

# Extraction d'architecture et de composants de codes patrimoniaux

Jean-Claude Royer, Professeur d'informatique  
Equipe ASCOLA, Mines de Nantes-INRIA, LINA  
4 rue Alfred Kastler, 44307 Nantes cedex 3, France  
`Jean-Claude.Royer@emn.fr`

10 février 2010

La programmation par composants propose d'améliorer la qualité interne des applications logicielles en rendant explicite leur architecture et en accroissant la modularité de leurs briques de base. Les propositions de langages ou les cadres logiciels utilisés couramment sont encore loin des langages académiques en ce qui concerne l'utilisation de hiérarchies complexes, les propriétés comme l'intégrité des communications ou encore le sous-typage. De plus beaucoup d'applications sont encore écrites sans avoir recours aux composants. Il y a donc plusieurs raisons pour extraire une architecture et les informations de composant d'un code patrimonial. Le code n'a pas du tout été pensé en termes de composants ou pas avec la rigueur voulue et une évolution du code vers une structure propre à composants est souhaitable. Un code même si il utilise des composants peut avoir dérivé de son architecture suite aux évolutions et le code n'est plus conforme à l'architecture initiale. Pour remédier à ces problèmes une approche est d'analyser le code source et d'en extraire les informations concernant les composants et l'architecture. Il est ensuite possible de calculer la différence avec l'architecture originale ou supposée, comme dans l'approche "réflexive" de Murphy et al. [7]. Ceci permet aux concepteurs d'élaborer une restructuration en composants de leurs codes en suivant une approche itérative, incrémentale et guidée par l'architecture voulue.

Dans l'optique de mesurer la qualité d'un code ou pour le restructurer en composants il est nécessaire d'avoir des mesures et des outils supports. Toutefois les outils d'analyse actuels reposent sur des approches par métriques. Par exemple [6, 5] se basent sur des mesures de couplage et de cohésion, et leurs combinaisons. Ces approches ont certains avantages comme une relative indépendance avec l'implémentation sous-jacente des composants. Elles ne sont pas forcément bien adaptées à du code patrimonial et demande

un calibrage des métriques. Elles ne prennent généralement pas en compte des propriétés fines des composants comme l'intégrité des communications ou le sous-typage. Nous pensons qu'une approche centrée autour du principe d'intégrité des communications peut permettre d'autres avantages complémentaires [4].

Pour ce faire nous proposons l'étude d'un langage concret comme Java et l'élaboration des règles d'extraction des composants. Ces règles posent plusieurs défis :

- Introduire des règles de typage saines mais pas trop contraignantes en pratique.
- Intégrer dans une telle analyse les bibliothèques existantes qui ne sont pas à composants et qui ne le deviendront peut-être jamais.
- Les types génériques demandent une attention particulière.
- Les classes internes et quelques autres constructions comme les exceptions ou les tableaux vont demander des règles spécifiques.
- Obtenir un ensemble cohérent de règles et utile en pratique.

On ne peut pas évoquer l'intégrité des communications sans faire référence aux travaux autour de ArchJava [2, 1] et aussi d'AliasJava [3]. La différence essentielle est d'établir un jeu de règles d'inférence permettant de séparer les types et signatures du code Java pour distinguer les vrais composants des types de données. Une comparaison avec l'utilisation d'approches basées sur les métriques devrait montrer les complémentarités et l'intérêt d'un couplage de ces deux types d'approches. En plus de la production de résultats conceptuels, un travail d'expérimentation non négligeable sur des études de cas est à faire. Un résultat concret de ce travail est l'implémentation d'un prototype mettant en œuvre les règles étudiées et pouvant servir de base à la restructuration de codes patrimoniaux en code à composants.

Ce travail s'inscrit dans le cadre du projet de l'équipe ASCOLA. Cette équipe s'intéresse à la définition de constructions avancées pour la structuration des applications, aux transformations et aux évolutions d'applications de grande taille. L'équipe se concentre toutefois sur les aspects langages aussi bien au niveau architectural qu'au niveau implémentation et exploite des techniques d'analyse statique, de sémantique formelle et de vérification de propriétés.

## Références

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367. Springer Verlag, 2002.
- [2] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference*

*on Software Engineering (ICSE-02)*, pages 187–197. ACM Press, 2002.

- [3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 311–330, New York, November 4–8 2002. ACM Press.
- [4] Pascal André, Nicolas Anquetil, Gilles Ardourel, Jean-Claude Royer, Petr Hnetynka, Tomas Poch, Dragos Petrascu, and Vladuela Petrascu. Javacompext : Extracting architectural elements from java source code. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), tool demonstration*, pages 377–378, Lille, France, October 2009.
- [5] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, and Dalila Tamzalit. Extraction of component-based architecture from object-oriented systems. In *WICSA*, pages 285–288. IEEE Computer Society, 2008.
- [6] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse engineering software-models of component-based systems. In *CSMR*, pages 93–102. IEEE, 2008.
- [7] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models : Bridging the gap between design and implementation. *IEEE Computer Society Transactions on Software Engineering*, 27(4) :364–380, April 2001.