Vers des Composants Logiciels Interfaçables

Luc Fabresse, Christophe Dony, Marianne Huchard, Olivier Pichon

Vers des composants logiciels interfaçables

Luc Fabresse[1], Christophe Dony[1], Marianne Huchard[1], Olivier Pichon[2]

[1] L.I.R.M.M – 161, rue Ada
34392 Montpellier Cedex 5 – FRANCE –
{fabresse,dony,huchard}@lirmm.fr

[2] Gatonero SA
Espace Innovation 2 – Parc Georges Besse
110, allée Charles Babbage
30000 Nîmes
op@gatonero.com et http://www.gatonero.com

Résumé : Cet article présente une étude relative aux domaines du développement à base de composants et de la production semi-automatisée d'interface homme-machine (IHM). Dans ce contexte, nous étudions le concept de composant interfaçable, c'est-à-dire une entité logicielle assemblable selon une logique métier mais également dotée de caractéristiques qui faciliteraient la production semi-automatique d'IHM par des outils d'interfaçage. L'article propose et présente un modèle de composants interfaçables et un langage (ICL) permettant de développer de tels composants et de les assembler. ICL a été couplé avec l'outil d'interfaçage $Things^{TM}$ [1] afin de réaliser un exemple complet d'application développée en ICL et dont l'IHM a été produite semi-automatiquement par $Things^{TM}$.

Mots-clés : Langage à composants[2], génération d'IHM, AOP, Java, Réflexivité.

1 Introduction

Un des points clé actuel dans les recherches en génie logiciel est l'étude du développement d'applications par assemblage de composants. On étudie aujourd'hui des mod`eles de composants (*Java Beans* (Sun, 1997), CCM (OMG, 2002), EJB (Blevins, 2001), Fractal (Coupaye & Stefani, 2002)), des langages pour les décrire ou les connecter (Langages de Description d'Architecture (Medvidovic & Taylor, 1997)) et/ou des architectures (J2EE, Fractal) pour les assembler. Cet intérêt pour les composants résulte aussi bien de la volonté de réduire les coûts de développement en augmentant la réutilisation que de la nécessité d'inventer de nouvelles formes de développement pour prendre en

[1] Logiciel d´evelopp´e par la soci´et´e Gatonero.

[2] En accord avec l'expression « un langage `a objets », bien que l'on rencontre souvent l'expression « un langage de composants » dans la litt´erature.

compte la complexité structurelle sans cesse croissante des applications. Doter les applications d'interfaces homme-machine (IHM) sophistiquées est un des nombreux facteurs de cette augmentation de complexité.

On essaye ainsi de séparer, dans les spécifications et le code des applications, la *partie métier* et la *partie IHM* grâce `a des mod`eles architecturaux tels que MVC (*Model View Controler*) ou PAC (Présentation Abstraction Contrôleur). Des recherches plus récentes vont plus loin et proposent des outils pour produire le plus automatiquement possible les IHM en utilisant une description de la partie métier des applications. Par exemple, *Génova* (Arisholm, 1998), D•OM (Wouters, 2002) et *Pollen* (Wouters, 2002) utilisent des diagrammes UML tandis que $Things^{TM}$ [1] utilise du code Java compilé. Avec tous ces outils, la production semi-automatique d'une IHM est conditionnée par le choix préalable d'un *schéma d'interfaçage*. Un tel schéma comprend la structure de l'IHM `a produire en terme de fenêtres et de menus, ainsi que des r`egles d'affichage qui associent `a certains éléments métiers une représentation graphique. Une r`egle peut par exemple spécifier qu'un entier se représente par un champ texte contenant sa valeur. Chaque outil int`egre un ou plusieurs schémas d'interfaç age qui lui sont propres. Ce processus semi-automatique de production d'IHM n'est par conséquent ni générique, ni universel.

Dans cette étude, nous utilisons l'outil $Things^{TM}$ qui analyse le bytecode Java[3] de la partie métier d'une application et fabrique une IHM selon un *schéma d'interfaçage* prédéfini. Ce schéma, défini par étude de nombreux cas, structure une IHM en une fenêtre principale dotée d'une barre de menus et contenant des fenêtres internes appelées *vues*. Les menus contiennent des items donnant acc`es aux méthodes publiques trouvées dans les classes de l'application. Chaque vue représente un objet métier `a travers trois zones (cf. figure 1). La premi`ere `a gauche affiche un arbre d'exploration (*TreeView*) qui s'inspire de l'interface courante des lecteurs de mails et des gestionnaires de fichiers. La seconde, en haut `a droite, contient un ensemble de champs de saisie (*FormView*) qui permet l'édition de données sous forme de formulaire. Cette zone, dans le cas d'un lecteur de mails, correspond `a la zone o`u sont représentés les champs To, Cc, Subject. La derni`ere zone située en bas `a droite est dédiée `a l'affichage d'une représentation particuli`ere d'un objet métier (*PreView*). Par exemple, le lecteur de mails est paramétré pour afficher dans cette zone une représentation adaptée du contenu d'un mail suivant qu'il s'agit d'une image, d'un document HTML ou de texte brut. Il est aussi possible d'indiquer des informations supplémentaires manuellement dans un fichier XML appelé *fichier de contexte*, afin d'adapter ou d'améliorer l'IHM produite. La figure 1 illustre le fonctionnement de $Things^{TM}$. On y voit d'abord le code Java d'une classe COMPTEUR (utilisée par $Things^{TM}$ sous forme compilée) puis un fichier de contexte minimal qui précise o`u se trouve le bytecode `a analyser et enfin l'IHM produite par $Things^{TM}$ pour manipuler des compteurs. Dans cette IHM, la zone Preview est vide car aucune représentation particuli`ere n'a été spécifiée pour un compteur.

Tous les outils d'interfaç age actuels tels que $Things^{TM}$ présentent des limites similaires. Il leur est notamment difficile :

1. de détecter quels sont les objets métiers `a représenter sur l'IHM,

[3] Forme compil´ee des programmes ´ecrits en Java.

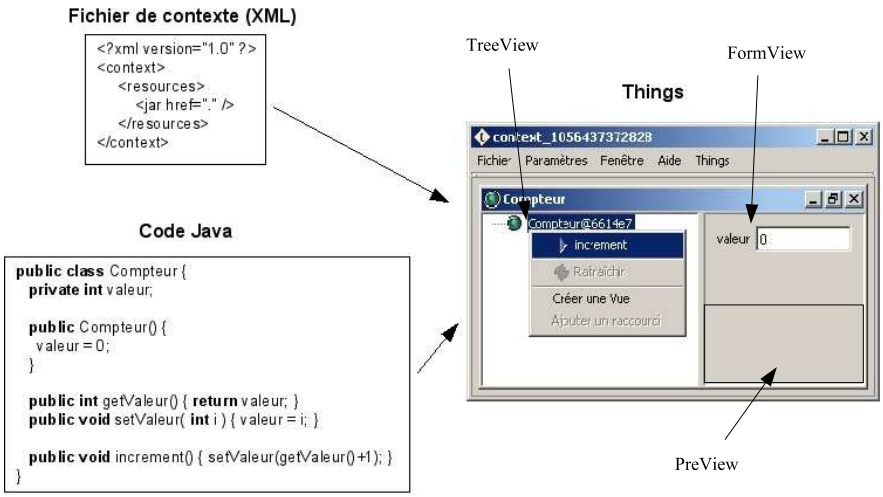

FIG. 1 – Fonctionnement de l'interfaç age semi-automatique avec $Things^{TM}$

2. de représenter les objets métiers d'une faç on spécifique, par exemple un entier sous forme de chiffres romains,

3. d'interpréter les liens de composition entre objets métiers et de produire une représentation des objets composites,

4. de synchroniser les valeurs affichées dans l'IHM avec celles des objets métiers en mémoire.

Notre idée est d'utiliser une approche composants pour développer des applications qui seraient plus simples `a interfacer et dont l'IHM résultante serait de meilleure qualité. Nous proposons donc un mod`ele et un langage originaux de composants dits interfaç ables. En plus d'être composables pour produire des applications métier, les composants interfaç ables sont spécifiquement conç us pour que la partie métier soit semi-automatiquement interfacée.

Cet article s'articule comme suit. La section 2 présente deux mod`eles de composants possédant des caractéristiques intéressantes pour notre étude : le mod`ele développé dans le cadre du projet RainBow et le mod`ele de composants *Java Beans*. La section 3 décrit notre proposition de mod`ele de composant interfaç able. La section 4 présente le langage `a composants ICL qui permet de programmer des applications `a base de composants interfaç ables et décrit son implémentation actuelle en OpenJava. La section 5 détaille un exemple d'utilisation d'ICL et $Things^{TM}$ depuis la programmation jusqu'`a la production semi-automatique d'une IHM. Enfin, la section 6 conclut cet article par une synth`ese et les perspectives les plus notables de ce travail.

2 Modèles de composants

Dans cette section, nous étudions des mod`eles de composants qui présentent des caractéristiques utiles pour définir un langage `a composants facilitant l'interfaç age semi-automatique. Avant de présenter le mod`ele de composants du projet RainBow et celui des *Java Beans*, il nous faut définir la notion de composant car il n'existe pas actuellement de définition unique de ce terme (Szyperski, 2002) (chap 11).

Nous considérons les composants comme des entités logicielles configurables dotées d'un ensemble d'interfaces permettant de les paramétrer, les utiliser et/ou les assembler. Une interface déclare des services fournis ou requis par un composant. Deux composants peuvent être connectés si l'un fournit un service qui est requis par l'autre. Nous définissons un composant métier comme un composant utilisé lors de la réalisation de la partie métier d'une application.

2.1 Les composants dans le projet RainBow

Les recherches menées au sein du projet RainBow s'inscrivent dans le cadre du développement d'applications `a base de composants. Deux catégories de composants y sont proposées : les composants métiers et les composants d'IHM. Un composant métier peut être associé `a plusieurs composants d'IHM. L'un des objectifs de ce projet est de « *permettre la composition et l'adaptabilité des IHM en exploitant les résultats acquis dans le domaine des composants métiers* » (Fierstone *et al.*, 2003). Ceci implique par exemple d'interpréter utilement les liens de composition afin de produire un composant d'IHM pour un composant composite répondant ainsi `a la limitation 3. Pour atteindre ce but, deux langages ont été spécifiés : SUNML et ISL.

SUNML (Simple Unified Natural Markup Language) est un langage de description d'IHM qui permet de décrire la partie abstraite OIA (Objet d'Interactions Abstrait) des composants d'IHM. Un OIA est un arbre obtenu par la réification des informations contenues dans un fichier SUNML. L'OIA est indépendant de toute plate-forme et permet de manipuler dynamiquement une IHM afin de l'adapter ou de l'assembler `a une autre. Pour qu'un composant d'IHM soit utilisable sur une plate-forme donnée, il faut projeter `a l'exécution son OIA vers un OIC (Objet d'Interactions Concret) spécifique `a une architecture. Cette transformation est effectuée par application de r`egles. Par exemple, un nœud *fenêtre* d'un OIA est transformé en nœud *JFrame* dans un OIC lors de la projection en langage Java. Un composant d'IHM est l'union de l'OIA et d'OIC.

ISL (Interaction Specification Language) est un langage dédié `a l'assemblage dynamique des composants et permet de décrire des *interactions logicielles*. L'utilisation d'interactions logicielles permet de modifier dynamiquement le comportement des composants par modification de l'arbre d'exécution des méthodes inter-agissantes, mais également d'assembler de mani`ere comportementale des composants par invocation de méthodes. Une interaction correspond au Contrôleur dans le patron MVC et permet d'éviter tout lien statique entre le mod`ele et la vue. Avec ce mécanisme, RainBow propose un mod`ele de composition des composants d'IHM basé sur la fusion des OIA.

L'exemple fourni dans (Fierstone *et al.*, 2003) présente deux composants métiers : COMMERCIAL et CLIENT. COMMERCIAL est associé au composant d'IHM FICHECOMMERCIAL et le composant CLIENT `a FICHECLIENT. Comme un COMMERCIAL démarche plusieurs CLIENTS (ces deux composants métiers sont connectés), un composant d'IHM que nous appellerons FICHECLIENTÈLE peut être obtenu par composition des composants d'IHM FICHECOMMERCIAL et FICHECLIENT.

Les aspects les plus intéressants du point de vue de notre travail sont la séparation des concepts de composant métier et de composant d'IHM ainsi que l'indépendance de ces derniers vis-`a-vis d'une architecture cible ou d'un langage. Nous considérons comme une limitation le fait qu'il faille associer `a chaque composant métier non composite une description compl`ete d'une IHM. De plus, nous pensons qu'automatiser la fusion d'OIA n'est possible que pour des cas simples. Des probl`emes tels que la redondance d'information ou la déduction d'un agencement ergonomique des éléments semblent difficiles. La production d'une IHM compl`ete dans ce cadre paraît donc difficilement automatisable et des interventions humaines seront nécessaires pour régler ces probl`emes. Un autre point important est la cohérence globale d'une IHM finale. En effet, il est difficile de garantir des propriétés ergonomiques générales, comme la présence d'un menu d'aide dans la barre de menus de toute fenêtre. Nous pensons améliorer cela en n'associant pas une IHM compl`ete `a chaque composant. En effet, l'IHM d'une application sera produite globalement par un outil d'interfaç age `a partir de l'étude de la partie métier compl`ete.

2.2 Les composants JavaBeans

Un *Java Bean* est défini comme un « *composant logiciel réutilisable qui peut être manipulé graphiquement dans un environnement de développement* » (Sun, 1997). C'est-`a-dire que tout composant *Java Bean* peut être utilisé graphiquement[4] et s'int`egre dans l'environnement de développement fourni par Sun Microsystems. Les mécanismes utilisés pour cette intégration constituent les apports fondamentaux de ce mod`ele de composants et sont intéressants pour notre problématique. En effet, le mod`ele *Java Bean* montre en quoi le protocole « publish/subscribe » (schéma de conception Observateur (Gamma *et al.*, 1995)), déj`a utilisé pour mettre en œuvre l'architecture MVC, est également un protocole d'assemblage non anticipé de composants.

Un composant *Java Bean* est un objet Java standard qui poss`ede des attributs et des méthodes, des mécanismes d'enregistrement/notification pour les événements qu'il produit, des méthodes de traitement d'événements et des propriétés. Un propriété dénote une caractéristique paramétrable du composant. Par exemple, les méthodes *getValeur()* et *setValeur(int)* d'un COMPTEUR indiquent que ce composant poss`ede une propriété nommée *valeur*. On remarque que les propriétés sont définies implicitement par convention de nommage.

Un composant *Java Bean* peut s'enregistrer comme écouteur d'un ou plusieurs types

[4] Attention, tous les *Java Beans* sont manipulables graphiquement mais ne sont pas n´ecessairement des composants graphiques (Boutons, etc.).

d'événements aupr`es d'un autre *Java Bean*. Chaque *Java Bean* g`ere donc une liste d'écouteurs pour chacun des types d'événements qu'il émet. Lorsqu'un *Java Bean* (source) émet un événement, il invoque pour tous les écouteurs enregistrés leur méthode de traitement de cet événement. Les événements fournissent le mécanisme qui permet de connecter deux *Java Beans* développés indépendamment pourvu que l'un produise et l'autre écoute un même type d'événement.

Couplé avec le mécanisme de *publication/souscription*, les propriétés peuvent être dotées de comportements particuliers. C'est le cas des propriétés de type *lié* (*BoundProperties*) qui émettent un événement `a chaque fois que leur valeur est modifiée. Sur le même principe de fonctionnement, les propriétés de type *véto* (*VetoableProperties*) permettent `a leurs écouteurs de s'opposer (droit de véto) `a leur changement de valeur.

Nous retenons de ce mod`ele de composants le mod`ele d'assemblage qui est novateur et puissant puisqu'il permet de programmer des composants séparément (dans l'espace et le temps) puis de les connecter afin qu'ils accomplissent une tâche et collaborent sans se connaître préalablement. Nous avons donc adopté ce mécanisme d'assemblage des composants qui va aussi nous permettre de mettre en place de faç on non anticipée l'architecture MVC entre nos composants métiers et l'IHM produite semi-automatiquement. Nous avons aussi remarqué la notion de propriété et l'existence de divers types de propriétés qui sont implicites et sous-étudiés dans ce mod`ele.

3 Un modèle de composants interfaçables

Nous présentons dans cette section un mod`ele de composants intégrant des notions pour l'interfaç age semi-automatique. Nous commenç ons par définir le concept de *composant interfaçable*. Ensuite, nous approfondissons la notion implicite de *propriété* du mod`ele de composants *Java Beans* et présentons une modélisation de cette notion. Enfin, nous illustrons comment assembler deux composants interfaç ables en utilisant des caractéristiques des propriétés.

3.1 Notion de composant interfaçable

Un outil d'interfaç age analyse la partie métier d'une application construite par assemblage de composants et produit une IHM. En partant de ce constat, nous posons la définition suivante pour un composant interfaç able.

Composant interfaçable : Composant métier constitué de propriétés intégrant des informations et mécanismes qui facilitent la production semi-automatique d'IHM par un outil d'interfaç age.

La partie métier d'une application peut donc être construite par assemblage de composants interfaç ables. De tels composants sont spécifiquement conç us pour faciliter la production d'une IHM par un outil d'interfaç age car ils int`egrent explicitement la notion de propriété.

3.2 Un modèle de propriétés

Nous pensons que les propriétés des composants doivent êtres définies de faç on beaucoup plus fine et explicite que dans les mod`eles de composants connus car elles sont fondamentales au paramétrage et `a l'assemblage. Nous proposons une définition de ce concept en nous inspirant du mod`ele *Java Bean* o`u la notion de propriété est implicite.

Propriété : Unité sémantique qui affecte l'apparence, définit l'essence ou induit le comportement d'un composant.

Nous proposons pour le mod`ele de composants interfaç ables, le mod`ele de propriétés présenté par le schéma UML (Group, 2003) de la figure 2. Un composant interfaç able est constitué de propriétés qui poss`edent toutes un nom et un type. Ensuite, nous avons dégagé différentes sortes de propriétés suivant des caractéristiques d'acc`es, de valuation et d'observabilité. Ce premier niveau dans notre taxonomie des propriétés peut être enrichi par de nouvelles sortes de propriétés comme le montre la contrainte *incomplete*. La contrainte *overlapping* sur le lien de spécialisation indique qu'il n'y a pas disjonction entre les différentes sous-catégories et qu'une propriété peut tr`es bien être dans plusieurs d'entre elles, par exemple `a la fois accessible, valuée et observable. Voyons maintenant la signification de ces trois catégories de propriétés.

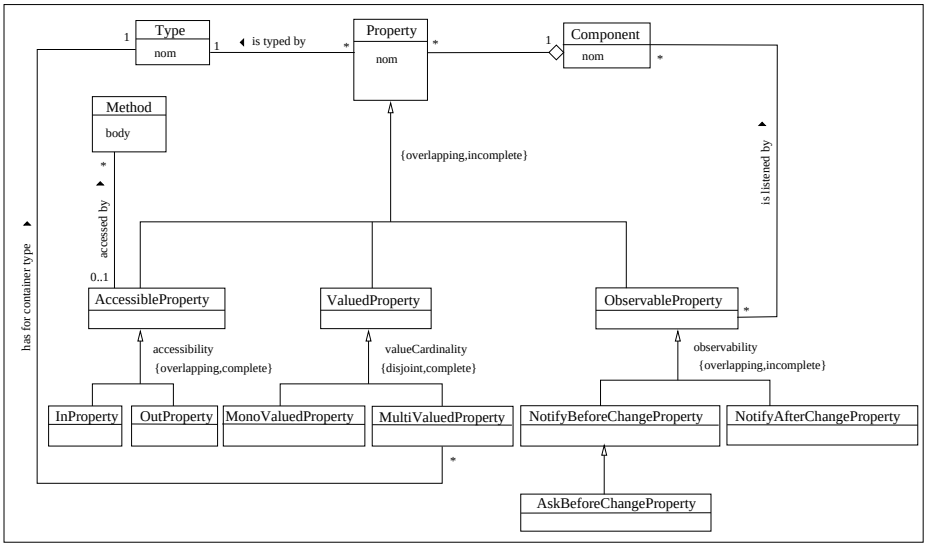

FIG. 2 – Un mod`ele de propriétés pour composants interfaç ables

Les propriétés *OutProperty* et *InProperty* ont respectivement leur valeur accessible en lecture et en écriture par des méthodes spécifiques appelées accesseurs. Ces méthodes d'acc`es sont définies sur le composant interfaç able possédant la propriété. Une propriété peut être `a la fois une *OutProperty* et une *InProperty*.

Une propriété est soit mono-valuée soit multi-valuée. Par exemple, un composant *compteur* peut avoir une propriété *historique* contenant l'ensemble de ses valeurs au cours du temps. Une telle propriété est une collection (une liste ou un tableau) de valeurs enti`eres.

Un composant peut observer une propriété d'un autre composant. On dit qu'il est écouteur de cette propriété. Les caractéristiques d'observabilité d'une propriété définissent son comportement vis-`a-vis de ses écouteurs lorsque sa valeur change. Nous avons recensé au moins trois types de propriété intéressants qui n'ont pas prétention `a l'exhaustivité et peuvent être combinés :

NotifyAfterChange (NAC) une propriété de ce type notifie `a ses écouteurs le fait que sa valeur a été modifiée. Ce mode d'observabilité correspond aux *BoundProperties* des *Java Beans* aussi nommées propriétés liées,

NotifyBeforeChange (NBC) type propriété notifiant `a ses écouteurs que sa valeur va être modifiée,

AskBeforeChange (ABC) type propriété signalant `a ses écouteurs que sa valeur va être modifiée mais ces derniers ont un droit de véto et peuvent s'opposer `a ce changement. Si aucun écouteur n'a utilisé ce droit, la nouvelle valeur est affectée `a la propriété. Cela correspond aux *VetoProperties* des *Java Beans* dites propriétés vétos.

La possibilité de subordonner un traitement `a l'acc`es `a une donnée s'inspire de la notion de démon ou d'attachement procédural dans les langages `a frames (Minsky, 1975). Dans notre approche, tout composant préalablement enregistré comme écouteur d'une propriété peut effectuer des traitements `a chaque notification de changement de la propriété. La section qui suit décrit l'intérêt de ce mécanisme pour l'assemblage et l'interfaç age des composants.

3.3 Un modèle d'assemblage basé sur la notion de propriété observable

Deux composants interfaç ables peuvent être assemblés si l'un s'enregistre comme écouteur d'une propriété observable de l'autre. Ce mécanisme d'enregistrement et notification permet d'assembler de faç on non anticipée des composants écrits indépendamment et sans aucune idée de leur utilisation future. Les caractéristiques d'observabilité d'une propriété offrent donc des supports pour l'assemblage des composants interfaç ables. Chaque type de propriétés permet une forme d'assemblage différente comme nous l'avons présenté en section 3.2. Par exemple, un composant JOURNAL écouteur de la propriété *valeur* de type *NAC* (ou *lié*) d'un composant COMPTEUR peut effectuer des traitements lors de la modification de cette propriété et ainsi sauvegarder un historique des valeurs de cette propriété. Un autre exemple est un composant VANNE doté de la propriété *état* de type *ABC* (ou *véto*) prenant les valeurs *ouvert* ou *fermé*. Un composant RÉGULATEUR inscrit en tant qu'écouteur de cette propriété peut ainsi s'opposer `a l'ouverture ou la fermeture de la vanne.

4 Un langage à composants interfaçables

Cette section présente notre langage (ICL) qui permet d'écrire des programmes `a base de composants interfaç ables. Notre objectif est de produire un langage opérationnel permettant d'utiliser $Things^{TM}$ comme outil d'interfaç age. $Things^{TM}$ ne fonctionne qu'`a partir de code Java compilé standard. Nous avons donc décidé d'implanter ICL par extension de Java. Cette section commence par une présentation d'ICL, décrit ensuite les mécanismes mis en place pour l'interfaç age des composants interfaç ables et détaille enfin l'implémentation actuelle d'ICL en OpenJava.

4.1 Présentation du langage ICL

Nous avons introduit dans le langage ICL des mots clés et des r`egles syntaxiques qui permettent de déclarer des composants interfaç ables et leurs propriétés définies dans le mod`ele présenté en section 3.2. Ces mots-clés sont les suivants :

component <**component-name**> déclare un composant interfaç able,

property annonce la déclaration d'une propriété,

in indique que la propriété est accessible en écriture via une méthode sur le composant,

out indique que la propriété est accessible en lecture via une méthode sur le composant,

nota indique que la propriété notifie ses écouteurs apr`es changement de valeur (NAC),

notb indique que la propriété notifie ses écouteurs avant changement de valeur (NBC),

askb indique que la propriété demande l'autorisation de changer de valeur `a ses écouteurs (ABC),

collection of <**element-type**> annonce que la propriété est multi-valuée et contient des éléments de type <element-type>.

Il est possible de combiner (en respectant les contraintes énoncées dans la figure 2) ces mots-clés lors de la déclaration d'une propriété afin d'obtenir tous les types propriétés définies dans le mod`ele.

La figure 3 montre un exemple de code métier ICL déclarant un composant BOOK. La propriété *title* de type *String* est déclarée `in out` c'est-`a-dire accessible en lecture et écriture par des accesseurs qui sont automatiquement générés. Ceci s'apparente aux mécanismes du langage CLOS o`u des directives telles que *:reader*, *:writer* et *:accessor* permettent lors de la déclaration d'un attribut d'indiquer que des accesseurs seront disponibles. La déclaration de la propriété *isbn* contient `askb` ce qui permet par exemple `a un composant dédié `a la vérification des ISBN (chaîne alphanumérique codifiée) de s'opposer `a des valeurs incorrectes.

```
Component Book {

  /** The title of this book */
  property in out String title;

  /** The author(s) of this book. */
  property in out Collection authors collection of Author;

  /** The ISBN of this book. */
  property in out askb String isbn;

  /** The publisher of this book. */
  property in out Publisher publisher;

  /** The cover delivers file location */
  property in out nota String imageFilename;

  [...]
}
```

FIG. 3 – Extrait du code ICL d'un composant BOOK

4.2 Problème de la visualisation non-anticipée

Les composants interfaç ables sont des composants métiers constitués de propriétés qui sont utiles pour l'assemblage des composants mais aussi pour leur interfaç age. En effet, les propriétés traduisent des caractéristiques configurables des composants et sont par conséquent potentiellement intéressantes `a représenter sur une IHM. Toutefois, ces propriétés sont déclarées par le programmeur de la partie métier d'une application qui n'a pas `a faire de choix d'interfaç age. Lors de la réalisation d'une IHM, il faut donc indiquer `a l'outil d'interfaç age les propriétés *visualisables*, c'est-`a-dire celles `a représenter sur l'IHM. Nous sommes ici confrontés `a un probl`eme de visualisation non-anticipée de certaines propriétés métiers. En effet, il est difficile de garantir des mécanimes utilisables par un outil d'interfaç age pour les propriétés visualisables puisqu'elles sont déclarées a posteriori. Face `a cette problématique, nous avons identifié au moins deux mécanismes `a intégrer aux composants interfaç ables.

Le premier doit permettre de synchroniser les valeurs des propriétés visualisables avec celles effectivement représentées sur l'IHM. Une solution consiste `a stocker pour chaque propriété visualisable sa liste d'écouteurs et fournir des méthodes permettant l'abonnement et la résiliation des écouteurs. Il faut ensuite pouvoir notifier aux écouteurs les modifications subies par la propriété. Il existe plusieurs faç ons d'implémenter cela suivant les possibilités offertes par le langage utilisé. Dans un langage `a frames, on utiliserait une table globale contenant une liste d'écouteurs pour chaque propriété et des attachements procéduraux (Minsky, 1975) sur les accesseurs de la propriété afin de notifier les écouteurs inscrits dans la table. Dans un langage `a objet réflexif comme Smalltalk, on pourrait aussi stocker les écouteurs dans une table globale, par contre on modifierait directement le code des accesseurs de propriétés qui doivent être liées. Dans un langage orienté aspects, on pourrait considérer les caractéristiques des propriétés comme des aspects (Kiczales *et al.*, 1997) non fonctionnels spécifiés de faç on déclarative lors de la déclaration d'une propriété via nos mots-clés. Dans ce cas, il fau-

drait, dans l'hypoth`ese o`u c'est possible, ajouter dynamiquement un aspect relatif `a une propriété. Enfin, dans un langage tel que Java (statiquement typé et `a réflexivité limitée en standard), notre idée est de stocker dans chaque composant une *hashtable* qui associe `a un nom de propriété[5] soit la valeur *null* indiquant que la propriété n'est pas visualisable, soit la liste de ses écouteurs. Tout composant interfaç able doit donc posséder les méthodes *addListener*, *RemoveListener* et *setVisible* dont le code est présenté sur la figure 4. Ensuite, lors de la génération des accesseurs d'une propriété, il faut inclure le code rendant possible la visualisation non-anticipée qui teste la valeur associée au nom de la propriété dans la *hashtable* et notifie la modification `a ses écouteurs si besoin. On remarque que cette solution fonctionne mais impose un test dans tous les accesseurs de propriétés.

```
private Hashtable hash;

/** Ajoute aListener à la liste des écouteurs de la propriété aPropertyName */
public void addListener( String aPropertyName, Component aListener ) {
  Collection listeners = (Collection) hash.get( aPropertyName );
  if( listeners != null ) {
    listeners.add( aPropertyName );
    hash.put( aPropertyName, listeners );
  }
}

/** Enlève aListener de la liste des écouteurs de la propriété aPropertyName */
public void removeListener( String aPropertyName, Component aListener ) {
  Collection listeners = (Collection) hash.get( aPropertyName );
  if( listeners != null ) {
    listeners.remove( aPropertyName );
    hash.put( aPropertyName, );
  }
}

/** Rend la propriété aPropertyName visualisable */
public void setVisible( String aPropertyName ) {
  Collection listeners = (Collection) hash.get( aPropertyName );
  if( listeners == null )
    hash.put( aPropertyName, new Vector() );
}
```

FIG. 4 – Extrait du code Java nécessaire pour les *propriétés visualisables*

Un outil d'interfaç age peut alors produire une IHM qui s'enregistre en tant qu'écouteur des propriétés visualisables qu'elle représente. Les avantages de cette technique sont la bonne séparation du code métier et du code de l'IHM et la synchronisation `a tout instant des données métiers avec celles affichées. Prenons l'exemple illustré sur la figure 5 o`u un composant *compteur* a une propriété *valeur*. Lors de l'interfaç age, on indique que la propriété *valeur* est visualisable en invoquant la méthode *setVisible* du composant COMPTEUR. L'IHM produite peut donc s'inscrire en tant qu'écouteur de la propriété *valeur* du compteur par la méthode *addListener* de ce dernier. La propriété notifiera alors tous ses changements de valeurs `a l'IHM qui pourra actualiser les diverses représentations de cette propriété.

[5]Il ne peut pas y avoir deux propri´et´es de mˆeme nom dans un composant.

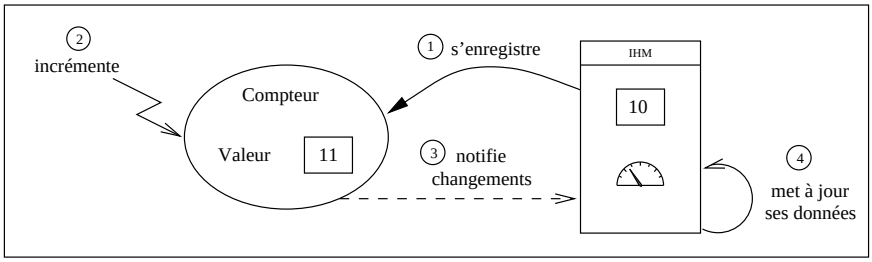

FIG. 5 – Exemple d'utilisation du mécanisme intégré aux composants interfaç ables pour le propriétés visualisables

Le deuxi`eme mécanisme que nous avons identifié doit permettre d'associer `a un composant interfaç able une représentation graphique particuli`ere qui pourra être utilisée pour le représenter sur une IHM. Par exemple, on peut vouloir doter le composant BOOK d'une représentation particuli`ere. Les composants sont ainsi dotés d'une propriété *specificRepresentation* dont la valeur est un « widget » directement intégrable `a une IHM pour représenter le composant. En Java, nous fixons donc le type de cette propriété `a *JPanel*. Associer une représentation spécifique `a un composant interfaç able nécessite alors d'utiliser l'accesseur en écriture *setSpecificRepresentation(JPanel)* de cette propriété et l'IHM peut ensuite l'obtenir via l'accesseur en lecture. Un composant composite peut combiner visuellement les représentations spécifiques de ses constituants afin de s'en fabriquer une.

4.3 Mise en œuvre d'un prototype du langage ICL

Le langage Java étant dépourvu de méta-niveau explicite, nous avons utilisé le Protocole Méta-Objet (Pavillet, 2000) OpenJava (Tatsubori *et al.*, 1999) car les méta-programmes sont exécutés durant la compilation ce qui permet d'obtenir en sortie du code Java standard utilisable par $Things^{TM}$ dans sa version actuelle.

Un composant interfaç able est implémenté par une méta-classe OpenJava qui en définit la structure et les traitements nécessaires pour obtenir une classe Java standard. Les éléments présentés dans la section 4.2 ont été mis en œuvre pour implémenter les composants et les propriétés. 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, 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'interfaç age introduites dans nos composants dans une forme utilisable par

$Things^{TM}$ dans sa version actuelle. C'est ainsi que sont générés un fichier de contexte, des éléments d'interface composites, etc.

5 Exemple d'une application de gestion de bibliothèque

Nous avons programmé en ICL une application simplifiée de gestion d'une biblioth`eque constituée des composants : LIBRARY, BOOK, AUTHOR, PUBLISHER et USER. La figure 6 montre le schéma UML simplifié de cette application exemple. Quelques extraits du code ICL des composants BOOK et LIBRARY apparaissent respectivement dans les figures 3 et 7.

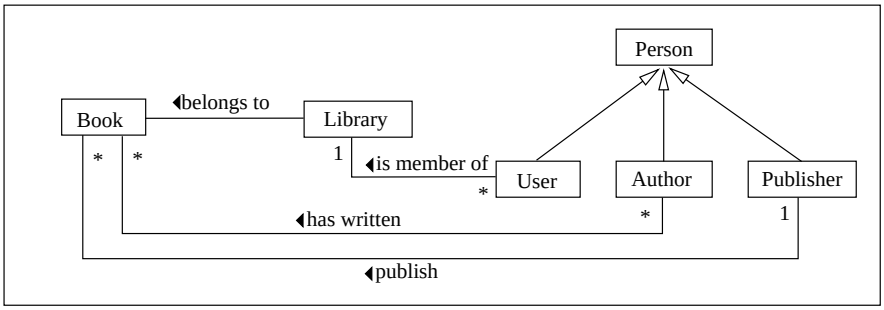

FIG. 6 – Diagramme UML simplifié d'une application de gestion de biblioth`eque

```
Component Library {

  /** The books contained in the library. */
  property inout Vector books collection of Book;

  /** The Members of the library the library. */
  property in out Vector members collection of User;

  [...]
}
```

FIG. 7 – Extrait du code ICL du composant LIBRARY

Notre chaîne de compilation permet de traduire ce code en Java et gén`ere un fichier de contexte pour $Things^{TM}$. Ensuite, il est possible d'utiliser les possibilités offertes par les composants interfaç ables comme indiquer qu'un livre doit être représenté par l'affichage de l'image de sa couverture qui est disponible via sa propriété *fileName*. La figure 8 montre l'IHM produite par $Things^{TM}$.

Les plus-values apportées par notre langage sont d'une part un code métier simplifié par l'expression directe de propriétés et d'autre part une IHM de meilleure qualité. Dans la fenêtre intitulée « Books - All », le champ titre du livre sélectionné est représenté

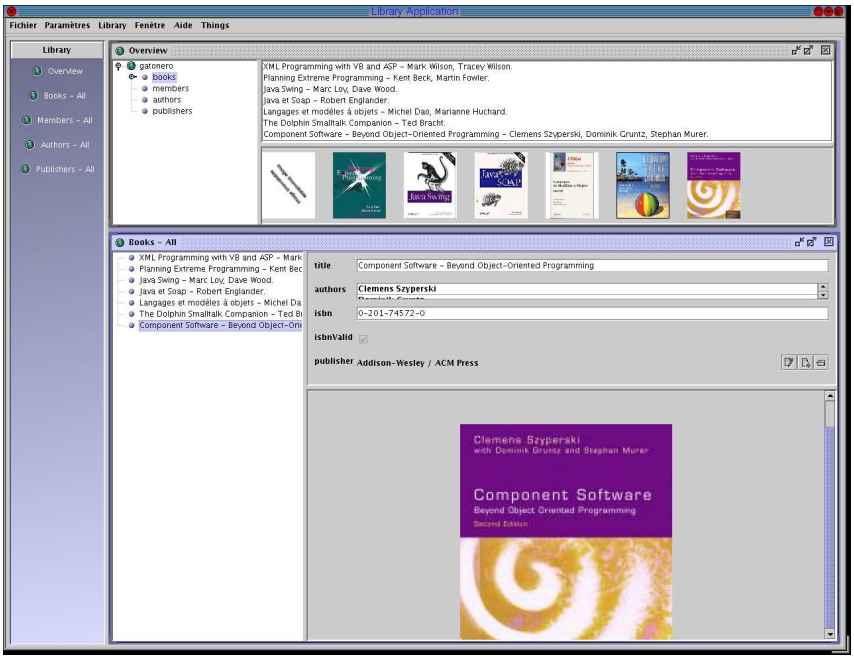

FIG. 8 – Capture d'écran du résultat de l'interfaç age automatique par $Things^{TM}$ d'une application de gestion de biblioth`eque écrite en ICL

par un champ texte éditable car cette propriété est qualifiée d'`in out` dans le code. Si un programme tiers modifie le titre d'un livre, le changement sera immédiat sur l'IHM grâce au mécanisme de notification imposé sur les propriétés représentées sur l'IHM. Les livres sont représentés par l'affichage de leur couverture. Comme une biblioth`eque a une propriété multi-valuée de livres (cf. figure 7), elle est représentée de faç on spécifique comme la juxtaposition des couvertures des livres.

6 Conclusion et Perspectives

Dans le contexte du développement par composants, nous avons étudié ce que pourraient être des composants interfaç ables. De tels composants permettent d'améliorer l'automatisation de l'interfaç age des applications. Ils assurent aussi une meilleure qualité de l'IHM produite puisque nous avons étudié et proposé une solution `a des limitations des outils d'interfaç age actuels. A partir de ces études, nous avons donné une spécification et un mod`ele de la notion de composant interfaç able et proposé le langage ICL pour écrire de tels composants. Ce langage int`egre une syntaxe et des mécanismes spécifiques permettant la connexion des applications avec un outil d'interfaç age du type de $Things^{TM}$. Un prototype du langage ICL a été programmé en OpenJava.

Une *chaîne de compilation* a été mise en œuvre pour traduire le code ICL des composants interfaç ables en code Java. Finalement, nous avons programmé en ICL la partie métier d'une application et réalisé son interfaç age par $Things^{TM}$. L'interface produite montre la plus value apportée par la programmation en ICL pour l'automatisation de l'interfaç age.

Une perspective intéressante de ce travail serait de considérer l'interfaç age comme un service orthogonal. Cette idée va dans le sens des travaux sur la séparation de aspects (Hürsch & Lopes, 1995) (separation of concerns) qui visent `a séparer les aspects métiers d'autres aspects dits orthogonaux. Cette séparation est mise en pratique dans certaines approches comme J2EE o`u les composants doivent s'exécuter dans des conteneurs qui fournissent aux composants des savoirs-faire orthogonaux comme la persistance, le versionning, etc. Cela permet aux programmeurs des composants de bénéficier de ces services sans les écrire puisqu'ils sont rendus par la plate-forme d'exécution. Il serait intéressant d'étudier si l'interfaç age ne pourrait être un service orthogonal offert par le conteneur afin de présenter les composants dont il supporte l'exécution dans une IHM. Pour cela, le conteneur intégrerait un outil d'interfaç age du type de $Things^{TM}$ adapté `a l'architecture matérielle sous-jacente qui réaliserait l'interfaç age par assemblage dynamique avec les composants souhaitant être interfacés.

Références

ARISHOLM E. (1998). Incorporating rapid user interface prototyping in object-oriented analysis and design with genova. In *Proceedings of NWPER'98 Nordic Workshop on Programming Environment Research, Bergen*, p. 155–161.

BLEVINS D. (2001). Overview of the Enterprise JavaBeans component model. p. 589–606.

COUPAYE E. B. T. & STEFANI J. (2002). Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)* : Springer-Verlag.

FIERSTONE J., PINNA A.-M. & RIVEILL M. (2003). *Architecture logicielle pour l'adaptation et la composition d'IHM - mise en oeuvre avec le langage SUNML*. Rapport interne, Rapport de recherche, Laboratoire I3S — Université de Nice-Sophia Antipolis —. *http://rainbow.essi.fr*.

GAMMA E., HELM R., JOHNSON R. & VISSIDES J. (1995). *Design Pattern : Element of Reusable Object Oriented Software*. Addison-Wesley.

GROUP O. M. (2003). *OMG Unifi ed Modeling Language Specifi cation, Version 1.5*.

HÜRSCH W. & LOPES C. V. (1995). *Separation of concerns*. Rapport interne, Rapport de recherche, Northeastern University.

KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M. & IRWIN J. (1997). Aspect-oriented programming. In M. AKSIT & S. MATSUOKA, Eds., *ECOOP'97 — The 11th European Conference on Object-Oriented Programming*, volume 1241 of *lncs*, p. 220–242, Jyväskylä, Finland : sv.

MEDVIDOVIC N. & TAYLOR R. N. (1997). A framework for classifying and comparing architecture description languages. In M. JAZAYERI & H. SCHAUER, Eds., *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, p. 60–76 : Springer–Verlag.

MINSKY M. (1975). A Framework for Representing Knowledge. In P. WINSTON, Ed., *The Psychology of Computer Vision*, p. 211–281. ny : mgh.

OMG (2002). *Manual of Corba Component Model V3.0. http ://www.omg.org/technology/-documents/formal/components.htm.*

PAVILLET G. (2000). *Des langages de programmation à objets à la repr´esentation des connaissances à travers le MOP : vers une int´egration.* PhD thesis, Université de Montpellier II, Ecole Doctorale I2S.

SUN (1997). *JavaBeans^{TM}. http://java.sun.com/products/javabeans/docs/spec.html.*

SZYPERSKI C. (2002). *Component Software : Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley.

TATSUBORI M., CHIBA S., ITANO K. & KILLIJIAN M.-O. (1999). OpenJava : A class-based macro system for java. In *OORaSE*, p. 117–133.

WOUTERS S. (2002). Analyse comparative de générateurs d'interfaces d'applications de gestion à partir d'un diagramme de classes UML. Master's thesis, Université catholique de Louvin, Département d'Administration et de Gestion.