatique de Toulouse, UMR CNRS 5505,  
ENSEEIHT, 2 rue Camichel, BP 7122, 31071 Toulouse cedex 7*

*(\*) Université de Cantho, Vietnam  
{mauran,padiou,ptxl}@enseeiht.fr*

---

*RÉSUMÉ. Nous nous intéressons à l'adaptation des composants à l'usage qui en est fait par leurs clients. Ceci passe par une définition du comportement de ces clients ou usagers de telle façon qu'un contrôle puisse être fait sur l'usage effectif des composants et éventuellement envisager des mesures d'adaptation pour améliorer le service offert aux usagers par ces composants. À travers un exemple illustratif, nous définissons le problème de la sûreté d'usage de composants par leurs usagers. Nous proposons une approche pour garantir cette sûreté d'usage grâce à la notion de profil. Pour terminer, le principe d'une mise en œuvre d'un tel service de sûreté d'usage est abordé.*

*ABSTRACT. We investigate the adaptability of components to their client use. This goal implies to specify the user behavior to control the effective use of components. Furthermore, this control may be completed by carrying out the dynamic adaptation of components to increase the provided service. Through an illustrative sample, we first define the problem of use safety. Then, we propose an approach to insure this safety thanks to the notion of profile. Last, a pattern is proposed for the implementation of a safety service.*

*MOTS-CLÉS : composants, profils d'usage, sûreté de service, adaptation dynamique*

*KEYWORDS: components, use profiles, safety of service, dynamic adaptability*

---

## 1. Introduction

Cet article aborde la question de l'adaptation d'un composant sous un angle neuf, à notre connaissance. Habituellement l'adaptation est réalisée entre un composant et son environnement d'exécution : structure d'accueil pour réaliser l'adaptation à une plateforme, container pour apporter au composant des services système de base (J2EE [DEM 01],[BRI 02]; CCM [MIS 99]), ou encore spécialisation du code des composants [MCN 01]. Dans cet article, nous envisageons l'adaptation du composant à (chacun de) ses contextes d'utilisation. Autrement dit, nous abordons le thème de la bonne utilisation des composants, du point de vue de l'utilisateur, davantage que du point de vue du concepteur, de l'installateur ou de l'administrateur de composants.

L'adéquation entre le service proposé par le composant et son usage est une question cruciale du point de vue de la réutilisabilité. Cette question peut être développée sous divers aspects : définition, composition et association de services aux composants [BAR 02], spécification et courtage des services proposés et requis, contrôle statique ou dynamique de l'écart entre service attendu et service fourni, spécification et gestion de la liaison entre composant et application, définition et utilisation de métaprotocoles pour mettre cette liaison en œuvre dynamiquement [GEI 97].

Nous abordons cette question sous l'angle suivant : au lieu de considérer et gérer des spécifications élaborées par les concepteurs de composants, nous proposons de partir d'une expression de l'utilisation, fournie selon un point de vue proche de l'utilisateur final. Cette expression, que nous appelons « profil », doit permettre à l'utilisateur de caractériser les comportements attendus, possibles, ou anormaux du composant au regard de l'utilisation particulière qu'il en fait, ainsi que les adaptations demandées au comportement du composant. L'idée, que l'on peut rapprocher du classique argument de bout en bout, est que la connaissance du contexte d'utilisation est nécessaire pour interpréter et adapter correctement le comportement du service utilisé, et que cette connaissance ne peut pas être anticipée ou intégrée au moment de la conception du composant. Le composant ne pouvant (pratiquement par définition) prévoir l'ensemble de ses (ré)utilisations possibles, l'adaptation au contexte d'utilisation paraît devoir être réalisée de manière dynamique, et spécifiée de manière externe au composant sous la forme d'un profil d'usage. Les propriétés spécifiées dans un tel profil seront généralement :

- globales, c'est-à-dire, exprimées indépendamment de l'architecture (fonctionnelle) du composant, ou de la spécification de son déploiement (autres aspects),
- exprimées de manière plus abstraite, c'est-à-dire avec des références à l'univers de l'utilisation davantage qu'à l'univers de l'implémentation, et donc dans un langage distinct des langages de mise en œuvre.

Nous développons cette approche en proposant un service d'adaptation, ou de « sûreté de service », permettant de contrôler la conformité des comportements de composants par rapport aux profils spécifiés. Plus largement, il s'agit d'offrir la possibilité d'articuler le comportement défini par le concepteur du composant et celui qui est lié à son utilisation. Ce service est conçu comme une couche d'interposition, entre l'ap-

plication utilisatrice et le conteneur du composant [KRA 01]. Il peut s'avérer utile de plusieurs façons :

- pour contrôler dynamiquement le comportement du composant, en le restreignant, afin de garantir des propriétés fixées arbitrairement et a posteriori par l'utilisateur (sûreté de service). Cette exploitation du mécanisme peut être vue comme une extension de la notion de bac à sable.

- pour adapter le service assuré par le composant, en associant un traitement aux propriétés définies par le profil, ce qui peut permettre de composer (ou de superposer) le comportement du composant avec un comportement défini par l'utilisateur. Cette exploitation du mécanisme d'adaptation peut être vue comme une extension du mécanisme classique d'exception.

- pour permettre l'évolution dynamique du service assuré par le composant : il serait ainsi possible de modifier ou de compléter un composant en lui attachant systématiquement un profil donné.

La section suivante présente un exemple destiné à permettre d'illustrer la notion de profil, motivée et introduite par les sections qui suivent. Enfin, la dernière section présente les principes de réalisation du mécanisme d'adaptation

## 2. Exemple illustratif de la notion d'usage

À partir de la spécification d'un service réparti de change de monnaies, on propose une architecture logicielle de ce service en termes de composants. Les usages du service sont mis en évidence et permettent de poser le problème de la sûreté d'usage ainsi que les bases de l'approche pour le résoudre.

### 2.1. *Un exemple de service*

On considère une spécification d'un service en langage idl décrite pour une implantation sur une plateforme répartie utilisant CORBA. Le service considéré est un service de change de monnaies. On suppose qu'un serveur central enregistre les offres de change faites par des institutions bancaires. Cet enregistrement se fait sous la forme de couples de monnaies liées par un taux de change qui peut donc être mis à jour par les institutions. Les clients potentiels ou les institutions elles-même devront s'adresser à ce serveur central pour obtenir une liste des institutions acceptant le change d'un couple de monnaies fixées.

L'interface `ChangeDeMonnaies` décrit l'interface du serveur central. La méthode `Enregistrer` sera appelée par une institution pour enregistrer un nouveau couple de monnaies qu'elle accepte d'échanger. La méthode `Supprimer` exécute l'opération inverse. Les opérations `Suspendre` et `Reprendre` permettent de suspendre momentanément le change d'un couple de monnaies. La méthode `Disponibles` sera appelée

par un client ou une institution pour connaître les institutions de change acceptant l'échange d'un couple de monnaies fixées.

L'interface **Changeur** décrit le service de change proprement-dit d'un **couple particulier** de monnaies offert par une institution. Une même institution peut enregistrer plusieurs couples de monnaies qui sont autant d'objets de classe **Changeur**. La méthode **Combien** permet de tester le change d'une des deux monnaies du couple en spécifiant en paramètre la monnaie à convertir et le montant. La méthode **Changer** exécute effectivement le change.

```

module Conversion {
  /* Service de change proposé par les institutions */
  interface Changeur {
    readonly attribute float  tauxDeChange ;
    readonly attribute string devSrc, devCible ;
    readonly attribute string institution ;
    float modifierTaux ( in float nouveauTaux ) ;
    float Combien(in string devVal, in float val ) ;
    /* renvoie devVal * TauxDeChange si devVal = devSrc
       devVal / TauxDeChange si devVal = devCible */
    float Changer ( in string devVal, in float val ) ;
    /* même résultat que Combien mais avec change effectif */
  };
  /* Serveur central enregistrant les changes disponibles */
  exception ChangeInconnu {} ;
  typedef sequence<Changeur> Changeurs;
  interface ChangeDeMonnaies {
    void Enregistrer (in Changeur unChangeur);
    void Supprimer ( in Changeur unChangeur) ;
    void Suspendre ( in Changeur unChangeur ) ;
    void Reprendre ( in Changeur unChangeur ) ;
    Changeurs Disponibles(in string Dev1, in string Dev2) ;
  };
};

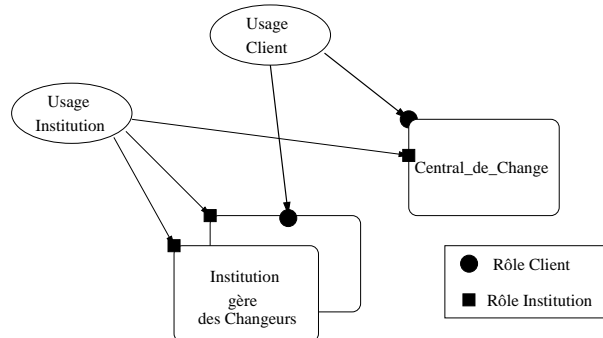
```

## 2.2. Architecture logicielle du service

L'architecture logicielle du service en termes de composants est illustrée par la figure 1. Celle-ci peut être décrite dans un langage de description tel que Acme[ALL 97].

Chaque institution représente un composant qui gère des monnaies et met à disposition des offres de change sous la forme d'objets implantant l'interface **Changeur**. Le composant serveur central enregistre les offres des institutions. Chaque composant accepte deux rôles d'utilisation : l'un pour tout usager institution et l'autre pour

tout usager client du service. Les méthodes définissant ces rôles caractérisent donc les droits d'usage de chaque catégorie d'utilisateur.



**Figure 1.** Architecture logicielle du service de change

### 3. Sûreté d'usage des composants

#### 3.1. Interactions usagers-composant

Nous nous intéressons à l'adaptabilité des composants à partir de règles d'usage spécifiées par les usagers. En effet, les usagers (clients) d'un composant (service) peuvent fournir implicitement ou explicitement des informations sur l'usage qu'ils font du composant.

Des informations implicites peuvent être observées par le composant lui-même : elles concernent généralement la « journalisation » des interactions client-serveur sous forme de trace d'appel, de calcul de fréquences, etc. Le composant peut alors utiliser ces informations pour s'adapter aux besoins de ses usagers (en terme de qualité de service en particulier).

Nous considérons plus spécifiquement les informations explicites que pourraient fournir l'utilisateur d'un composant. Ces informations peuvent concerner différents aspects : contraintes de qualité de service, performances, scénarios particuliers, invariants que l'utilisateur souhaiterait voir garantis, ainsi que les traitements devant être réalisés, dans le cas où les contraintes ne sont pas satisfaites. Ces informations constituent un « profil d'usage » caractérisant les particularités de l'utilisation d'un composant par un usager. Un tel profil définit des règles sur la trace des interactions entre des composants et des usagers. Le tableau de la figure 2 résume les différentes traces envisageables. L'utilisation d'un profil d'usage peut, pour sa part, porter :

- sur la trace globale d'un usager relativement aux composants qu'il référence : c'est le « container » de l'utilisateur qui exploitera le profil ;
- sur la trace globale des usagers d'un composant : c'est le container du composant qui exploitera le profil.

interactions	un composant	tous les composants
un usager	trace locale	trace globale d'un usager
tous les usagers	trace globale d'un composant	trace globale système

**Figure 2.** *Traces des interactions usager(s)-composant(s)*

### 3.2. *Le contrôle de l'usage des composants*

Une application fait un usage particulier d'un ensemble de composants. On dispose en général d'une description partielle de l'usage correct des composants. En effet, les utilisations possibles d'un composant sont souvent incomplètement spécifiées. On peut distinguer classiquement deux types de propriétés à vérifier : les propriétés de sûreté et les propriétés de vivacité. En ce qui concerne les propriétés de sûreté, tout composant est caractérisé par un plus fort invariant qui définit les états possibles du composant. L'utilisateur du composant ne connaît pas forcément ce plus fort invariant et l'usage qu'il fait du composant peut alors être incorrect de deux façons :

- soit l'utilisateur utilise mal le composant : l'exécution d'une méthode lève une exception si cette exécution conduit à un état erroné pour le composant. Par exemple, demander de changer un montant supérieur aux disponibilités d'une institution.

- soit l'utilisateur suppose un invariant plus restrictif que celui garanti par le composant. Auquel cas, l'utilisateur risque de ne pas obtenir les résultats qu'il espérait en utilisant le composant. Par exemple, l'utilisateur suppose que l'offre de change qu'il obtient d'une institution est valide durant une période longue alors que l'institution change fréquemment ses taux de change.

C'est ce deuxième cas qui nous intéresse. Il nous semble important que l'utilisateur donne le maximum d'informations sur les propriétés qu'il suppose sur les composants qu'il utilise.

## 4. *Spécification des règles d'usage*

### 4.1. *Un exemple de profil*

Un profil permet de spécifier l'usage de composants par un client. Par exemple, nous considérons le profil *Institution* qui définit le comportement d'une institution en tant que cliente à la fois du composant serveur *Central* et des autres institutions.

La définition d'un profil comporte les éléments suivants :

- les composants référencés par l'utilisateur (clause *use*) ;
- le(s) scénario(s) d'usage défini(s) par l'utilisateur : il s'agit de définir ici les appels de méthodes sur les composants référencés par l'utilisateur . On utilise la notion de

causalité pour décrire les propriétés de sûreté dans l'ordonnement des opérations exécutées par l'utilisateur. Pour les propriétés de vivacité, on utilise simplement des directives précisant si une opération est finalement appelée une seule fois (once), appelée au moins une fois (at\_least\_once) ou au plus une fois (at\_most\_once). On définit un ordre partiel entre ces opérations dans la mesure où l'utilisateur aura un comportement particulier lors de chaque exécution. Néanmoins, les relations d'ordre partiel entre appels de méthodes spécifient un profil dynamique d'utilisateur.

- des contraintes pouvant porter sur plusieurs aspects non fonctionnels. Ici, on considère des contraintes temporelles et/ou de performance ;

- des préférences, c'est-à-dire des propriétés voulues par l'utilisateur mais que les composants utilisés ne garantissent pas de façon fiable. Par exemple, la détection de l'invalidation d'une préférence sera assurée par la levée d'une exception.

Dans l'exemple ci-dessous, le scénario indique qu'une institution interroge toutes les autres institutions pour connaître les changeurs qu'elles offrent pour un couple de devises avant de modifier éventuellement son propre taux de change pour ce couple.

```

Profil Institution = {
  Use = { Central_de_Change Central ; }
  Object c : Changeur (d1,d2) ∈ self ;
  Scenario = {
    forall a ∈ Central.Disponibles(d1,d2) :
      a.Combien(*) < c.modifierTaux(*) ;;
    once Central.Enregistrer(c) ;;
    forall a ∈ Central.Disponibles(d1,d2) :
      at_least_once a.Combien(*) ;;
  }
  Constraints = {
    rate(c.setTaux)<2/h ;;
    date(Central.Reprendre(c))-date(Central.Suspendre(c))<1h ;;
    rate(Central.Suspendre(c))<2/j ;;
  }
  Preferences = { a ∈ Central.Disponibles(d1,d2) :
    when |(c.Taux - a.Taux)/c.Taux| ≤ 10%
      { ... code d'ajustement du taux ... };;
  }
}

```

La description des intentions d'utilisation peut recouvrir une variété d'aspects tant fonctionnels que non fonctionnels. Chaque aspect fera a priori appel à ses propres modèles, introduira un langage et des constructions qui lui sont spécifiques. En outre, il paraît difficile d'établir une liste définitive et figée des aspects pouvant être introduits dans l'expression des intentions d'utilisation. Dans ce contexte,

- le recours à un mécanisme dynamique comme le chargement de « plug-ins » nous paraît une solution répondant à la variété des aspects possibles, en ce qui

concerne l'expression et l'évaluation des propriétés, ainsi que les réactions qui peuvent être associées à ces dernières.

– la définition des différents aspects en référence à un cadre unique nous semble l'approche la plus sûre, en ce qu'elle permettra d'établir et de définir des correspondances entre les différents modèles et langages, et d'intégrer la sûreté de service à un environnement plus global. Dans cette perspective, le choix d'un formalisme de (méta)modélisation à vocation universelle, comme le MOF[OMG 02] nous semble plus adéquat que le choix d'étendre un formalisme comme Z [POT 91] bénéficiant d'une sémantique solide, mais dont les extensions pourraient rester discutables.

– à court terme, en revanche, notre approche est de choisir et développer quelques aspects spécifiques, mais que nous estimons significatifs (ordonnancement, sécurité, cohérence et qualité des données...), afin d'aboutir assez vite à une plateforme expérimentale pour la sûreté de service, afin d'en évaluer sa pertinence.

#### **4.2. Un patron de sûreté d'usage**

Pour vérifier que des usagers utilisent de façon sûre des composants, nous proposons l'approche suivante :

– un usager précise un profil d'usage des composants qu'il référence. Ce profil contient des informations concernant divers aspects définissant l'usage correct des composants du point de vue de l'usager;

– Côté usager : Tout usager est encapsulé dans un container qui dispose du profil et contrôle dynamiquement le respect des propriétés spécifiées dans le profil. Si une propriété est mise en défaut, une exception d'usage erroné est levée. L'usager peut alors décider du traitement de cette exception.

– Côté composant : Tout composant est encapsulé dans un container qui connaît le profil de ses usagers. Ce container peut alors comparer les besoins des usagers par rapport aux propres possibilités du composant. Par exemple, l'évaluation et le contrôle du nombre d'usagers simultanés peuvent être exploités au minimum pour en informer les usagers, au mieux pour réguler la charge.

### **5. Conclusion**

La définition des aspects d'usage les plus pertinents dans un profil, ainsi que leur exploitation dynamique afin d'adapter le composant à chacune de ses utilisations constitue un point fondamental de l'approche proposée.

De ce point de vue, les langages de description de composants (ADL : Architecture Definition Language) peuvent apporter des éléments dans la mesure où de tels langages spécifient de nombreux aspects d'un composant. Une certaine homogénéité de description doit même être recherchée dans la mesure où l'on peut envisager de valider a priori un profil d'usage par rapport à la description du composant.

Un autre thème important est de trouver les aspects d'usage auxquels un composant puissent s'adapter. Il existe au moins un aspect possible, celui des ressources. La spécification dans un profil de la qualité de service espéré par l'utilisateur du composant peut être utilisée pour adapter l'ordonnement des requêtes de l'utilisateur et l'allocation de ressources au composant.

Par ailleurs, une étude de l'intégration des profils à la description architecturale des composants devra être entreprise de manière similaire à l'étude réalisée pour J2EE avec le langage d'architecture Armani[SOU 00].

Enfin, une mise en œuvre dans un environnement CORBA ou J2EE permettra de mieux cerner les problèmes de coût de ce service de sûreté d'usage dans le contexte d'une plate-forme répartie de composants.

## 6. Bibliographie

- [ALL 97] ALLEN R., GARLAN D., « A Formal Basis for Architectural Connection », *ACM Transactions on Software Engineering and Methodology*, vol. 6, n° 3, 1997, p. 213-249.
- [BAR 02] BARTORELLO M., MAGUIN H., OCCELLO A., BLAY-FORNARINO M., DERY A.-M., RIVEILL M., « Intégration de services au sein d'un serveur d'EJB », *L'Objet*, vol. 8, n° 1-2, 2002, p. 163-184, HERMES.
- [BRI 02] BRILL G., *CodesNotes for J2EE*, RANDOM HOUSE, 2002.
- [DEM 01] DEMICHEL L. G., YALCINALP L. U., KRISHNAN S., *Enterprise JavaBeans<sup>TM</sup> Specification Version 2.0*, SUN microsystems inc., April 2001.
- [GEI 97] GEIB J.-M., GRANSART C., MERLE P., *Corba : des concepts à la pratique*, Inter-Editions, 1997.
- [KRA 01] KRAKOWIAK S., « Architecture des systèmes, passé et avenir », *CFSE*, Avril 2001.
- [MCN 01] MCNAMEE D., WALPOLE J., PU C., COWAM C., KRASIC C., GOEL A., WAGLE P., CONSEL C., MULLER G., MARLET R., « Specialization Tools and Techniques for Systematic Optimization of Software Systems », *ACM Transactions on Software Engineering and Methodology*, vol. 19, n° 2, 2001, p. 217-251.
- [MIS 99] MISCHKINSKY J., « CORBA 3.0 New Components Chapters », TC Document - CCM FTF Draft n° ptc/99-10-04, october 1999, OMG.
- [OMG 02] OMG, « Meta-Object Facility (MOF), version 1.4 », Document – formal/02-04-03, april 2002, OMG.
- [POT 91] POTTER B., SINCLAIR J., TILL D., *An introduction to formal specification and Z*, Prentice Hall International, Series in Computer Science, 1991.
- [SOU 00] SOUSA J. P., GARLAN D., « Formal Modeling of the Enterprise JavaBeans<sup>TM</sup> Component Integration Framework », Report n° CMU-CS-00-162, september 2000, Carnegie Mellon University.