



Programmation de Composants Interfaçables

Luc Fabresse

► To cite this version:

Luc Fabresse. Programmation de Composants Interfaçables. JOCM'04: Journées du Groupe Objets Composants et Modèles, Mar 2004, Lille, France. pp.19-25. lirmm-00108645

HAL Id: lirmm-00108645

<https://hal-lirmm.ccsd.cnrs.fr/lirmm-00108645>

Submitted on 23 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Programmation de composants interfaçables

Luc Fabresse

L.I.R.M.M – 161, rue Ada
34392 Montpellier Cedex 5 – FRANCE –
+33 4 67 41 85 33

RÉSUMÉ : *Cet article présente un prototype de langage de programmation de composants interfaçables. Nous y définissons les composants interfaçables comme des entités logicielles dotées de propriétés et assemblables entre elles, selon une logique “métier” mais qui en plus sont dotées de caractéristiques spécifiques permettant de les assembler de façon semi-automatique avec des composants d’interfaçage. Nous présentons un nouveau langage (ICL) permettant de développer de tels composants, de les assembler et de les interfacier avec le logiciel dédié *ThingsTM* de la société Gatonero. L’article décrit ce langage, sa mise en oeuvre avec *OpenJava* et la connexion des programmes avec *ThingsTM*.*

MOTS-CLÉS : *Composants Logiciels, Interface Homme-machine, Interfaçage semi-automatique, Java, Réflexivité.*

1 Introduction

Dans le cycle de développement d’une application, la réalisation de l’interface Homme-machine (IHM) est une tâche longue et coûteuse car de nombreux facteurs complexes comme, l’ergonomie [1] qui contribue largement au succès du logiciel, la maintenance, les possibilités d’évolution, la facilité de portage, sont à prendre en compte. La prise en compte de ces facteurs a conduit à l’invention de modèles architecturaux comme MVC (Model View Controller) ou PAC (Présentation Abstraction Contrôleur) grâce auxquels il est notamment possible de séparer, dans le code des applications, la partie métier de la partie IHM. Dans cette optique de “separation of concerns” [2], on essaye aujourd’hui de n’écrire que le code de la partie métier des applications et d’utiliser des outils pour produire semi-automatiquement la partie IHM. Par exemple, l’outil d’interfaçage *ThingsTM*, développé par la société Gatonero SA produit des interfaces pour des programmes écrits en Java. Nous réalisons avec cette société une étude visant à proposer de nouveaux langages à base de composants permettant d’écrire des programmes plus simples à interfacier et dont les interfaces seraient de meilleure qualité sémantique que celles obtenues par analyse de programme Java.

La section 2 rappelle les deux principales approches actuelles pour interfacier semi-automatiquement des applications et positionne l’outil *ThingsTM* dans cette catégorisation. Une spécification et une modélisation de la notion de composant interfaçable sont développées en section 3. La section 4 décrit notre proposition de nouveau langage (ICL) pour écrire de tels composants. Cette section présente aussi la réalisation d’un programme à base de composants interfaçables écrits en ICL et l’interface utilisateur produite par *ThingsTM*. La mise en oeuvre en *OpenJava* du langage ICL est brièvement décrite en section 4.2. Finalement, nous concluons par une synthèse et la présentation des suites les plus intéressantes à donner à ce travail.

2 Différentes approches pour l'interfaçage semi-automatisé

Deux grandes approches peuvent être comparées pour interfacier semi-automatiquement des applications.

La première approche vise à proposer des langages et des outils de description abstraite des IHM, descriptions qui sont ensuite transformées en code en fonction d'un langage et d'une architecture cibles. Un exemple, dans cette approche, est le langage de description SUNML (Simple Unified Natural Markup Language) développé au sein du projet RainBow [3].

La seconde approche consiste à générer l'IHM d'une application à partir, d'une part de l'utilisation d'un schéma général d'interface, et d'autre part, de l'étude d'une description de sa partie métier pouvant se présenter sous différentes formes : diagrammes UML, code source ou compilé. Par exemple, les outils tels que *Génova* [4], *D•OM* [5] et *Pollen* [5] analysent les diagrammes UML décrivant la partie métier d'une application et produisent le code source d'une IHM, et SmartTools [6] génère des environnements de développement par analyse de la syntaxe abstraite d'un langage.

Le logiciel *ThingsTM* que nous utilisons, se classe dans cette catégorie. *ThingsTM* analyse des exécutables Java représentant la partie métier d'une application afin d'en déduire des informations d'interfaçage. Ces informations peuvent être enrichies manuellement via un fichier XML et sont utilisées pour paramétrer le schéma d'interface "générique" intégré à *ThingsTM*. C'est ainsi que toutes les interfaces produites par *ThingsTM* présentent le même schéma d'interface découpé en zones (arbre d'exploration, zone de saisie et affichage spécifique) et paramétré pour une application métier lors de l'interfaçage.

3 Proposition d'un modèle de composants interfaçables

Pour rendre des programmes à base de composants plus simple à interfacier et améliorer la qualité de l'IHM produite de façon semi-automatique, nous avons étudié et modélisé la notion de composant interfaçable.

3.1 Notion de composant interfaçable

Notre réflexion [7] nous a conduit à distinguer les concepts suivants :

Composant : Brique logicielle configurable permettant la construction d'une application par assemblage [8].

Composant métier : Composant issu de la partie métier d'une application.

Élément d'interface : Tout élément susceptible d'être utilisé pour l'interfaçage graphique d'un composant.

Composant interfaçable : Un composant métier à la description duquel ont été ajoutés un ou plusieurs éléments d'interface.

3.2 Un modèle de composants interfaçables

Le modèle de composant que nous proposons [7] pour mettre en oeuvre la notion de composant interfaçable est inspiré de ceux de Corba [10] et de Java Bean [11]. Du modèle Corba,

nous avons retenu les notions, de facette [9] qui permettent de différencier des points de vue sur les composants, et de puits et sources d'évènements qui permettent d'assembler les composants. Du modèle implicite des Java Beans, nous avons extrait la notion de propriété, “une unité sémantique publique qui affecte l'apparence ou le comportement d'un *JavaBean*” [11], et dégagé différents “types” de propriétés. Par exemple, les propriétés liées sont des propriétés dont la modification de valeur entraîne la levée d'un événement. Nous avons ensuite intégré toutes les propriétés indispensables pour l'interfaçage dans notre modèle. Les composants interfaçables sont aussi dotés d'éléments d'interfaces qui peuvent être visuels, sonores, dédiés à un outil d'interfaçage, etc.

En synthèse, nous proposons un modèle dans lequel un composant est explicitement constitué de :

- propriétés, et plus généralement, de propriétés ayant un ensemble de caractéristiques, soit liées à la communication et à l'assemblage (notification de modification, etc), soit liées à l'interfaçage (visibilité, etc),
- d'éléments d'interfaces qui peuvent être dédiés à un outil d'interfaçage.

4 Proposition d'un langage de composants interfaçables

Cette section présente le langage ICL qui permet d'écrire des programmes à base de composants interfaçables.

4.1 Présentation par l'exemple

Plutôt que de présenter notre langage de façon abstraite, nous l'illustrons au travers d'un exemple présenté par la figure 1 qui montre le code ICL déclarant un composant *Book* représentant un livre dans une application de gestion de bibliothèque.

<pre> Component Book { /** The title of this book */ property viewable inout String title; /** The author(s) of this book. */ property viewable inout Vector authors collection of Author; /** a specific book representation */ ihmelement Preview imageCouverture { component = "ImageViewer"; }; [...] } </pre>	<pre> public class Book { java.lang.String title; private PropertyChangeSupport propSupp = new PropertyChangeSupport(this); public String getTitle() { return title; } public void setTitle(String newTitle) { String oldTitle = getTitle(); title = newTitle; propSupp.firePropertyChange("Title", oldTitle, getTitle()); } [...] } </pre>
--	---

FIG. 1 – Extrait du code ICL du composant *Book*

FIG. 2 – Extrait de la traduction du code Icl de *Book* en Java

Le mot-clé *property* permet la déclaration de propriétés et *Book* contient ainsi les propriétés *title* et *authors*. Elles sont déclarées *inout* c'est-à-dire accessibles en lecture et écriture. Le mot-clé *viewable* indique que la propriété peut être visualisée. Une propriété *viewable*

est obligatoirement une propriété liée. La figure 2 montre le code Java produit par le compilateur ICL pour la propriété *title* de *Book*.

Le composant *Book* est aussi doté de l'élément d'interface *imageCouverture* de type *Preview*. Ce type *Preview*, défini dans le modèle des éléments d'interface, représente un élément d'interface visuel dédié à l'outil d'interfaçage *ThingsTM*. Le paramètre de création *component* indique que le composant ICL *ImageViewer* est capable de créer une "widget" dessinant l'image de la couverture d'un composant *Book*. La "widget" produite pourra alors être intégrée par *ThingsTM* au sein d'une IHM.

4.2 Un exemple de programme et connexion avec *ThingsTM*

Nous avons programmé en ICL, les composants métiers d'une application simplifiée de gestion d'une librairie constituée de livres, d'auteurs, d'utilisateurs, etc. La figure 3 montre l'interface utilisateur produite par *ThingsTM* après traduction du code ICL (cf. figure 1) et analyse du code Java résultat (cf. figure 2) par *ThingsTM*.

Les plus-values apportées par notre langage sont d'une part au niveau du code métier qui a été simplifié par l'expression directe de propriétés et d'autre part au niveau de l'interface. Dans la fenêtre intitulée "Books - All", le champ titre du livre sélectionné est représenté par un champ texte car la propriété titre du composant *Book* a été déclarée *viewable*. Ce champ texte est éditable car cette propriété a été caractérisée d'*inout* dans le code. Si un programme tiers modifie le titre d'un livre, le changement sera immédiat sur l'interface grâce à la possibilité de déclarer explicitement qu'une propriété d'un composant ICL est liée. La déclaration de l'élément d'interface *imageCouverture* permet de représenter les livres par l'image de leur couverture dans certaines zones de l'interface.

5 Implantation

Les quelques indications suivantes décrivent notre implantation d'ICL en *OpenJava* [12]. Cela permet de mieux comprendre le fonctionnement global de l'interfaçage d'une application ICL avec notre prototype. Un composant interfaçable est implanté par une méta-classe *OpenJava* qui en définit la structure et les traitements nécessaires pour obtenir une classe Java standard. Les propriétés et les éléments d'interface présentés en section 3.2 ont aussi été implémentés par des méta-classes *OpenJava*. Pour traduire le code ICL de nos composants en Java, nous avons mis en place une *chaîne de compilation* composée :

- d'un *traducteur de code source ICL vers OpenJava* qui ajoute dans le code ICL des informations telles que la méta-classe, remplace le mot-clé *class* par *component*, et plus généralement, nous permet de nous affranchir de certaines limitations syntaxiques,
- d'une *version modifiée du compilateur OpenJava* qui traite les fichiers sortis de notre traducteur et invoque nos générateurs de codes,
- de *plusieurs générateurs de codes* qui fournissent une traduction des informations d'interfaçage introduites dans nos composants dans une forme utilisable par *ThingsTM* dans sa version actuelle. C'est ainsi que sont générés, un fichier de contexte, des éléments d'interface composites, etc.

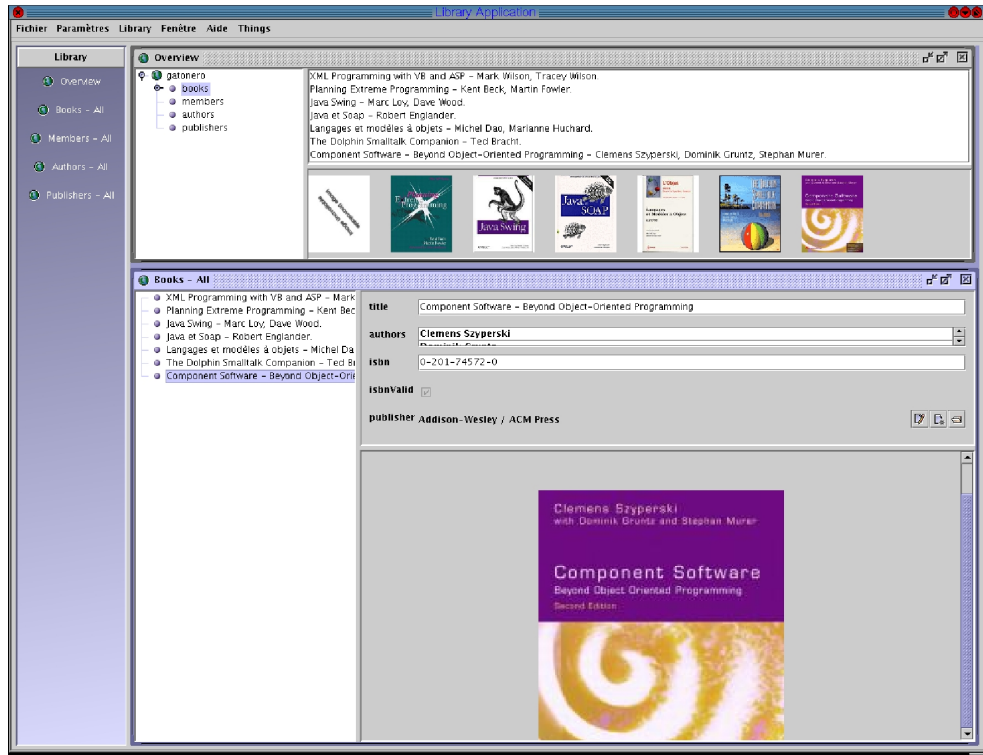


FIG. 3 – Capture d’écran du résultat de l’interfaçage automatique par *ThingsTM* d’une application de gestion de bibliothèque écrite en ICL.

6 Conclusion et Perspectives

Face à la problématique du développement des interfaces utilisateurs, nous avons étudié : le domaine de l’interfaçage automatique dont le logiciel *ThingsTM* et les composants logiciels. A partir de ces études, nous avons donné une spécification de la notion de composant interfaçable et proposé le langage ICL pour écrire de tels composants. Ce langage intègre un modèle de composant interfaçable, une syntaxe et des mécanismes spécifiques permettant la connexion des applications écrites dans ce langage avec un outil d’interfaçage du type de *ThingsTM*. Un prototype du langage ICL a été programmé en OpenJava. Une “chaîne de compilation” a été mise en oeuvre pour traduire le code ICL de composants interfaçables en code Java. Finalement, nous avons programmé en ICL la partie métier d’une application et réalisé son interfaçage par *ThingsTM*. L’interface produite montre la plus value apportée par la programmation en ICL pour l’automatisation de l’interfaçage.

Deux grands axes intéressants pour la suite de ce travail sont possibles :

L’interfaçage : un service orthogonal ? Dans l’optique de séparation des aspects, l’IHM d’un composant pourrait être considéré comme un aspect orthogonal au même titre que la persistance ou la sécurité et être rendu par les plates-formes d’exécution (e.g J2EE) qui intégreraient alors un composant d’interfaçage du type de *ThingsTM* et réaliserait l’interfaçage automatique des composants par assemblage dynamique.

Intégration à l’approche MDA (Model Driven Architecture) qui vise à définir les composants sous forme de modèle indépendant d’une plate-forme spécifique. Cette indépendance est d’autant plus importante pour les informations d’interfaçage.

Références

- [1] D. Scapin & al. Methodes pour l'ergonomie des logiciels interactifs. Technical report, Rapport de recherche, INRIA, 2001.
- [2] W. Hürsch et C. Videira Lopes. Separation of concerns. Technical report, Rapport de recherche, Northeastern University, 1995.
- [3] A-M. Pinna et M. Riveill J. Fierstone. Architecture logicielle pour l'adaptation et la composition d'ihm - mise en oeuvre avec le langage sunml. Technical report, Rapport de recherche, Laboratoire I3S — Université de Nice-Sophia Antipolis —, janvier 2003. <http://rainbow.essi.fr>.
- [4] Proceedings of NWPER'98 Nordic Workshop on Programming Environment Research, Bergen. *Incorporating Rapid User Interface Prototyping in Object-Oriented Analysis and Design with Genova*, Juin 1998.
- [5] S. Wouters. Analyse comparative de générateurs d'interfaces d'applications de gestion à partir d'un diagramme de classes uml. Master's thesis, Université catholique de Louvain, département d'Administration et de Gestion, 2002.
- [6] International Conference on Compiler Construction CC'01. *SmartTools : a Generator of Interactive Environments Tools*, Avril 2001. <http://www-sop.inria.fr/oasis/SmartTools/>.
- [7] L. Fabresse. Proposition d'un langage de programmation à base de composants logiciels interfaçables et application à la génération semi-automatisée d'ihm. Technical report, Rapport de DEA, LIRMM, 2003.
- [8] C. Szyperski. *Component Software : Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.
- [9] *Manual of Corba Component Model V3.0*, 2002. <http://www.omg.org/technology/documents/formal/components.htm>.
- [10] C. Gransart et P. Merle J.-M. Geib. *Corba, des concepts à la pratique*. InterEditions, 1997.
- [11] Sun. *JavaBeans^TM*, 1997. <http://java.sun.com/products/javabeans/docs/spec.html>.
- [12] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava : A class-based macro system for java. In *OORaSE*, pages 117–133, 1999.