Préservation de choix architecturaux lors de l'évolution d'un composant

Chouki Tibermacine, Régis Fleurquin, Salah Sadou

Laboratoire Valoria
Université de Bretagne Sud
Campus de Tohannic
F-56000 Vannes Cedex
(Chouki.Tibermacine,Regis.Fleurquin,Salah.Sadou)@univ-ubs.fr

RÉSUMÉ. *Tout logiciel doit évoluer pour répondre aux exigences changeantes de ses utilisateurs et aux modifications de son environnement. Ces changements, souvent imprévisibles, réalisés par un tiers et dans l'urgence, mènent parfois le logiciel vers un état que ses créateurs n'auraient pas souhaité. Nous présentons dans ce papier un cadre pour une évolution contrôlée d'applications à base de composants ; ce contrôle garantissant la préservation de certaines propriétés architecturales et par là d'attributs qualité.*

ABSTRACT. *Applications should evolve to respond to changing client requirements and their evolving environments. These changes, often unforeseen, done by tiers and in urgency, lead sometimes the software to a state initially undesired. We present, in this paper, a framework to preserve the consistency of component-based applications during their evolution. This control aims at preservation of some desired architectural properties and thus some desired quality attributes.*

MOTS-CLÉS : *Génie logiciel basé composants, Contrats d'évolution, Maintenance préventive, Attributs qualité*

KEYWORDS: *Component-based software engineering, Evolution contracts, Preventive software maintenance, Quality attributes*

1. Introduction

Une caractéristique intrinsèque d'un logiciel, représentant une activité du monde réel, est la nécessité d'évoluer pour satisfaire de nouvelles exigences (fonctionnelles, d'adaptation ou de perfectionnement). La première loi de Lehman, issue de constatations sur le terrain, stipule ainsi qu'un logiciel doit nécessairement évoluer faute de quoi il devient progressivement inutile [LEH 85]. En conséquence, l'activité de maintenance, dont l'évolution constitue une grande part, représente une part trop importante du chiffre d'affaire des sociétés développant des logiciels (de 50 à 80% selon les études [LIE 80, MCK 84, NOS 90]). Il est donc nécessaire de proposer des méthodes, des techniques et des outils facilitant ces activités tout en diminuant leur coûts.

Un des axes majeurs de recherche de ces dernières années a été de répondre au problème posé par la propagation des changements : garantir qu'une modification faite à un endroit n'affecte pas à tort une autre partie de l'application. La détection et l'analyse de ce type de conflits portent à ce jour uniquement sur des aspects syntaxiques (par exemple, le problème classique de la fragilité de la classe de base : si une classe évolue, toutes les classes qui héritent de celle-ci doivent suivre l'évolution) et fonctionnels (par exemple, si les pré/post-conditions d'une méthode sont renforcées ou affaiblies, toutes les méthodes qui dépendent de cette dernière doivent évoluer en conséquence) [STE 96, RAU 00, RAJ 99].

Dans cet article, nous mettons en valeur l'intérêt d'étendre ce type d'étude à des aspects non fonctionnels, en particulier à ceux portant sur les propriétés architecturales d'un composant. En effet, le choix d'un type d'architecture donnée est dicté par le souci de respecter les exigences qualité établies lors de l'étape de spécification [CLE 02, BAS 03]. Une analyse et conception architecturales dirigées par les attributs qualité [KLE 99, BAS 01], selon le modèle défini par le SEI[1] [CLE 02, BAS 00], permettent entre autres de garantir ces exigences qualité lors du développement. Par contre, le maintien d'un choix architectural, lors d'une évolution, est un enjeu majeur malheureusement rarement constaté.

Dans la partie qui suit, nous illustrons, à l'aide d'un exemple, l'intérêt du maintien de choix architecturaux. Nous introduisons, ensuite, une solution à ce problème utilisant un mécanisme contractuel. Enfin, un outil support pour cette solution est décrit.

2. Problématique à travers un exemple

L'exemple de la figure 1 représente une vue fonctionnelle simplifiée de l'architecture d'un système de contrôle d'accès à un bâtiment (cas d'un musée). Cette architecture respecte le patron *pipe & filter* : le système reçoit en entrée des données permettant l'authentification d'un usager. Après identification, l'information est envoyée au composant contrôleur d'accès. A cette information sont ajoutées d'autres données (l'heure

1. SEI : Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA

d'entrée, la galerie dans le musée, etc.) puis passées au composant responsable de l'archivage local (composant `Logging`). Ces mêmes informations sont ensuite transmises (composant `Transmission`), via le réseau à un serveur central d'archivage de l'entreprise qui s'occupe de la sécurité du musée. A travers le choix d'une architecture en pipe, les développeurs de ce système ont cherché à respecter les exigences de maintenabilité formulées dans le document de spécification. En effet, l'architecture en pipe est celle qui offre le couplage le plus faible entre des composants.

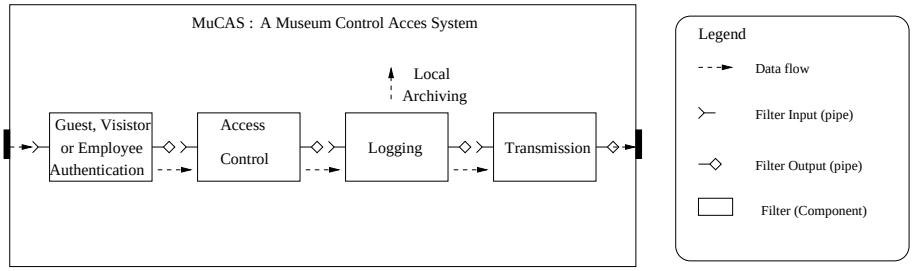

Figure 1. *Architecture simplifiée d'un système de contrôle d'accès à un musée*

Supposons maintenant, que pour des raisons de maintenance perfective, il soit décidé que l'archivage local de certaines informations (voir composant `logging`) est désormais inutile et qu'il faille les transmettre directement au serveur central. Les personnes en charge de cette modification peuvent décider de créer un lien direct entre le composant contrôleur d'accès et le composant responsable de la transmission au serveur central. Le composant `Access Control` se retrouve avec deux liens : le premier avec le composant `Logging`, pour le flux non affecté par la modification ; le second avec le composant `Transmission` pour les données à transmettre directement (voir figure 2). Cette modification fait perdre à l'application les bénéfices de l'architecture en pipe et par conséquent abaissent son niveau de maintenabilité.

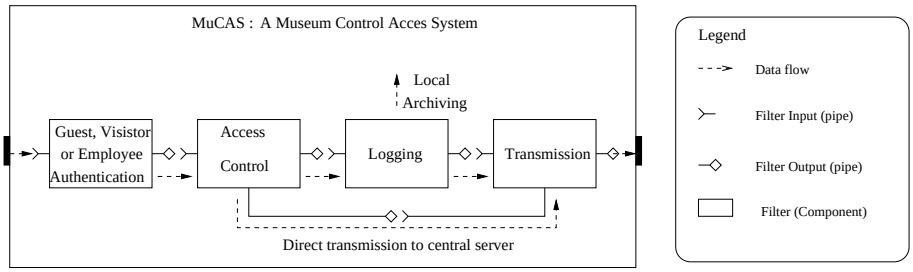

Figure 2. *Architecture du système de contrôle d'accès à un musée après modification*

Le problème, illustré à travers l'exemple ci-dessus, a plusieurs origines : tout d'abord, les modifications faites à un logiciel sont souvent réalisées à distance de la précédente version et généralement par des tiers ; ensuite, les raisons des choix architecturaux effectués sont rarement explicités. Même dans le cas où ils le sont, le code est souvent le seul artefact modifié et parcouru sans considération pour les documentations associées. Un réflexe acquis par la plupart des développeurs est donc de n'accorder du crédit, malheureusement avec raison, qu'au seul code source.

3. Notre solution : Le contrat d'évolution

Afin de garantir la permanence de choix architecturaux, nous proposons non seulement de les expliciter de manière formelle mais également de vérifier leur respect lors de chaque évolution. Nous introduisons pour cela la notion de **contrat d'évolution**. Ce contrat constitue un accord entre deux parties. D'un côté, l'architecte du composant, qui exprime les propriétés architecturales dont il souhaite la permanence afin de garantir le respect de certaines exigences non fonctionnelles formulées dans le document de spécification. De l'autre côté, la personne conduisant l'évolution qui doit se conformer au contrat pour ainsi maintenir le niveau des exigences non fonctionnelles.

Il est important de noter que ces exigences non fonctionnelles et donc les propriétés architecturales qui leur sont attachées peuvent avoir pour origine la politique qualité de l'entreprise (détaillée dans son manuel qualité), des impératifs propres à une ligne de composants ou à un produit particulier (exprimés dans le plan qualité associé à ce composant). Dans les deux premiers cas on constate qu'un contrat d'évolution peut être écrit en dehors de tout projet de développement particulier et être imposé par la suite à certains projets de développement avant même leur démarrage.

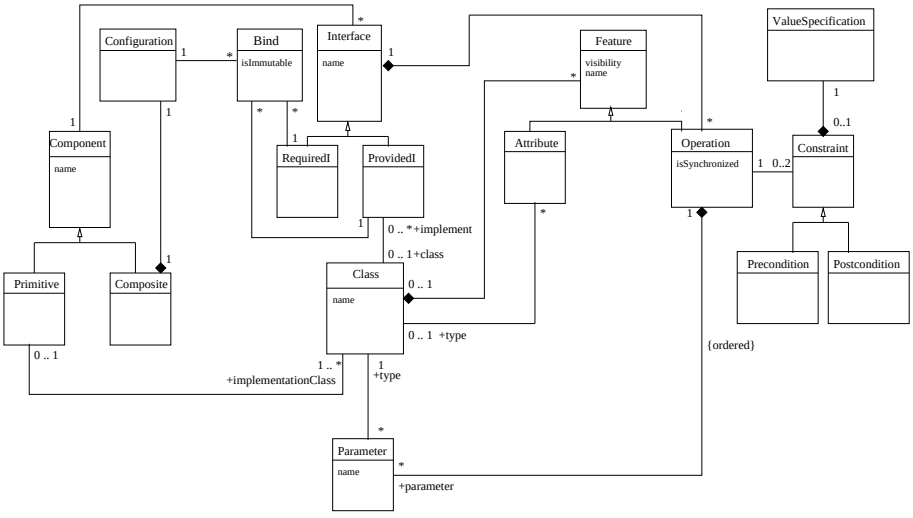

Figure 3. *Le méta-modèle décrivant l'architecture d'un composant*

On utilise pour définir de tels contrats une notation formelle s'appuyant sur un langage existant et largement adopté dans le monde objet, le standard OCL [OMG 03] de l'OMG. Ce langage permet l'expression de contraintes sur des modèles objets exprimés dans des méta-modèles décrits en MOF (Meta Object Facility). L'originalité de nos travaux réside dans l'application de ce langage dans le contexte du modèle en parcourant son méta-modèle. Le méta-modèle décrivant ce qu'est une architecture à base de composants est explicité par la figure 3. Nous introduisons de ce fait un mécanisme d'introspection qui va permettre de spécifier, pour un composant, des contraintes portant sur son architecture.

Revenons sur l'exemple présenté dans la section 2. La définition d'une contrainte stipulant le fait qu'une architecture en pipe doit être maintenue, lors de chaque évolution, peut être traduite sous la forme du contrat suivant :

```
context MuCAS
inv: self.Configuration.isArchSimple() and
     self.Configuration.isArchConnexe() and
     ((self.Configuration.Bind->size()) =
     (((self.Configuration.Bind.RequiredI.Component->
     union(self.Configuration.bind.ProvidedI.Component)))->size() - 1))
     and ((self.Configuration.Bind->size()) =
     (((self.Configuration.Bind.RequiredI.Component->
 intersection(self.Configuration.Bind.ProvidedI.Component)))->size() + 1))
```

Cet exemple montre qu'une contrainte s'exprime dans le contexte d'un composant (ici le composant nommé MuCAS), en usant du mécanisme de navigation d'OCL appliqué au méta-modèle. Par exemple, `self.Configuration.Bind` désigne, pour le composant MuCAS, la liste des liens entre ses sous-composants.

Afin de rendre l'expression du contrat plus facile et rapide, nous avons introduit de nouveaux opérateurs tels que isArchSimple() ou isArchConnexe() (voir l'exemple précédent). Nous sommes allés encore plus loin en introduisant des opérateurs du genre isArchPipe(). La contrainte précédente peut, donc, s'écrire simplement de la manière suivante :

```
context MuCAS
self.Configuration.isArchPipe()
```

Nous avons validé cette approche à travers des outils, que nous vous présentons dans la section suivante.

4. Outils supports pour les contrats d'évolution

Le support logiciel des contrats d'évolution se compose de trois outils.

– un vérificateur de conformité : il vérifie la conformité du modèle de l'application par rapport au méta-modèle,

– un éditeur de contrats : il permet la rédaction d'un contrat et sa validation en se basant sur la grammaire OCL (adaptée) et le méta-modèle dans lequel navigue cette contrainte (fourni au format XMI). Pour réaliser cet outil, nous sommes partis d'OCL Compiler [DRE 02], que nous avons adapté à nos besoins,

– Évaluateur de contrats : il reçoit en entrée le contrat d'évolution et les descriptions des architectures de l'ancienne et la nouvelle version du composant, au format XML usant d'une DTD représentant notre méta-modèle. Il valide ou non la nouvelle version du composant. Dans ce dernier cas, l'outil fournit le détail de la non conformité.

5. Conclusion

Notre approche consiste à expliciter les propriétés architecturales d'un logiciel afin de lui garantir une évolution cohérente. Cette proposition répond au problème soulevé par l'évolution non-anticipée sur le plan architectural. Nos travaux se rapprochent très fortement de la maintenance préventive. Nous pouvons par exemple empêcher l'accroissement de la complexité d'un composant au fil de ces évolutions.

Dans un futur proche, nous projetons de cataloguer toutes les propriétés architecturales utiles de manière à enrichir OCL avec de nouveaux opérateurs. Nous étudierons, par la suite, la composition de contrats et le problème sous-jacent relatif à la cohérence inter-contrats. En effet, lors de l'assemblage de composants (chacun avec son contrat), plusieurs contraintes dans différents contrats peuvent entrer en conflit. Nous essayerons de fournir des moyens pour détecter ce type de conflit et proposer des mécanismes de recouvrement. A plus long terme, le contrat d'évolution nous apparaît comme un moyen permettant de définir et de vérifier un typage plus fort dans le monde des composants allant au delà des signatures (comme dans les langages typés tel que JAVA) et des contraintes fonctionnelles (comme dans le langage EIFFEL avec les pré, post-conditions et invariants) car prenant en compte des contraintes non fonctionnelles.

6. Bibliographie

[BAS 00] BASS L., KLEIN M., BACHMANN F., « Quality Attribute Design Primitives », Notes techniques cmu/sei-2000-tn-017, décembre2000, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

[BAS 01] BASS L., KLEIN M., BACHMANN F., « Quality Attribute Design Primitives and the Attribute Driven Design Method », *Proceedings of the 4th International Workshop on Product Family Engineering*, Bilbao, Spain, octobre2001.

[BAS 03] BASS L., CLEMENTS P., KAZMAN R., *Software Architecture in Practice, 2nd Edition*, Addison-Wesley, avril2003.

[CLE 02] CLEMENTS P. KAZMAN R. K. M., *Evaluating Software Architectures, Methods and Case Studies*, Addison-Wesley, 2002.

[DRE 02] DRESDEN T. U., « OCL Compiler web site », http ://dresden-ocl.sourceforge.net/, 2002.

[KLE 99] KLEIN M., KAZMAN R., « Attribute-Based Architectural Styles », Rapport interne cmu/sei-99-tr-022, octobre1999, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

[LEH 85] LEHMAN M., BELADY L., *Program Evolution : Process of Software Change*, London : Academic Press, 1985.

[LIE 80] LIENTZ B., SWANSON E., *Software Maintenance Management*, Reading MA : Addison-Wesley, 1980.

[MCK 84] MCKEE J., « Maintenance as Function of Design », *Proceedings of AFIPS National Computer Conference*, Reston, Virginia, USA, 1984, p. 187–193.

[NOS 90] NOSEK J., PALVIA P., « Software Maintenance Management : Changes in the last decade », *Journal of Software Maintenace*, vol. 2, n° 3, 1990, p. 157–174, John Wiley & Sons, Inc.

[OMG 03] OMG, « UML 2 OCL Final Adopted Specifi cation », Object Management Group web site : http ://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML, octobre2003.

[RAJ 99] RAJLICH V., « Modeling software evolution by evolving interoperation graphs », *Annals of Software Engineering*, vol. 9, 1999, p. 235–248.

[RAU 00] RAUSCH A., « Software evolution in componentware using requirements/assurances contracts », *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, juin2000, ACM Press, p. 147–156.

[STE 96] STEYAERT P., LUCAS C., MENS K., D'HONDT T., « Reuse contracts : managing the evolution of reusable assets », *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Jose, California, USA, 1996, ACM Press, p. 268–285.