Un modèle abstrait de composants adaptables

Carine Courbis[*] — Pascal Degenne[**] — Alexandre Fau[**] — Didier Parigot[*]

[*] *INRIA Sophia Antipolis Projet OASIS et* [**] *W3C*
2004, route des Lucioles BP 93
F-06902 Sophia-Antipolis cedex, France
Prénom.Nom@sophia.inria.fr

RÉSUMÉ. *Depuis quelques années, l'approche par composants dans la conception de logiciels complexes est apparue pour aider à résoudre les problèmes de répartition, de déploiement et de réutilisation de code. Cette approche paraît la mieux adaptée pour assurer la pérennité, l'évolution, l'interopérabilité et la maintenance d'applications. Cet article présente le modèle de composants adopté pour la réalisation d'un atelier de développement. L'originalité de ce modèle est d'être extensible et exportable vers d'autres modèles plus connus tels que les Web Services, CCM (CORBA Component Model) ou EJB (Entreprise Java Bean) tout en respectant nos besoins. La validité de ce modèle est aussi éprouvée puisqu'il a été utilisé pour l'implantation de notre outil. Cette approche pragmatique nous a permis de concevoir un modèle de composants simple et réaliste.*

ABSTRACT. *Since a few years, the component approach for the design of complex software has appeared to help to solve problems of code repartition, deployment and reusability. This approach turns out to match the best result which ensures application evolution, interoperability and maintenance. This article introduces the component model used to implement an Integrated Development Environment. The salient feature of this model is that it is extensible and transposable into well-known models such as Web-Services, CORBA Component Model or EJB. The validity of this model has been assessed as it was used to implement our tool. This pragmatic approach was helpful for us to design an simple and realistic component model.*

MOTS-CLÉS : *modèle de composants, technologies XML*

KEYWORDS: *component model, XML technologies*

Technique et science informatiques. Volume x - n° x/2004. Systèmes à composants adaptables et extensibles , pages 10 à 30

1. Introduction

Avec l'émergence d'internet, la conception et le développement d'applications complexes doivent impérativement prendre en compte les aspects de répartition (application distribuée), déploiement et réutilisation de code. Cette évolution a pour conséquence un changement radical dans la programmation de telles applications. Avec ces bouleversements, l'approche objet, adoptée pour la production de logiciels, présente des limitations. Les mécanismes offerts sont de trop bas niveau pour exprimer des interconnexions complexes entre objets ou inexistants pour gérer des objets distribués sur plusieurs machines. Ces limitations ont conduit à l'émergence d'un nouveau paradigme de programmation, le composant [SZY 98, MAR 02, MAR 01, BRU 02, Jon, BRU 01], afin de mieux séparer les aspects de communication et les parties fonctionnelles, de s'abstraire des langages de programmation, et de construire des applications par assemblage de composants. Ce passage de la programmation à la petite échelle (*in the small*) à la programmation à grande échelle (*in the large*) promet une meilleure réutilisation du code.

Cet article présente notre démarche de conception d'une architecture par composants pour un atelier logiciel nommé SMARTTOOLS [1] [PAR 02, COU 03]. Tout d'abord, les caractéristiques spécifiques à un tel logiciel sont explicitées ainsi que les contraintes ayant influencé notre démarche de conception de l'architecture. L'énumération de ces contraintes nous semble importante pour justifier nos choix de conception du modèle de composants sous-jacent. De plus, nous pensons que ces contraintes, dans quelques années, seront communes à de nombreux développements logiciels et pas seulement à notre contexte, afin d'assurer une meilleure évolution et adaptabilité aux applications produites. L'apport de cet article est donc de décrire une démarche de conception d'architecture logicielle transposable pour de nombreuses autres applications.

Avoir une architecture par composants, pour notre outil, était nécessaire et ce pour plusieurs contraintes :

– SMARTTOOLS est un générateur d'outils pour des langages métiers ou de programmation. Il était donc vital de pouvoir *séparer*, de manière modulaire, les outils de base génériques (le cœur du système) et ceux générés spécifiquement à un langage donné afin d'éviter d'augmenter, à l'infini, la taille de notre atelier logiciel.

– Notre méta-outil doit pouvoir être configurée (restreinte) en fonction de l'application souhaitée par l'utilisateur, afin d'éviter de charger en mémoire des outils inutiles; une application étant composée d'un ensemble d'outils. Il s'avère aussi important d'offrir la possibilité de charger en cours d'exécution, à la demande, de nouveaux outils associés à des langages. Son interface graphique et son architecture doivent s'adapter aux diverses configurations possibles, différentes pour chaque langage traité. Elles doivent être configurables et extensibles. Il est donc important que les applications générées soient seulement composées des *outils nécessaires* et aient des *ar-*

1. http://www-sop.inria.fr/oasis/SmartTools

chitectures dynamiques. Un mécanisme de gestionnaire de composants est utile pour interpréter cette configuration de composants au lancement d'une application et pour intégrer de nouveaux composants pendant l'exécution; c'est une sorte de «machine virtuelle de composants».

– Notre atelier est basé sur une approche de programmation générative [COU 03] et, de plus, utilise fortement des techniques standards (XML - *Extensible Markup Language*) afin de réduire les coûts de développement. Pour bénéficier des outils développés autour de ces standards, il doit être possible de les intégrer (*importer*) facilement en ne modifiant, ni le cœur du système, ni les outils générés. Ainsi notre plate-forme est ouverte et évolutive. En effet, quand ces standards et donc les outils associés évoluent, elle évolue aussi de concomitance.

– Les applications (logiciels) produites ont vocation à être utilisés en dehors de notre atelier. Il faut qu'ils puissent être facilement *exportés* vers d'autres applications. Il est donc important qu'ils soient indépendants de toute technologie de composants pour simplifier leur transposition vers les technologies de composants actuelles ou futures. Cette caractéristique donne une certaine latitude d'évolution à nos outils pour les prochaines années.

Toutes ces raisons nous ont donc conduits à opter pour une architecture par composants à modèle dédié à nos besoins, indépendant de toute technologie, et facilement transposable vers les technologies connues. En effet, la dernière raison énumérée ci-dessus nous interdisait de choisir une technologie particulière. Cette démarche correspond à la nouvelle stratégie défendue par l'OMG (*Object Management Group*), MDA (*Model-Driven Architecture*) [GRO 00], qui est basée sur une notion de transformation de modèles. Un des résultats importants de notre démarche est de montrer, sur un exemple particulier, l'intérêt de définir un modèle indépendant de l'implantation (PIM - *Platform Independant Model*) et ensuite de proposer différentes transformations vers des modèles spécifiques (PSM - *Platform Specific Model*).

L'avantage principal de concevoir son propre modèle de composants est d'être en parfaite adéquation avec les besoins. Nos langages de description de composants sont ainsi adaptés et offrent un moyen abstrait (indépendant d'une technologie) pour décrire précisément les composants et comment les assembler pour former une application, dans notre contexte. L'implantation effective du modèle (par génération de conteneurs à partir de ces descriptifs) prend en compte toutes les particularités de notre plate-forme, sans pour autant nuire à la lisibilité de ces descriptifs. Cette génération de conteneurs automatise la programmation des parties non-fonctionnelles des composants. Avoir un générateur facilite la prise en compte de nouveaux besoins ou de nouvelles technologies par simples modifications de ce dernier qui sont automatiquement reportées sur l'ensemble des applications produites. Cela facilite l'évolution de la plate-forme dans son ensemble.

Cette séparation entre le modèle et sa mise en œuvre rend possible l'utilisation de nos composants dans d'autres technologies. En effet, à partir de ces descriptifs, un outil de transformation génère les descriptions (ou interfaces) équivalentes pour les *Web Services*, les composants CORBA (*Common Object Request Broker Architecture*)

ou encore les EJB (*Entreprise Java Bean*). Cette exportation nous a permis d'identifier les particularités, avantages et inconvénients de chaque modèle. Cela a confirmé que l'utilisation d'une de ces technologies aurait été préjudiciable.

En effet, certaines caractéristiques dynamiques de nos composants auraient été difficilement exprimables sans une spécialisation des composants en fonction des langages traités. Pour assurer une forte généricité à nos composants, il était important de pouvoir les adapter aux spécificités de chaque langage traité. Cela nous a imposé de prévoir l'ajout de nouveaux ports à un composant lors de son cycle de vie. Cette caractéristique était pertinente pour rendre nos composants de visualisation génériques.

L'article se décompose en quatre sections. La première section positionne nos travaux par rapport à des travaux similaires en indiquant quel est l'apport de notre démarche. Puis, les deuxième et troisième sections exposent, respectivement, le modèle de composants et sa mise en œuvre choisis en fonction des besoins de notre atelier logiciel. Enfin, la dernière section indique, en retour d'expérience, les caractéristiques que devraient posséder un modèle de composants et sa mise en œuvre afin d'obtenir une application facilement adaptable et évolutive.

2. Positionnement des travaux

Dans cette section, nous allons expliciter les contributions de notre approche par rapport aux nombreux travaux de recherche sur les composants et sur l'approche MDA. Depuis quelques années, l'objectif principal de ces travaux de recherche est d'assurer une meilleure ouverture et adaptabilité des applications, en proposant, par exemple, des modèles de composants plus ouverts [VAD 01] et adaptables dynamiquement. L'intérêt principal est d'obtenir une meilleure évolution des applications vis-à-vis des récents bouleversements dans la conception du logiciel (applications distribuées).

Dans notre cadre, deux contraintes particulières émergent pour assurer un fort potentiel d'évolution à l'outil :

– avoir un modèle de composants indépendant de toute technologie pour pouvoir utiliser (exporter) les applications générées dans n'importe quel environnement quelle que soit sa technologie;

– avoir des composants adaptables pour assurer la généricité de l'outil.

En étudiant les nombreux travaux de recherche sur le domaine des composants [MAR 01, BRU 02, Jon, BRU 01, PAW 01], nous nous sommes rendus compte que notre démarche semblait être particulière, car peu de ces travaux s'intéressent aux mêmes contraintes. Ces travaux peuvent être résumés dans les trois grandes orientations suivantes, complémentaires à nos préoccupations :

– l'ajout de nouveaux services (sécurité, transaction, persistence, etc.) sur un port donné;

– la modification du cycle de vie d'un composant (session, entité, etc.). Par exemple, le remplacement d'un composant en cours d'exécution;

– la modification des interactions entre composants (synchrone, événements, flux, etc.).

Pour positionner nos travaux, il nous semble donc important d'insister sur cette originalité, plus que sur les aspects communs à tout modèle de composants. De plus, sur ces aspects, la mise en œuvre de notre modèle reste encore très embryonnaire, car cela ne constitue pas, pour nous, une priorité. Notre démarche n'a pas la prétention de proposer un modèle générique (en remplacement des autres modèles) mais plutôt de montrer les avantages de définir un modèle de composants dirigé par les besoins.

En ce qui concerne la première contrainte, les ateliers logiciels ou les générateurs d'environnement de programmation [BAT 98, KAS 98, KLI 93, LIE 97, HED 01] s'en préoccupent peu. L'exportation des applications produites n'est pas facilitée. Ce point qui induit une forte dépendance des applications produites avec l'outil lui-même est souvent l'une des importantes critiques de ces outils. Plus généralement, ce soucis d'être indépendant vis-à-vis d'une technologie de composants a motivé l'approche MDA [GRO 00, BéZ 01] et la création d'un modèle UML (*Unified Modeling Language*) pour les composants. Mais tout cela est encore en gestation [MIG 02] et demandera certainement des approfondissements [BEL 01]. Il nous semble que notre approche pourra jouer un rôle d'intermédaire entre ces différentes approches

En ce qui concerne la deuxième contrainte, les travaux sur les composants adaptables, que nous avons étudiés, s'intéressent peu, en général, à étendre l'interface des composants (les conteneurs) par de nouveaux ports d'entrée ou de sortie comme nous le faisons. Dans [OCC 02], les auteurs ont aussi identifié ce besoin mais dans un cadre différent. Cette possibilité est souvent impossible [MAR 01], à cause de l'utilisation d'un typage fort pour la connexion ou d'une approche statique. Les travaux de recherche sur les composants s'intéressent plus à rendre adaptables les parties non-fonctionnelles des composants qu'à introduire de nouvelles fonctionnalités.

Notre modèle et sa mise en œuvre paraissent être complets puisqu'ils semblent être en adéquation avec les rôles usuellement admis dans le processus de production d'applications à base de composants [MAR 02]. Par contre, l'originalité de notre approche apparaît clairement lorsque l'on s'intéresse aux aspects dynamiques des ports proposés et à l'indépendance vis-à-vis d'une technologie. Finalement, notre modèle caractérise les services spécifiques à notre fabrique d'applications orientées langages. Par comparaison, les modèles de composants, en particulier les industriels (EJB et CORBA), correspondent à des fabriques d'applications distribuées.

3. SMARTTOOLS et son modèle de composant abstrait

Cette section présente brièvement notre atelier logiciel et introduit son modèle de composants. Lors de la conception de ce modèle, nous nous sommes imposés deux objectifs : avoir une nette séparation entres les parties fonctionnelles et non-

fonctionnelles, et avoir un modèle qui, bien que spécifique à nos besoins, soit «projetable» vers d'autres technologies existantes.

3.1. *Brève présentation de* SMARTTOOLS

Notre atelier logiciel est un générateur d'outils pour les langages de programmation ou métiers qui s'appuie fortement sur les technologies objets et XML [COU 03, PAR 02]. A partir de la description de structure de données d'un langage, il produit automatiquement (voir figure 1) des outils de manipulation des programmes (ou documents) pour ce langage. En interne, ces programmes sont représentés sous forme d'arbre de syntaxe abstraite. Avec les outils génériques offerts par l'atelier (tels que l'interface graphique, des vues graphiques génériques ou un analyseur syntaxique XML) et les outils générés pour un ou plusieurs langages, l'utilisateur peut rapidement obtenir un squelette d'application (environnement). Il peut ensuite l'enrichir, à l'aide de SMARTTOOLS, avec d'autres éléments spécifiques à son application comme :

– des vues graphiques sur les documents (*pretty-printers*) ;

– des analyseurs lexicaux et syntaxiques (*parsers*) ;

– des traitements particuliers (transformation, vérification et interprétation) liés à ce langage, écrits en Java avec le patron de conception visiteur (*visitor design pattern*) et la programmation par aspects, ou tout simplement des transformations écrites en XSL (*Extensible Stylesheet Language*) du W3C (*World Wide Web Consortium*).

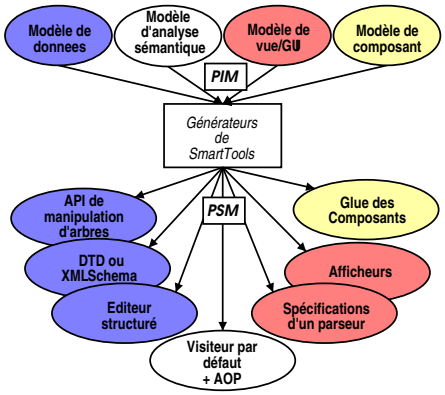

Figure 1. *Vue fonctionnelle de* SMARTTOOLS

Notre atelier accepte, en entrée, les formalismes de description de structure de données du W3C (DTD - *Document Type Definition* - et XML schema), pour être ouvert et surtout pour avoir un vaste champ d'applications dans des domaines variés. Avec cette caractéristique, il n'est pas seulement limité à la conception d'outils pour les langages de programmation, mais pour tout type d'application où les structures de données sont définies à l'aide de ces formalismes.

Fonctionnement de base

Comme son fonctionnement a influencé le modèle de composants, il est très rapidement décrit pour mieux comprendre nos choix. L'atelier peut être utilisé en ligne de commande ou de manière interactive grâce à son interface graphique paramétrable. Dans les deux cas, la première opération effectuée sur le document à traiter est une analyse syntaxique pour obtenir l'arbre de syntaxe abstraite associé qui est la pierre angulaire de tous les outils produits. Puis, sur cette structure, peuvent être appliqués des outils d'analyses sémantiques (compilateurs), de transformation ou de visualisation.

Avoir une interface graphique implique un logiciel plus complexe puisqu'il faut gérer les interconnexions entre les documents, les outils, et les vues. De plus, dans notre cas, l'interface de l'atelier est hautement configurable afin de pouvoir l'adapter aux diverses applications produites. Elle est composée de fenêtres, d'onglets, de vues graphiques d'un ou plusieurs documents, et de menus contextuels (voir figure 2).

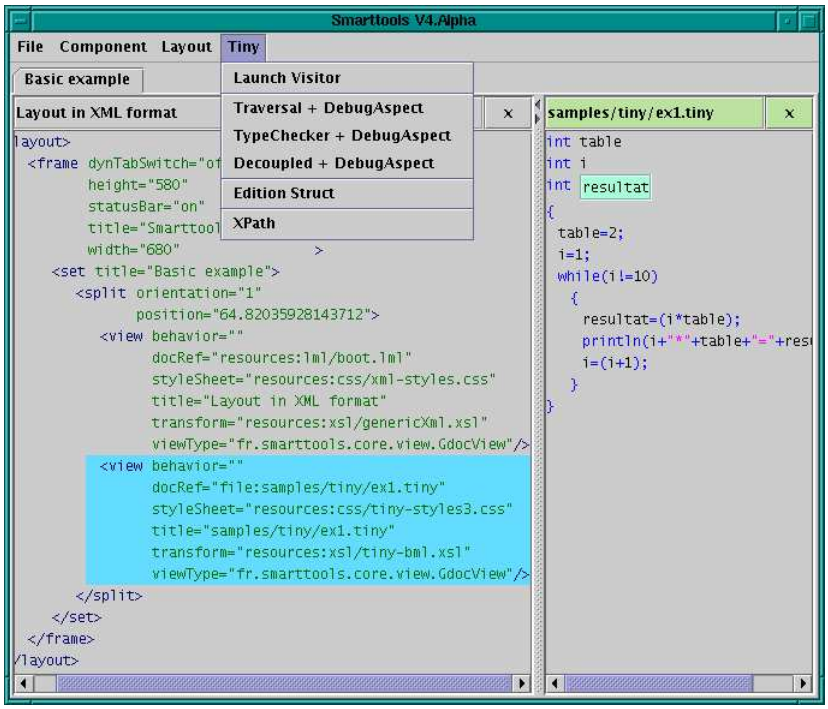

Figure 2. *Exemple d'interface graphique de* SMARTTOOLS

Toute vue graphique dépend d'un document et est obtenue en appliquant une transformation à celui-ci. Le document et ses vues associées doivent rester consistants : toute modification incrémentale du document (changement de la position courante, ajout d'un nouveau nœud, etc) doit être répercutée sur ses vues selon le concept

modèle-vue-contrôleur. Le fonctionnement des vues est générique quel que soit le langage d'appartenance des documents correspondants. Seules les données d'initialisation - le document traité, la transformation à appliquer, et la feuille de styles - diffèrent ainsi que les actions à ajouter au menu, spécifiques à chaque langage.

3.2. *Modèle de composants*

Les deux principaux types de composants manipulés dans notre plate-forme sont celui des *vues* et celui des *documents*. Lors de la conception du modèle de composants abstrait de notre outil, nous devions prendre en compte les spécificités de chaque type. Pour cela, nous avons défini un langage de description de composant dont la syntaxe abstraite est décrite dans la figure 3.

```
component(formalism?, containerclass?, facadeclass?, parser*, lml?, behavior*,
          dependance*, attribute*, (input|output|inout)*)
   attributs : name, type?, extends?

formalism()
   attributs : name, file, dtd
containerclass()
   attributs : name
facadeclass()
   attributs : name, userclassname?
parser(extension)
   attributs : type, classname, extension, generator?, file?
lml()
   attributs : name, file
behavior()
   attributs : file
dependance()
   attributs : name, jar
attribute()
   attributs : name, javatype
input(parameters*)
   attributs : name, method
parameter()
   attributs : name, javatype
arg()
   attributs : type, ref
output(parameter*)
   attributs : name, method
inout(parameter*)
   attributs : name, method, outputname, outputmethod
```

Figure 3. *Syntaxe abstraite de notre modèle de composants*

Des informations communes à ces deux types (*vues* et *documents*) peuvent être identifiées. Pour permettre la factorisation des descriptions, une notion simple d'héritage (attribut *extends*) a été introduite dans notre langage. Par exemple, le type de composant `abstractContainer` de la figure 4 (voir figure 5 pour sa représentation graphique équivalente) est importé par tous nos types de composant.

Notre modèle de composant est composé des éléments de base suivants :

– le nom du composant (attribut `name` de l'élément `component`);

– le nom du type étendu (héritage - attribut `extends` de l'élément `component`);

– le nom du conteneur (`containerclass`) et celui de la façade (`facadeclass`);

– les dépendances vis-à-vis d'autres modules de code (`dependance`);

– les attributs de configuration de l'état du composant (`attribute`);

– les services (ports) en entrée ou en sortie avec leur mode de communication. Pour chaque service, on indique son nom logique, la méthode à appeler dans la partie métier (dans la façade du composant), et les paramètres de la méthode. Les modes de communication sont soit asynchrone (`input` ou `output`), soit synchrone (`inout`).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="abstractContainer">
  <input method="quit" name="quit"/>
  <output name="connectTo" method="connectTo">
    <parameter name="ref_src" javatype="java.lang.String"/>
    <parameter name="type_dest" javatype="java.lang.String"/>
    <parameter name="id_dest" javatype="java.lang.String"/>
    <parameter name="actions" javatype="java.util.HashMap"/>
  </output>
  <output name="exit" method="exit"/>
  <output name="initData" method="initData">
    <parameter name="inits" javatype="java.util.HashMap"/>
  </output>
  <inout name="requestData" method="request"
         output="initData" outputParameter="inits"/>
</component>
```

Figure 4. *Description du composant abstrait `abstractContainer`*

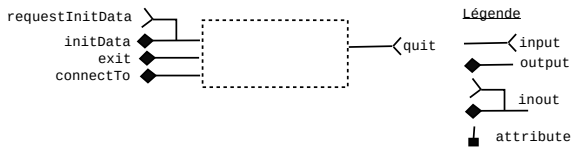

Figure 5. *Schéma du composant abstrait `abstractContainer`*

La figure 6 donne un exemple de description de composant (celle du composant visualisation des graphes) et la figure 7 sa représentation graphique.

Composant vue

Pour éviter de définir un composant vue pour chaque langage et pouvoir proposer des vues génériques, un type de composant vue a été défini, indépendant de tout langage. Ce type décrit le comportement minimal et requis pour tous les composants vues. Sa représentation graphique[2] est donnée en figure 8. Pour ajouter les services spécifiques à un langage, un mécanisme d'extension dynamique des composants vues a été introduit par l'intermédiaire de l'attribut de configuration *behavior* et du service `addBehavior`; ce mécanisme sera décrit dans la section de mise en œuvre.

2. Par la suite, seule la représentation graphique des types des composants est utilisée, plus lisible que la forme XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="graph" type="graph" extends="abstractContainer">
  <containerclass name="GraphContainer"/>
  <facadeclass name="GraphFacade"/>
  <dependance name="koala-graphics" jar="koala-graphics.jar"/>
  <attribute name="nodeType" javatype="java.lang.String"/>
  <input name="addComponent" method="addNode">
    <parameter name="nodeName" javatype="java.lang.String"/>
    <parameter name="nodeColor" javatype="java.lang.String"/>
  </input>
  <input name="addEdge" method="addEdge">
    <parameter name="srcNodeName" javatype="java.lang.String"/>
    <parameter name="destNodeName" javatype="java.lang.String"/>
  </input>
</component>
```

Figure 6. *Description du composant Graphe*

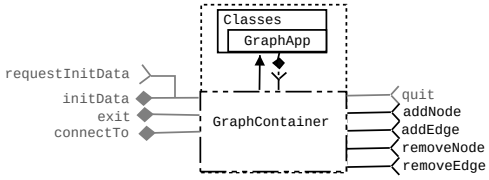

Figure 7. *Schéma du composant Graphe*

Composant document

Nos composants documents ont besoin, pour être construits et pour fonctionner, d'un ensemble d'informations supplémentaires telles que :

– le formalisme de base du langage (`formalism`);

– les diverses manières de lire un document (`parser`);

– les différentes vues associées par défaut (`lml`);

– les informations pour construire les menus spécifiques au langage dans l'interface utilisateur (`behavior`).

Comme les composants vues, les composants documents doivent respecter et réaliser un ensemble de services communs pour fonctionner. Ainsi, ils héritent tous d'un type de composant document standard (abstrait) décrit dans la figure 9. Par exemple, le service `launchVisitor`, commun à tous nos composants documents, permet de lancer un traitement (à base de visiteur) sur le document.

Ces informations supplémentaires (comme le formalisme) sont aussi utilisées pour générer les parties du code métier du composant. À partir du formalisme, les structures de données du composant sont produites, utiles pour la construction d'un document (arbre). Le formalisme sert à valider les données complexes sérialisées en XML, lors des échanges entre le composant document et ses vues. Par exemple, la figure 10 montre le descriptif de composant du langage TINY (notre langage jouet).

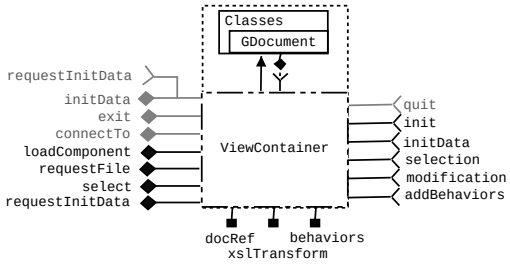

Figure 8. *Schéma du composant Vue*

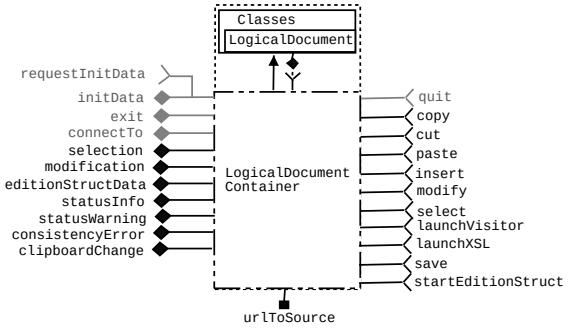

Figure 9. *Schéma du composant Document*

Les informations associées à l'analyseur syntaxique précisent les différents protocoles de lecture du document en fonction des extensions des fichiers. Pour produire ces divers protocoles (élément `parser` de la figure 10), elles indiquent aussi le type de générateur à utiliser (attribut `generator`) et le fichier de spécifications correspondant (attribut `file`).

Pour qu'un composant document puisse fonctionner en mode interactif, il est nécessaire de préciser les composants vues à instancier qui vont permettre d'interagir avec lui à travers l'interface utilisateur. Cette information (`lml`) définit, de manière locale, pour chaque composant document, la topologie associée. La topologie de l'application résulte de la juxtaposition des topologies associées à chaque composant document instancié. Dans la section suivante, ce mécanisme (format `lml`) sera précisé.

Un composant document doit pouvoir aussi préciser quels sont les services spécifiques à ajouter aux composants vues (`behavior`) pour les adapter à ce dernier. Tous ces services, lorsqu'une de ses vues est sélectionnée, doivent être accessibles à travers les menus associés à la vue. Ce mécanisme doit pouvoir s'appliquer sur n'importe quel type de vue et, en particulier, sur les vues génériques fournies par défaut.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="tiny" group="document" extends="logicaldocument">
   <formalism name="tiny" file="tiny.absynt" dtd="tiny.dtd"/>
   <containerclass name="TinyContainer"/>
   <facadeclass name="TinyFacade"/>
   <parser type="xml" classname="tiny.parsers.TinyXMLParser"
           extention name="tinyxml"/>
   <parser type="plain-text" generator="ANTLR" file="tiny.g"
           classname="tiny.parsers.TinyParser" extention name="tiny"/>
   <lml  name="DEFAULT" file="resources:tiny-default.lml"/>
   <behavior file="resources:tiny-behaviors.xml"/>
</component>
```

Figure 10. *Description du composant du langage* TINY

Finalement, ce modèle de composant a permis de rendre accessible à travers des spécifications abstraites (indépendantes d'une technologie), l'ensemble des services et possibilités de notre plate-forme.

4. Mise en œuvre

Pour l'implantation du modèle, un générateur de conteneur et un gestionnaire de composants ont été conçus en fonction des contraintes de notre plate-forme. Pour chaque application construite avec SMARTTOOLS, un fichier de lancement précise les composants à charger et à instancier au démarrage comme, par exemple, l'interface graphique. Enfin, les mécanismes d'extension des composants et d'exportation sont présentés afin de rendre les composants, respectivement, indépendants de tout langage et facilement exportables dans une autre technologie de composant.

Le générateur

Le générateur produit le conteneur du composant à partir de son descriptif. Il peut éventuellement étendre la façade si cette dernière est incomplète (vérifiée par introspection). Cette extension sert à mettre en place le mécanisme de communication façade vers conteneur (proche du patron de conception *Observer*) pour les services en sortie (`output`). Ce générateur produit aussi une archive (un paquetage de déploiement) contenant l'ensemble des classes dont le conteneur et les descriptifs du composant. Cette phase de génération rend transparents les moyens de communication mis en œuvre dans le conteneur et automatise les tâches de création de paquetage pour chaque nouveau langage traité.

Le gestionnaire de composants

Le gestionnaire de composants charge les paquetages des composants et crée les instances en fonction d'une description de lancement de l'application (figure 14) et des demandes interactives des utilisateurs (par exemple, l'ouverture du premier document associé à un langage donné). Il mémorise l'ensemble des paquetages chargés et des instances créées. En particulier, il offre le service `connectTo` pour connecter

deux composants (figure 11). Ce service permet aussi la création des composants s'ils n'existent pas encore. Cela assure qu'un fichier ne corresponde qu'à un seul composant document. Pour les connexions, le gestionnaire utilise les descriptifs respectifs de ces composantn fonction des noms, les connecteurs de sortie (*vs.* d'entrée) sont mis en relation avec les connecteurs d'entrée (*vs.* de sortie) de l'autre composant, s'ils existent. Après ce processus de connexion, les deux instances de composant dialoguent directement entre elles sans passer par le gestionnaire. La figure 12 décrit l'ensemble des services proposés par ce gestionnaire.

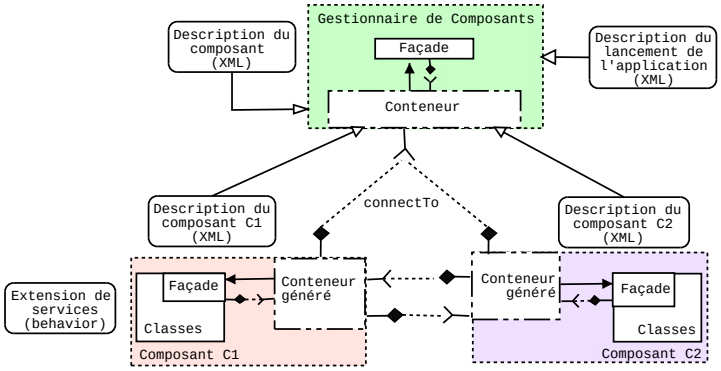

Figure 11. *Schéma de fonctionnement du gestionnaire*

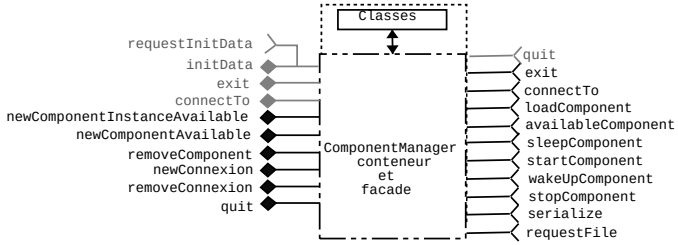

Figure 12. *Schéma du composant gestionnaire*

Interface utilisateur

L'interface utilisateur est réalisée suivant le même schéma que pour la visualisation d'un document : c'est une vue sur un arbre qui décrit l'ensemble des vues et des documents ouverts à un instant donné. Les actions sur l'interface (ajout d'un nouvel onglet, fermeture d'une vue, etc.) correspondent à des actions d'édition sur cet arbre. L'état de l'interface peut ainsi être sauvegardé (*serialisable* en XML). Cette sauvegarde peut jouer le rôle de fichier de configuration de l'application (voir la figure 13). Pour chaque vue, on indique le chemin vers le document à visualiser, le type de vue, la transformation à effectuer sur le document pour obtenir les objets graphiques de cette

vue et la feuille de style à appliquer sur ces objets. Pour un même document, il est donc possible d'avoir plusieurs vues graphiques associées.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:resources/lml.dtd">
<layout>
  <frame title="Smarttools V4.Alpha" statusBar="on" width="680"
         height="580" dynTabSwitch="off">
    <set title="Basic example">
      <split orientation="1" position="68">
       <view title="Layout in XML format"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="resources:lml/boot.lml"
              styleSheet="resources:css/xml-styles.css"
              transform="resources:xsl/genericXml.xsl" />
       <view title="samples/tiny/ex1.tiny"
              behavior=""
              viewType="fr.smarttools.core.view.GdocView"
              docRef="file:samples/tiny/ex1.tiny"
              styleSheet="resources:css/tiny-styles3.css"
              transform="resources:xsl/tiny-bml.xsl" />
        </split>
    </set>
  </frame>
</layout>
```

Figure 13. *Exemple d'un arbre de l'interface graphique (`boot.lml`)*

Lancement d'une application

Au démarrage de notre outil, le gestionnaire de composants est chargé et instancié par défaut. Puis, pour construire l'application, il va interpréter les actions d'un fichier de description de lancement. Les principales actions définies dans ce langage sont les suivantes :

– chargement d'un type de composant (`load_component`);

– création d'une instance (`start_component`);

– connexion entre deux composants (`connectTo`).

Par exemple, avec le fichier de lancement de la figure 14, il charge trois types de composant (`glayout`, `lml` et `tiny`) et établit une connexion entre lui-même et une instance du composant interface utilisateur (`glayout`). La création de cette dernière est paramétrée par les attributs de l'action `connectTo`. L'un de ces attributs (`docRef`) indique quel fichier de configuration de l'application est utilisé (*boot.lml* associé à l'application montrée dans la figure 2) pour instancier les composants documents et vues. L'ensemble des types de composant chargés (les rectangles) et les instances créées (les ovales) pour cet exemple sont décrits dans la figure 15.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<application repository="file:stlib/"  library="file:lib/">
  <load_component jar="view.jar"  name="glayout"/>
  <load_component jar="lml.jar" name="lml"/>
  <load_component jar="tiny.jar" url="file:extralib/tiny.jar" name="tiny"/>
  <connectTo id_src="ComponentManager" type_dest="glayout">
    <attribute name="docRef" value="file:resources/lml/boot.lml"/>
    <attribute name="xslTransform" value="file:resources/xsl/lml2bml.xsl"/>
    <attribute name="behaviors" value="file:resources/behaviors/bootbehav.xml"/>
  </connectTo>
</application>
```

Figure 14. *Exemple de descriptif de lancement correspondant à l'application de la figure 2*

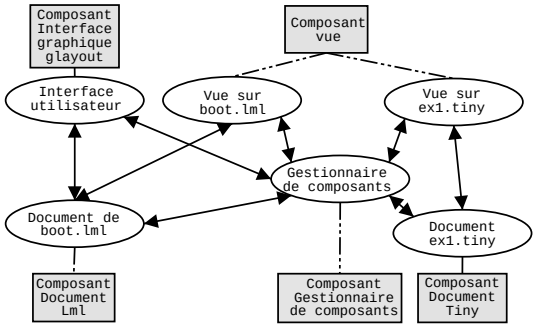

Figure 15. *Composants chargés et instances créées avec leurs connexions*

Après le lancement de l'application (en fonction des fichiers des figures 13 et 14), les instances de composant créées sont, comme montrées dans la figure 15, les suivantes :

– une instance du gestionnaire de composants (une seule instance possible de ce type de composant) ;

– une instance de composant document `lml` pour l'interface utilisateur et une instance pour le langage `tiny` ;

– trois instances de composant vue (deux pour l'interface et une pour le programme `ex1.tiny`).

Composants extensibles

Dans notre cadre, l'interface des composants de visualisation doit être extensible dynamiquement pour ajouter de nouveaux services. En effet, ceux-ci ne connaissent pas les services spécifiques des composants documents auxquels ils sont rattachés. Ils doivent donc être enrichis dynamiquement par ces services décrits dans un fichier d'extension du composant document (correspondant à l'élément `behavior` du descriptif du composant document). Ce fichier (voir figure 16) précise les éléments graphiques (menus et barre d'outils) à ajouter aux composants de visualisation (avec le

service `addBehavior` ou l'attribut de configuration `behavior` des composants vues) et les nouvelles connexions à établir entre ces composants et le document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<behaviors>
  <actions>
    <action name="Launch Visitor">
      <msg name="launchVisitor"
           chooser="fr.smarttools.core.document.LaunchVisitorDialogBox">
        <msgattr name="visitor"/>
        <msgattr name="aspect"/>
      </msg>
    </action>
    ...
  </actions>
  <menus>
    <menu name="File">
      <item action="Save Tiny"/>
    </menu>
    <menu name="Tiny">
      <item action="Launch Visitor"/>
      <item action="separator"/>
      <item action="Traversal + DebugAspect"/>
      ...
    </menu>
  </menus>
  ...
</behaviors>
```

Figure 16. *Exemple de services pour le langage* TINY *à rajouter à un composant vue (cf. menu de la figure 2)*

Une autre caractéristique du modèle est la possibilité d'ajouter des services qui agissent à l'intérieur même de nos composants. En effet, notre plate-forme fournit une technique de programmation par aspect [BOU 01, PAR 02] pour mieux spécifier (par *separation of concerns*) les traitements sur les arbres. Notre technique a l'avantage d'être totalement dynamique (sans transformation de programmes). L'effet de bord de cette possibilité est que les interfaces de nos composants documents doivent impérativement être extensibles. L'exemple type, que nous traitons déjà, est l'ajout d'un mode d'exécution pas-à-pas aux divers traitements, totalement réalisé à l'aide d'un aspect. Cet aspect utilise une vue graphique particulière et cela demande d'introduire dynamiquement de nouveaux services (dans les deux sens) entre le composant document et cette vue. Il nous reste encore à généraliser cette possibilité sur d'autres exemples plus complexes.

Détails de mise en œuvre

Comme notre outil est interactif, il était important que l'interface graphique ne soit pas bloquée lors de traitements sur les documents. Pour résoudre ce problème, chaque instance de composant est exécutée dans un processus indépendant (*Thread*) et le mode de communication est donc asynchrone (envoi d'événements stockés, à la réception, dans une file d'attente). Ces mécanismes sont transparents pour l'utilisateur car ils sont gérés au niveau du conteneur. Les mécanismes utilisés dans les

conteneurs générés sont simples et efficaces (appel direct des méthodes de la façade, *multi-threading*, *listener*, etc.).

Les types des données échangées entre les composants ont volontairement été limités pour éviter que les composants ne soient obligés de connaître tous les types échangés. Pour être échangées, les données plus complexes doivent être sérialisées en XML. Cela oblige à définir la DTD correspondante pour les valider. Cette contrainte est implicitement réalisée dans notre cadre puisque nos données complexes, principalement des arbres, sont forcément associées à une DTD.

Pour faciliter l'implantation de nos générateurs en Java, il existe des références explicites à des types ou méthodes Java dans notre modèle. Cette dépendance au langage Java, au niveau du modèle, pourrait être levée en introduisant des tables de correspondance pour divers langages de programmation.

Exportation des composants vers d'autres technologies

Pour valider notre approche, nous avons conçu un outil de transformation de nos descriptifs de composant vers les descriptions (ou interfaces) équivalentes pour les *Web Services*, les composants CORBA et les composants EJB. La figure 17 donne un aperçu des fichiers générés automatiquement par cet outil pour chaque technologie. Les détails techniques et particularités de cet outil sont décrits dans [VAR 02]. Notre composant de visualisation de graphe a ainsi été transformé automatiquement pour ces trois technologies. Cette transformation n'utilise que le descriptif du composant et aucun code source additionnel n'est à écrire. Par exemple pour les *Web Services* [3], cet outil traduit nos descriptifs de composant en des descriptifs *WSDL* (*Web Service Description Language*) équivalents. Par descriptif, il génère aussi le code source Java qui réalise les appels effectifs aux méthodes de la façade du composant (*SOAPBinding*). Ce code, normalement à la charge des développeurs, complète la génération de code source Java issu des outils associés aux *Web Services* comme AXIS. Ainsi, tous les services d'un composant peuvent être accessibles à travers un serveur Web comme *Tomcat*. Cet outil de transformation doit être considéré comme un prototype qui doit être approfondi sur des exemples plus complexes.

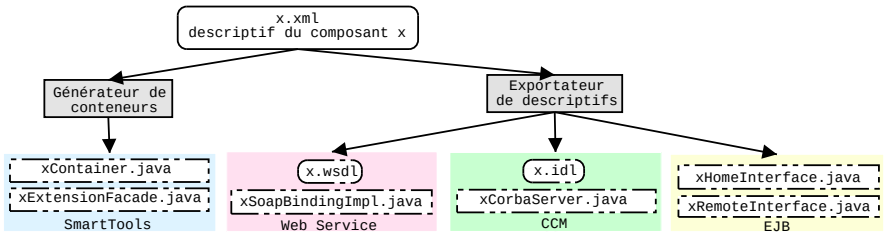

Figure 17. *Exemple de descriptif de composant*

3. L'une des technologies les plus simples de mise en œuvre et les plus proches de notre modèle.

5. Évaluation de notre modèle

Comme il existe un nombre important de modèles et de concepts, il est assez difficile de faire une comparaison précise et exhaustive de notre approche. Pour cela, nous utiliserons le patron d'évaluation, issu des principaux modèles de composants, proposé par [MAR 02] qui présente un état de l'art des différents modèles et une taxonomie des rôles dans le processus de production d'applications à base de composants. Nous allons regarder l'adéquation de notre démarche avec cette taxonomie pour préciser les avantages ou inconvénients de notre modèle.

Analyse des besoins en terme d'applicatif et de composant Notre modèle est issu d'une analyse des besoins spécifiques à notre plate-forme et est donc adapté aux applications produites.

Conception des types de composant Notre langage de description de composants permet de nous abstraire d'une technologie de composant particulière. Ce langage contient les trois aspects d'un composant : ses interfaces, ses propriétés, ainsi que la manière dont il coopère avec les autres composants.

Implantation des types de composant La partie non-fonctionnelle des composants est assurée par la génération automatique des conteneurs. Une partie des contraintes techniques (noms des conteneurs, des façades et des archives générés) est décrite dans les descriptifs des composants.

Diffusion des implantations de composants Notre générateur de conteneur prend en charge la constitution d'une archive (paquetage) pour chaque composant.

Assemblage des types de composant Les langages de déploiement et de description de l'interface graphique permettent de spécifier et de configurer l'assemblage des composants. Par contre, il n'est pas encore possible de considérer le résultat de cet assemblage comme un composant.

Déploiement des implantations de composant et des applicatifs Notre gestionnaire de composants joue le rôle de déploiement de composants. Il prend en charge la création de nouvelles instances et la recherche des instances existantes. Dans notre modèle, le déploiement n'est pas encore configurable en terme de structure d'accueil (par exemple, pour une version répartie sur différentes machines).

Utilisation des instances de composant et des applicatifs La découverte des services offerts par un composant s'effectue dans notre modèle exclusivement par l'intermédiaire des descriptifs des composants (format XML).

Au vue de notre expérience et de cette taxonomie des rôles, nous pouvons établir une liste de caractéristiques essentielles que tout modèle de composants devrait posséder, de notre point de vue, pour être flexible et évolutif. Ces caractéristiques sont les suivantes :

– Les descriptifs de composant doivent être indépendants de tout langage et de toute technologie, et proches des besoins de l'application. Cela induit une meilleure lisibilité des services offerts par les différents types de composant vu que le modèle

est adapté aux besoins (modèle conçu en fonction des besoins établis pendant l'analyse). L'utilisation des EJB implique *de facto* une approche client/serveur, CORBA une dépendance vis-à-vis de son protocole de communication (ORB - *Object Request Broker*) et les *Web-Services* une spécialisation Internet des composants. La construction de l'architecture d'une application devient aisée (même pour des débutants) si les composants à assembler ont des descriptifs compréhensibles (avec le vocabulaire du métier). L'interconnexion des composants est facilitée et est sans introspection car le générateur utilise des descriptifs au format neutre. Cette indépendance permet aussi de traduire ces descriptifs vers d'autres modèles de composants (WSDL, IDL, etc) dans le but de faciliter l'exportation des composants dans une application à technologie de composant différente.

– Les conteneurs doivent être produits automatiquement à l'aide d'un générateur. Le développeur se focalise seulement sur la partie métier, la valeur ajoutée, du composant lors du développement de ce dernier. Cette séparation entre la partie métier et la partie non-fonctionnelle est bénéfique car elle assure une indépendance de la partie métier vis-à-vis d'une technologie. Cette indépendance rend plus facile la projection vers d'autres technologies. Les dépendances d'un composant sont limitées aux entités (bibliothèques) utiles à son fonctionnement, sans faire référence à la technologie de composant utilisée. De plus, une application conçue à l'aide d'un générateur de conteneur a un meilleur potentiel d'adaptation et d'évolution. En effet, tout nouveau besoin ou toute nouvelle technologie (de communication, par exemple) peut, très rapidement, être intégré dans l'application, par simple modification de ce générateur. Par exemple, dans notre cadre, la prise en compte d'une version répartie devrait être facilement gérée par modification de notre générateur. Avoir son propre générateur permet aussi de produire du code spécifique à ses besoins (par exemple, pour effectuer des tests unitaires). La maintenance de la «glue» de l'application des composants est ainsi gérée automatiquement (*refactoring* d'applications) par le générateur.

– Il est important d'avoir une topologie locale, dynamique et configurable. Une architecture variable permet l'ajout de nouveaux composants, à la demande, en fonction des besoins. Dans notre plate-forme, il est possible de brancher dynamiquement un composant d'observation des messages échangés. Les composants doivent pouvoir localement décider de leurs connexions avec d'autres composants. Dans notre cadre, si un nouveau composant vue est créé, c'est lui qui va décider dynamiquement sa connexion avec le composant document associé. La construction de la topologie d'une application doit être décentralisée au niveau des composants eux-mêmes. Cette topologie doit aussi pouvoir être sauvegardée dans un fichier de déploiement pour donner la possibilité de relancer l'application dans le même état.

– Les composants doivent pouvoir communiquer directement entre eux. Cela évite les goulets d'étranglement au niveau d'un serveur d'application (gestionnaire de composants) et simplifie la mise en œuvre des communications (le gestionnaire de composants ne traite que des services liés à la création, au chargement, et à la connexion). Les composants restent autonomes.

– Pour la souplesse de l'application, il est important d'avoir des composants adaptables aussi bien pour les parties non-fonctionnelles que pour les parties fonction-

nelles. Pour les parties non-fonctionnelles, de telles possibilités ont été clairement identifiées dans le cadre des intergiciels [MAR 01]. Pour les parties fonctionnelles, notre expérience montre tout aussi clairement la nécessité d'un tel besoin.

6. Conclusion

La principale motivation de ce travail était de définir un modèle de composants en adéquation avec nos besoins et non d'en faire un modèle générique. Ce modèle est un moyen déclaratif de décrire l'architecture de SMARTTOOLS, indépendamment d'une technologie de composant et d'un langage de programmation. Les technologies existantes, comme les composants CORBA ou EJB, nous ont semblé être assez éloignées de nos préoccupations ou ne répondaient pas complètement à nos attentes. Pour autant, notre modèle s'appuie fortement sur les mêmes concepts de base mais sa mise en œuvre est adaptée à notre mode de fonctionnement. Nous avons préféré définir notre propre modèle, indépendamment des technologies existantes, tout en permettant une transformation vers celles-ci.

L'une des principales caractéristiques de SMARTTOOLS est d'être basée sur une approche par génération de code à partir de spécifications, non seulement pour les parties non-fonctionnelles des composants (conteneurs) mais aussi pour les parties fonctionnelles (API de manipulation des arbres, visiteur de parcours par défaut, etc.). Finalement, notre modèle caractérise les services spécifiques à notre fabrique d'applications orientées langages. Par comparaison, les modèles de composants, en particulier les industriels (EJB et CORBA), correspondent plus à des fabriques d'applications distribuées.

La principale contribution de nos travaux est de démontrer que ce concept de fabrique d'applications peut, d'une part, s'appliquer à différents niveaux dans une application (même sur les parties métiers) et, d'autre part, qu'elle doit être accessible à travers des modèles indépendants de toute technologie. Nous sommes plus convaincus de l'intérêt d'une approche par famille d'applications (fabrique) qu'à une approche de modèle générique de composants. En effet, il nous semble préférable de définir un modèle adapté au vocabulaire du métier sous-jacent. De plus, pour répondre aux besoins d'ouvertures et d'évolutions des applications, il est important que la fabrique (les générateurs) et les applications générées soient elles-mêmes adaptables.

Par son auto-utilisation dans sa propre construction, SMARTTOOLS offre un cadre pour tester concrètement les solutions proposées. C'est avant tout une plate-forme d'expérimentation pour nos futurs travaux de recherche, en particulier sur la composition d'aspects et de services pour les composants. De plus, les extensions naturelles de notre modèle (composant hiérarchique ou exécution répartie) offrent des perspectives et des problèmes (édition collaborative de documents) très intéressants.

Remerciements

Les auteurs tiennent à remercier les relecteurs qui, par leurs remarques constructives, ont permis d'améliorer la qualité de cet article. Ce projet est partiellement financé par le W3C dans le cadre du contrat européen QUESTION-HOW.

Carine Courbis *est en post-doctorant à University College London. Elle a un diplôme d'ingénieur de l'INSA de Lyon et un doctorat de l'Université de Nice Sophia Antipolis, préparé à l'INRIA. Elle s'intéresse à la programmation générative, la programmation par aspects, et les Web Services.*

Alexandre Fau *est ingénieur à ILOG. Il est titulaire d'un DEA en informatique de l'Université de Nice Sophia Antipolis. Il a travaillé comme ingénieur expert à l'INRIA Sophia Antipolis pour la réalisation de* SMARTTOOLS*, dans le cadre d'un contrat avec le W3C.*

Pascale Degenne *est chercheur au CIRAD. Il a un diplôme d'ingénieur du CNAM. Il a travaillé comme ingénieur expert à l'INRIA Sophia Antipolis pour la réalisation de* SMARTTOOLS*, dans le cadre d'un contrat avec le W3C. Il est maintenant intéressé par la télédectection, les systèmes d'information géographiques, et l'agronomie tropicale.*

Didier Parigot *est chercheur à l'INRIA Sophia Antipolis. Après avoir travaillé de nombreuses années dans le domaine des Grammaires Attribuées, il a recentré ses activités dans le domaine de l'ingénérie du logiciels. Plus particulièrement, il est intéressé par la programmation par composants, la programmation générative, la programmation par aspects et l'approche par transformation de modèle.*

7. Bibliographie

[BAT 98] BATORY D., LOFASO B., SMARAGDAKIS Y., « JTS: A Tool Suite for Building GenVoca Generators », *5th International Conference in Software Reuse*, june 1998.

[BEL 01] BELLOIR N., BRUEL J.-M., BARBIER F., « Formalisation de la relation Tout-Partie : application à l'assemblage de composants logiciels », *Actes des Journées composants : Flexibilité du système au langage (JC'2001)*, Besançon, France, October 2001.

[BOU 01] BOURAQADI-SAÂDANI N. M. N., LEDOUX T., « Le point sur la programmation par aspects », *Technique et Sciences Informatiques*, vol. 20, page 505 à 528, Hermès, 2001.

[BRU 01] BRUNETON E., RIVEILL M., « Experiments with JavaPod, a Platform Designed for the Adaptation of Non-functional Properties », *Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION 2001*, vol. 2192 de *LNCS*, Kyoto, Japan, sept 2001, p. 52–72.

[BRU 02] BRUNETON E., COUPAYE T., STEFANI J.-B., « Recursive and Dynamic Software Composition with Sharing », *In Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), June 2002.

[BéZ 01] BÉZIVIN J., « From Object Composition to Model Transformation with MDA », *TOOLS USA*, IEEE TOOLS-39, 2001.

[COU 03] COURBIS C., DEGENNE P., FAU A., PARIGOT D., « L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools », *revue L'Objet, numéro spécial XML et les objets*, vol. 9, n° 3, 2003, Hermès Sciences.

[GRO 00] GROUP O. S. S., SOLEY R., « Model-Driven Architecture », rapport, November 2000, OMG.

[HED 01] HEDIN G., MAGNUSSON E., « JastAdd—a Java-based system for implementing front ends », VAN DEN BRAND M., PARIGOT D., Eds., *Electronic Notes in Theoretical Computer Science*, vol. 44, Elsevier Science Publishers, 2001.

[Jon] « Jonas », http://jonas.objectweb.org/.

[KAS 98] KASTENS U., PFAHLER P., JUNG M., « The Eli System », KOSKIMIES K., Ed., *Compiler Construction CC'98*, vol. 1383 de *Lect. Notes in Comp. Sci.*, portugal, april 1998, Springer-Verlag, Tool demonstration.

[KLI 93] KLINT P., « A Meta-Environment for Generating Programming Environments », *ACM Transactions on Software Engineering Methodology*, vol. 2, n° 2, 1993, p. 176–201, Springer-Verlag.

[LIE 97] LIEBERHERR K. J., ORLEANS D., « Preventive Program Maintenance in Demeter/Java », *Proceedings of the 19th International Conference on Software Engineering*, ACM Press, 1997, p. 604–605.

[MAR 01] MARVIE R., MERLE P., GEIB J.-M., VADET M., « OpenCCM : une plate-forme ouverte pour composants CORBA », *Actes de la seconde Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, Paris, France, Avril 2001.

[MAR 02] MARVIE R., PELLEGRINI M.-C., « Modèles de composants, un état de l'art », *Numéro spécial de L'Objet*, vol. 8, n° 3, 2002, Hermès Sciences.

[MIG 02] DE MIGUEL M., JOURDAN J., SALICKI S., « Practical Experiences in the Application of MDA », *Lecture Notes in Computer Science*, vol. 2460, 2002, p. 128–??

[OCC 02] OCCELLO A., BLAY-FORNARINO M., DERY A.-M., RIVEILL M., « Vers une adaptation dynamique cohérente des composants », *Actes des Journées : Systèmes à composants adaptables et extensibles*, Grenoble, France, October 2002.

[PAR 02] PARIGOT D., COURBIS C., DEGENNE P., FAU A., PASQUIER C., FILLON J., HELP C., ATTALI I., « Aspect and XML-oriented Semantic Framework Generator: SmartTools », *ETAPS'2002, LDTA workshop*, Grenoble, France, April 2002, Electronic Notes in Theoretical Computer Science (ENTCS), ftp://ftp-sop.inria.fr/oasis/publications/2002/smartldta02.pdf.

[PAW 01] PAWLAK R., SEINTURIER L., DUCHIEN L., FLORIN G., « JAC: A Flexible Solution for Aspect-Oriented Programming in Java », *Lecture Notes in Computer Science*, vol. 2192, 2001, p. 1–??

[SZY 98] SZYPERSKI C., *Component Software: Beyond Object-Oriented Programming*, ACM Press and Addison-Wesley, New York, NY, 1998.

[VAD 01] VADET M., MERLE P., « Les containers ouverts dans les plate-forme à composant », *Actes des Journées composants : Flexibilité du système au langage (JC'2001)*, Besançon, France, October 2001.

[VAR 02] VARIAMPARAMBIL J. G., « Enabling SmartTools components with component technologies:WebServices, CORBA and EJBs », rapport, 2002, INRIA.

Annexe pour le service de fabrication

Article pour la revue :
Technique et science informatiques

Auteurs :
Carine Courbis — Pascal Degenne** — Alexandre Fau** — Didier Parigot**

Titre de l'article :
Un modèle abstrait de composants adaptables

Titre abrégé :
Un modèle abstrait de composants

Traduction du titre :
an abstract model of adaptable components

Date de cette version :
November 25, 2003

Coordonnées des auteurs :

– téléphone : 04 92 38 50 01
– télécopie : 04 92 38 76 44
– Email : Didier.Parigot@inria.fr

Logiciel utilisé pour la préparation de cet article :
LaTeX, avec le fichier de style `article-hermes.cls`,
version 1.4 du 1998/07/17.

Formulaire de copyright :
Joindre le formulaire de copyright signé, récupéré sur le web à l'adresse `http://www.hermes-science.com`