Un modèle de composants pour l'atelier de développement SMARTTOOLS

Carine Courbis* — Pascal Degenne — Alexandre Fau** — Didier Parigot* — Joseph Variamparambil***

* *INRIA Sophia Antipolis Projet OASIS et ** W3C*
2004, route des Lucioles BP 93
06902 Sophia-Antipolis cedex, France

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

Prénom.Nom@sophia.inria.fr

RÉSUMÉ. *Depuis quelques années, l'approche par composants dans la conception de logiciels complexes est apparue pour résoudre les problèmes de répartition, de déploiement et de réutilisation de code. Cette approche s'avère 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 some years, the component approach for the design of complex softwares has appeared to solve problems of code repartition, deployment and reusability. This approach turns out to suit the best to insure application evolution, interoperability and maintenance. This article introduces the component model used to realize an Integrated Development Environment. The original 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 experimented as it was used to implement our tool. This pragmatic approach was helpful for us to design an easy and realistic component model.*

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

KEYWORDS: *Component model, XML technologies*

1re soumission à *Systèmes à composants adaptables et extensibles 15 juillet 2002 19:09)*, le 15 juillet 2002.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

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. L'approche objet ne permet pas d'exprimer des interconnexions complexes entre les objets. Elle est trop orientée vers la programmation et se situe à un niveau trop fin. L'approche composant [MAR 02, BRU 01] est apparue pour une meilleure séparation entre les aspects de communication et les parties fonctionnelles, pour s'abstraire des langages de programmation et pour construire des applications par assemblage de composants.

Cet article décrit une approche composant dans le contexte particulier de la conception d'un atelier de développement, nommé SMARTTOOLS [PAR 02]. C'est un générateur d'environnements de développement pour des langages de programmation ou métiers qui s'appuie fortement sur les technologies objets et XML [COU 02]. A partir d'une description d'un langage, il produit automatiquement des outils de transformation ou de manipulation des programmes écrits dans ce langage. Ces programmes (ou documents) sont représentés en interne sous forme d'arbres de syntaxe abstraite. Ainsi, l'utilisateur obtient rapidement un environnement de développement. Il peut ensuite l'enrichir, à l'aide de SMARTTOOLS, avec d'autres éléments spécifiques 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 visiteur (*visitor design pattern*) ou tout simplement des transformations écrites en XSL (*eXtensible Stylesheet Language*) du W3C (*World Wide Web Consortium*).

SMARTTOOLS est constitué d'outils de base et de composants spécifiques pour chaque langage traité. Ces outils et composants offrent et requièrent des services qu'il faut connecter et rendre accessibles à travers une interface utilisateur. Par exemple, chaque manipulation au niveau d'une vue graphique (sélection, remplacement, insertion ou appel d'un traitement) va se traduire par l'invocation des services au niveau du document associé. L'aspect générique de SMARTTOOLS, un méta-outil, impose que son interface utilisateur et son architecture soient les plus configurables et extensibles possibles pour pouvoir les adapter aux services spécifiques de chaque nouveau langage traité. Il est impératif que toutes les entités logicielles produites soient autonomes et puissent être exécutées sans nécessiter la présence de l'interface, par exemple en mode commande (*batch*) ou dans d'autres environnements. Il doit aussi être facile d'ajouter des composants externes en ne modifiant ni la plate-forme, ni ces composants.

Dans les premières versions de l'outil, l'absence d'une vraie approche par composants était un handicap pour son évolution. Nos composants étaient indissociables d'un bus logiciel [BER 98] qui assurait toutes les communications. Toute intégration et toute exportation de composants impliquaient la présence de ce bus. L'approche

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

orientée composant s'est donc imposée pour la réalisation de notre outil. Aucun des modèles existants, tels que les *Web Services* [W3C], les composants CORBA (*CCM - CORBA Component Model*) [COR, MAR 01] ou les EJB [EJB], ne semblait correspondre parfaitement à nos besoins spécifiques et cette dépendance vis-à-vis d'une technologie nous paraissait préjudiciable à l'évolution de l'outil. Nous avons donc conçu notre propre modèle (indépendamment de ces technologies) adapté et restreint à nos besoins, mais surtout transposable vers ces technologies de composants.

Il ressort, de nos premières expériences, que concevoir un modèle abstrait permet d'être en parfaite adéquation avec les besoins et contraintes du champ d'application. Nos langages de description de composants sont ainsi adaptés et offrent un moyen déclaratif pour décrire précisément les composants et l'assemblage qui forme une application. L'implantation effective du modèle (par génération de conteneurs à partir de ces descriptifs) prend en compte toutes les particularités de notre domaine sans pour autant nuire à la lisibilité de ces descriptifs. Cette génération de conteneurs automatise la programmation des parties non-fonctionnelles des composants. Elle facilite l'ajout de nouvelles fonctionnalités, non initialement prévues. 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 ou encore les EJB. 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.

L'article se décompose en quatre sections. La première section décrit le modèle de composants et son implantation. La deuxième section traite des particularités dynamiques de ce modèle et présente la projection des composants vers ces technologies existantes. La troisième section évalue notre modèle suivant des critères communément admis [MAR 02]. Enfin, la dernière section dresse le bilan de cette approche.

2. Description du modèle et de sa mise en œuvre

Cette section introduit notre modèle de composants et son implantation. Lors de la conception de ce modèle, nous nous sommes imposés deux contraintes : 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. En respectant ces contraintes, nous avons conçu un langage de description de composants. Puis pour l'implantation, un générateur de conteneurs et un gestionnaire de composants ont été réalisés.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

2.1. *Le langage de description de nos composants*

Avec ce langage, on décrit le type d'un composant : les services (connecteurs) en entrée ou en sortie, les modes de communication et les attributs de configuration de l'état du composant. Pour chaque service, on indique son nom logique, la méthode à appeler dans la partie métier (dans la façade du composant), les paramètres (`parameter`) de la méthode. Les modes de communication sont soit asynchrone (`input` ou `output`), soit synchrone (`inout`). Il est possible de factoriser les parties communes des descriptions dans des descriptions abstraites. Par exemple, la description abstraite `abstractComponent` (Figure 1) est importée par toutes les autres descriptions. La Figure 2 donne la description du composant Graphe[1] et la Figure 3 sa représentation graphique.

```
<?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 1: *Description abstraite abstractComponent*

```
<?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 2: *Description du composant Graphe*

1. Les autres composants sont trop complexes pour être donnés en exemple.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

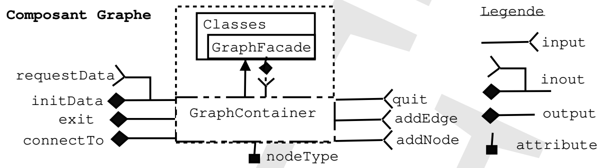

Figure 3: *Schéma du composant Graphe*

2.2. *La mise en œuvre du modèle*

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.

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 5) 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. Il offre aussi le service `connectTo` pour connecter deux composants (Figure 4). Ce service effectue aussi la création des composants s'ils n'existent pas encore. Pour les connections, le gestionnaire utilise les descriptifs respectifs de ces composants. En 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 connection, les deux instances de composant dialoguent directement entre elles sans passer par le gestionnaire.

Exemple de lancement d'une application

Au démarrage de notre outil, le gestionnaire de composants est chargé et créé automatiquement. Puis avec le fichier de lancement de la Figure 5, il charge trois types de composants (`glayout`, `lml` et `tiny`) et établit une connection entre lui-même

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

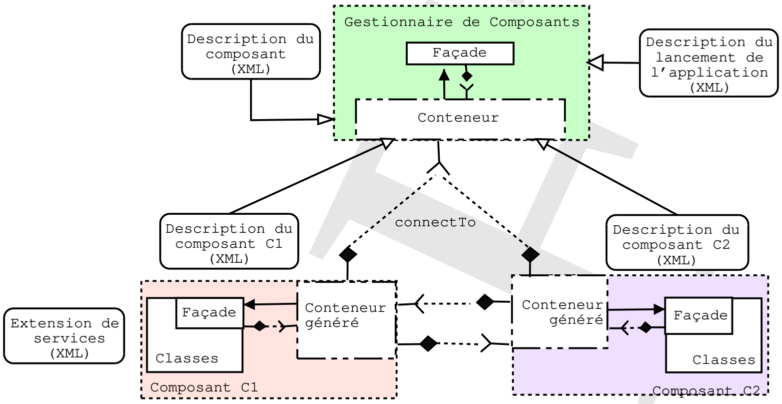

Figure 4: *Schéma de notre modèle de composant*

et une instance du composant interface utilisateur (`glayout`). La création de cette instance est paramétrée par les attributs du service `connectTo`. Par exemple, le fichier de configuration *boot.lml* (Figure 6) décrit les documents et les vues à ouvrir dans cette instance.

```
<?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 5: *Exemple de descriptif de lancement*

Interface utilisateur

L'interface utilisateur est réalisée suivant le même schéma que pour un programme : c'est une vue sur un arbre qui décrit l'ensemble des vues et des documents ouverts à un instant donné. L'état de l'interface peut ainsi être sauvegardé (*serialisable*). Les actions sur l'interface (ajout d'un nouvel onglet, fermeture d'une vue, etc) correspondent à des actions d'édition sur cet arbre. Le fichier de configuration `boot.lml` de l'interface utilisateur (voir la Figure 6) donne l'arbre dans un format XML. 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 possible d'avoir plusieurs vues graphiques associées.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:resources/lml.dtd">
<layout>
  <frame title="Smarttools V4.Alpha" statusBar="on">
    <set title="LML Layout">
      <view title="Layout" behavior="" viewType="fr.smarttools.view.GdocView"
            docRef="file:resources/lml/boot.lml"
            styleSheet="file:resources/css/lmlstyles.css"
            transform="file:resources/xsl/genericBoxes.xsl" />
    </set>
    <set title="Tiny Workbench">
      <view title="samples/tiny/ex1.tiny" behavior=""
            viewType="fr.smarttools.view.GdocView"
            docRef="file:samples/tiny/ex1.tiny"
            styleSheet="file:resources/css/tiny-styles.css"
            transform="file:resources/xsl/genericBoxes.xsl" />
    </set>
  </frame>
</layout>
```

Figure 6: *Exemple d'un arbre de l'interface graphique (`boot.lml`)*

La Figure 7 montre les types de composant chargés (les rectangles) et les instances créées (les ovales) pour cette configuration (les fichiers des Figures 5 et 6).

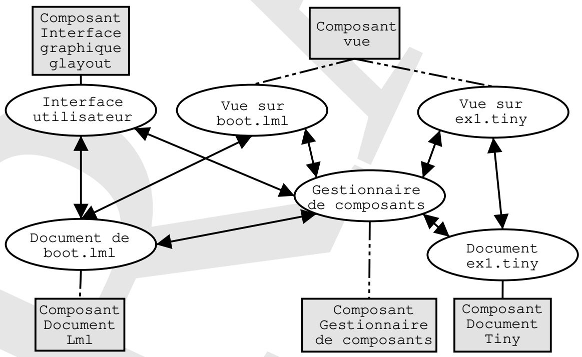

Figure 7: *Composants chargés et instances créées avec leurs connections*

Détails d'implantation

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 qui est généré. Les mécanismes utilisés dans les conteneurs générés sont simples et efficaces (appel direct des méthodes de la façade, *multi-threading*, *listener*, etc). Ce modèle de composants est opérationnel dans la version courante de SMARTTOOLS.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

3. Avantages du modèle

3.1. *Composants extensibles*

Dans notre cadre, l'interface des composants doit être extensible dynamiquement pour l'ajout de nouveaux services, par exemple, aux composants de visualisation. 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. Ce fichier précise les éléments graphiques (menus et barre d'outils) à ajouter aux composants de visualisation et les nouvelles connections à établir entre ces composants et le document.

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.

3.2. *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 8 donne un aperçu[2] des fichiers générés automatiquement par cet outil pour chaque technologie.

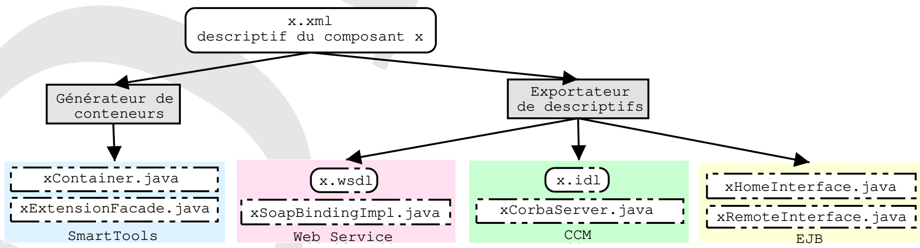

Figure 8: *Exemple de descriptif de composant*

2. Donner tous les détails techniques et particularités nuirait à la lisibilité de cette Figure.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

Mais surtout, il ressort de cette expérience que l'utilisation d'une de ces technologies correspond à faire un choix de conception dont on ne peut s'abstraire par la suite. Utiliser EJB implique un modèle 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. En d'autres termes, concevoir un modèle spécifique à ses besoins évite de choisir une solution trop liée à une technologie existante. De plus, certains de nos besoins (extensions de services) sont difficilement exprimables dans ces différents modèles. Par exemple, il faudrait spécialiser les composants vues pour chaque langage en fonction des services proposés par le composant document associé.

Les intérêts de définir un modèle abstrait sont les suivants :

– spécifications appropriées aux fonctionnalités des composants sans transposition des besoins et sans les contraintes liées à une technologie ;

– implantation efficace et en parfaite adéquation avec les contraintes de l'application ;

– projection plus simple vers les modèles existants.

4. 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. Nous utiliserons plutôt 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 à ce domaine d'application.

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 types de 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.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

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 d'une application. 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 ce 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). Nous utilisons une introspection sur ce format neutre, indépendant d'un langage de programmation particulier.

Bien que notre modèle soit dédié à notre outil, il possède de nombreuses bonnes propriétés d'un modèle de composants en regard de cette taxonomie. Il contient tout de même quelques particularités et restrictions.

La topologie d'une application est dynamique et configurable. Ce changement de topologie s'effectue en cours d'exécution et la prise de décision de connection est locale à un composant. Par exemple, si un nouveau composant vue est créé, c'est lui qui va prendre en charge dynamiquement sa connection avec le composant document associé.

Les possibilités d'extension de nos composants pour la qualité de service (comme les besoins de sécurité, de persistance et de transaction) sont actuellement peu traitées dans notre modèle. Ces besoins n'étaient pas vitaux pour un premier fonctionnement de notre plate-forme. De plus, l'exportation de nos composants vers d'autres modèles nous affranchit de fournir de tels services couramment proposés par les autres modèles. En les exportant, ils sont ainsi utilisables dans des applications ou environnements plus complexes qui requièrent ces types de services. Par contre, notre outil demande des services particuliers comme par exemple, la persistance des sessions d'édition. Pour cela, le choix de représenter le contenu de l'interface utilisateur sous la forme d'un arbre a permis de réaliser facilement ce service.

Mais l'intérêt d'un modèle par composant est de laisser ces parties non-fonctionnelles à la charge du gestionnaire de conteneurs. Ainsi, pour intégrer ces types de services afin de faire évoluer notre plate-forme, il suffit de modifier ce générateur. Par exemple, la réalisation d'une version répartie devrait être facilitée par ce générateur. De même, notre outil de transformation pourrait être étendu pour d'autres modèles de composants (par exemple ceux proposés par la plate-forme ObjectWeb [Obj]).

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

5. 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 général. Ce modèle est un moyen déclaratif de décrire l'architecture de SMARTTOOLS, indépendamment d'une technologie de composants 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. Cette démarche correspond à la nouvelle stratégie défendue par l'OMG, MDA (*Model-Driven Architecture*) [BéZ 01], 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 - *Plateform Independant Model*) et ensuite de proposer différentes transformations vers des modèles spécifiques (PSM - *Plateform Specific Model*).

Les perspectives intéressantes de cette approche sont de pouvoir étendre les composants en modifiant leurs comportements internes à l'aide de la technique de programmation par aspect. 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 mais aussi pour les parties fonctionnelles. Pour les composants documents, cette approche est utilisée pour offrir une technique dynamique de programmation par aspect au-dessus des visiteurs. Pour répondre aux besoins d'extensions ou de *separation of concerns* il existe essentiellement trois types d'approche [BOU 01] : par transformation de programme, par mécanisme de réflexivité [MAL 96] (MOP - *Meta-Object Protocol*) et enfin par génération de code. L'intérêt de l'approche par génération (conteneur ou visiteur) est de produire un code incluant directement les mécanismes appropriés d'extension. Cette approche, bien que non généraliste, rend ces mécanismes transparents à l'utilisateur et reste simple et efficace dans son implantation (vu nos premiers résultats).

Tous ces travaux cherchent à résoudre les problèmes rencontrés lors de la conception et réalisation d'un tel méta-outil. 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.

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -

Remerciements

Ce projet est partiellement financé par le W3C dans le cadre du contrat européen QUESTION-HOW[3].

6. Bibliographie

[BER 98] BERGSTRA J., KLINT P., « The discrete time ToolBus – A software coordination architecture », *Science of Computer Programming*, vol. 31, n° 2-3, 1998, p. 205–229.

[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.

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

[COR] « CORBA 3.0 CCM », http://www.omg.org/.

[COU 02] 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*, , 2002, Hermès Sciences, à paraître, ftp://ftp-sop.inria.fr/oasis/publications/2002/smartobjet02.pdf.

[EJB] « Entreprise Java Beans », http://java.sun.com/products/ejb.

[MAL 96] MALENFANT J., COINTE P., « Aspect-Oriented Programming versus Reflection », rapport, 1996, Ecole des Mines de Nantes.

[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.

[Obj] « ObjectWeb », http://www.objectweb.org/.

[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.

[W3C] « http://www.w3.org/ », The World Wide Web Consortium.

3. QUality Engineering Solutions via Tools, Information and Outreach for the New Highly-enriched Offerings from W3C (voir `http:// www.cordis.lu/ist/projets.html` pour plus d'information)

DRAFT -- DRAFT -- DRAFT -- DRAFT -- DRAFT -