Fabrique Logicielle SmartTools
INRIA

Encadrants :
PARIGOT Didier

Etudiants :
CIMINERA Frédéric
DAHON Benjamin
LEVENEUR Daniel

L'objectif de ce projet est de nous familiariser avec la programmation générative grâce à l'outil SmartTools.

Remerciements

Nous tenons tout d'abord à remercier Didier Parigot, chargé de recherche à l'Inria, de nous avoir proposé ce sujet et pour ses explications et sa disponibilité pendant toute la durée du projet.

Contexte du projet

L'environnement et les technologies dans le domaine informatique changent très rapidement en gardant en même temps une très haute diversité. Ce n'est alors pas seulement le développement des nouveaux outils informatiques et des logiciels mais aussi l'adaptation des moyens informatiques existants à l'environnement changeant qui est la préoccupation principale des créateurs de logiciels. Il est par conséquent très important pour les créateurs des logiciels de les protéger contre les bouleversements d'environnements informatiques et contre les changements des technologies.

C'est en utilisant les méthodes de programmation générative qu'on essaie de trouver des solutions pour protéger les logiciels contre les changements rapides dans le domaine informatique.

SmartTools est un générateur d'environnements de développement fortement fondé sur les technologies objets et XML. Grâce à une technique de génération automatique à partir de spécifications, SmartTools permet de développer très rapidement des environnements spécialisés pour des langages de programmation ou des langages métiers. En particulier, certaines de ces spécifications sont directement issues des technologies W3C, ce qui ouvre, des perspectives innombrables et variées de langages métiers. D'autre part, SmartTools s'appuie sur les technologies objets : utilisation des "visitor patterns", programmation par aspects, distribution des objets et composants, et une implantation en Java. Les avantages de ces technologies permettent de proposer, à moindre coût, une plate-forme de développement ouverte, interactive, uniforme, et évolutive.

La plate-forme SmartTools, conçue et developpée par l'équipe Oasis de l'INRIA, est un générateur de composants pour des ateliers de développement interactifs fondés sur les nouvelles technologies XML, objets (programmation par aspects) et composants (EJB, Web Services). L'un des objectifs principaux est de promouvoir et de valider de nouveaux concepts de développement très innovants pour le développement de logiciel pour les langages dédiés. En effet, SmartTools s'intègre parfaitement dans la nouvelle stratégie de conception de logiciel défini par l'OMG pour les années à venir (Model-Driven Architecture, MDA). La grande force de cette approche est l'unification des technologies liées aux langages (programmation par aspect), au Web (XML) et aux composants (architecture logicielle). Ainsi, le champ d'application de SmartTools est potentiellement trés vaste.

Table des matières

Table des figures

Section 1

Decription du cahier des charges

1.1 Définitions et acronymes

- ABSYNT : langage déclaratif utilisé pour définir une syntaxe abstraite d'un langage (équivalent à une grammaire)
- BEHAVIOR : language permettant de décrire le comportement d'un composant.
- BML : Bean Markup Language
- CDML (Component Description Markup Language) : langage qui décrit un composant.
- COSYNT : spécification composée de 2 parties : la première appelée BNF qui décrit la syntaxe concrète d'une grammaire et la seconde qui décrit l'affichage.
- CSS : Cascading Style Sheet
- DTD : Document Type Definition
- HUTN : Human-Usable Textual notation
- JAR : Java Archive
- JAVA : Langage de programmation multi-plateforme orienté objet crée par Sun
- LML (Layout Markup Language) : langage basé sur XML décrivant la structure d'une interface graphique définie par l'utilisateur.
- OMG : Object Management Group
- MDA : Model Driven Architecture
- UML : Unified Modeling Language
- W3C : World Wide Web Consortium
- XML : Extensible Markup Language
- XPATH : XML Path Language
- XSD : XML Schema Définition
- XSL : Extensible StyleSheet Language
- XSLT : XSL Transformation

Pour plus de détails, veuillez vous référer à http ://www-sop.inria.fr/smartool/distrib

1.2 Objectifs

Notre objectif principal consistait en la conception d'un **guide utilisateur** pour faciliter la tâche de construction d'un composant SmartTools. Cela nous a demandé d'une part, de perfectionner un outil de génération de boite de dialogue puis d'autre part d'approfondir les divers environnements des langages dédiés de SmartTools et enfin de perfectionner les divers transformations entre les méta-langages.
Le résultat de notre projet est un **environnement de "prise en main"** de l'outil SmartTools qui permet sans trop de difficulté d'utiliser toutes ces possibilités.

1.3 Lieu de Travail

Notre projet s'est déroulé dans les locaux de l'Inria et a été supervisé par Monsieur Didier Parigot, chercheur à l'Inria.

1.4 Moyens de contrôle

Afin de respecter les délais, Monsieur Parigot a mis en place différentes tâches à éffectuer afin de réaliser nos objectifs (cf section 3). Une fois une tâche éffectuée, il se chargeait de valider notre travail et de nous affecter à une autre tâche.

1.5 Moyen et outils utilisés

Le projet a été intégralement réalisé avec l'outil SmartTools.Nous avons néanmoins du écrire diverses fonctionnalités où nous avons utilisé les technologies Java, XML, XSL, CSS.
Nous disposions également d'un serveur CVS qui nous permettait de centraliser toutes nos données.

Section 2

Présentation de SmartTools

2.1 Principe de fonctionnement de SmartTools

SmartTools, développé entièrement à l'aide des technologies JAVA et XML, est un environnement de développement qui fournit une édition structurée et des outils sémantiques . Un des objectifs de SmartTools est de générer tous les outils nécessaires à l'implémentation de langages (métiers ou non).

Plus précisément, à partir d'une description d'un langage (Absynt, DTD ou Schema), la plate-forme génère un environnement de développement contenant un analyseur d'une forme concrète du langage (parser), l'afficheur associé (pretty-printer), un éditeur syntaxique et un ensemble de fichiers Java facilitant l'écriture de traitements sémantiques (analyses, transformations).

Nous allons par la suite définir quelques termes qui seront utiles pour comprendre le principe de fonctionnement de SmartTools.

Composants
La notion de composant est centrale dans la plate-forme SmartTools. Vous verrez par la suite la méthode pour créer un composant. Un composant est en fait un fichier JAR, comprenant l'ensemble des ressources nécessaires au bon fonctionnement de ce dernier.
SmartTools peut être vu véritablement comme une plate-forme à composants. En effet, l'unité principale est le composant, et chacun peut "dialoguer" avec les autres, cet échange de messages étant rendu possile par l'interconnexion de chaque composant à un un gestionnaire de composants (ComposantManager).
Ce gestionnaire permet l'échange de messages avec les composants en utilisant le conteneur associé à chacun d'entre eux. Un conteneur est une sur couche au composant qui encapsule un ensemble de services, générés automatiquement depuis une description formelle du composant, description que nous aborderons dans la prochaine partie. Ci-dessous, le principe de fonctionnement du gestionnaire de composants.

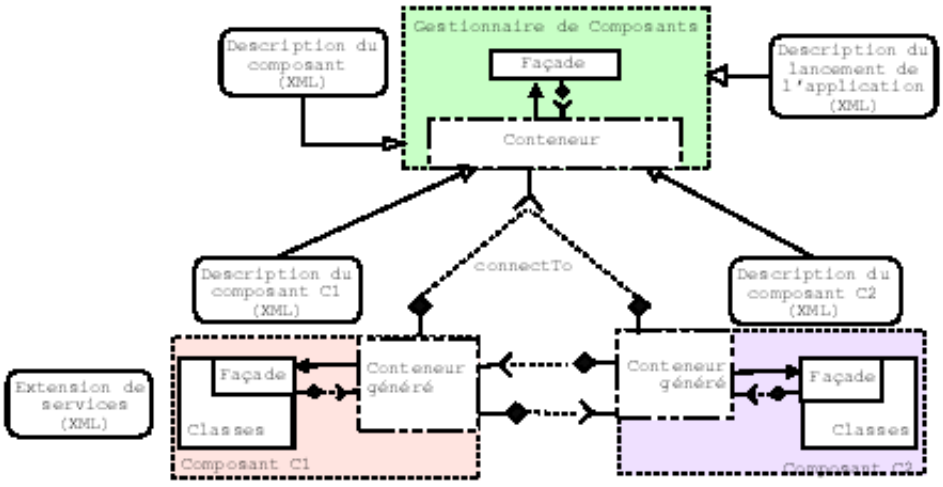

Fig. 2.1 – Principe de fonctionnement du gestionnaire de composants

Un composant est décrit de manière abstraite par l'intermédiaire du langage CDML. Cette étape de description des composants rend possible la projection du modèle de composants SmartTools vers d'autres plate-formes à composants (EJB, Composants CORBA). Afin de mieux comprendre le modèles de composants mis en oeuvre dans SmartTools, nous allons vous décrire brièvement les spécifications du langage CDML.

CDML

L'intérêt majeur de CDML est de permettre la description (sous forme de fichier XML) du comportement du conteneur. Vous trouverez ci-dessous, un exemple de fichier CDML.

```
<!-- Graph Component Declaration  -->
<component name="graph"
          type="graph"
          extends="abstractContainer"
          ns="graph">
   <containerclass name="GraphContainer"/>
   <facadeclass name="GraphApp"/>
       <dependance name="koala-graphics"
                   jar="koala-graphics.jar"/>
   <input name="addNode" doc="Add a graph node (Component Instance)"
             method="addNode">
      <attribute name="nodeName" ns="graph" doc="node name"
                    javatype="java.lang.String"/>
      <attribute name="nodeType" ns="graph" doc="node type (rect, oval)"
                    javatype="java.lang.String"/>
      <attribute name="nodeColor" ns="graph"
                    doc="node color (red, yellow, orange, white)"
                    javatype="java.lang.String"/>
   </input>

   <input name="addEdge" doc="Add a graph edge" method="addEdge">
```

```
        <attribute name="srcNodeName" ns="graph" doc="src node name"
            javatype="java.lang.String"/>
        <attribute name="destNodeName" ns="graph" doc="dest node name"
            javatype="java.lang.String"/>
    </input>
</component>
```

Dans l'exemple précédent, notre composant se nomme Graph, et il expose différentes opérations qui peuvent être vues graphiquement de la façon suivante :

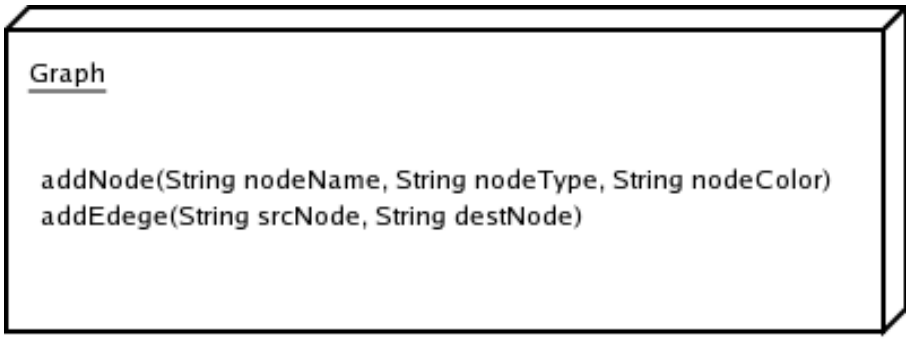

Fig. 2.2 – Composant Graph

Vous trouverez en annexe, la DTD relative au langage CDML. Le modèle de composants présenté dans SmartTools est composé des éléments de base suivants :

- le nom du composant (attribut name de <component>)
- le nom du type étendu (attribut extends de <component>)
- le nom du container(attribut name de <containerclass>)
- la classe de la facade du composant (attribut name de <facadeclass>)
- les differentes archives nécessaires au fonctionnement du composant(éléments <dependance>)
- les attributs
- les différents services exportés par ce composant

Absynt

Le langage Absynt correspond à la syntaxe abstraite de notre composant. On peut le voir comme une grammaire (DTD) d'un langage.

Absynt est caractérisé par 3 notions principales :

- Opérateur :noeud de l'arbre de la syntaxe abstraite.
- Type : ensemble d'opérateurs et de Types.
- Attribut : décoration de l'arbre de la syntaxe abstraite.

Pourquoi un tel formalisme ?

Le langage Absynt, indépendant de tout langage de programmation, a été créé afin d'obtenir une définition, de haut niveau, des syntaxes abstraites des langages. A partir d'une telle définition de langage, on génère des classes Java construites au-dessus de l'API DOM ; ces classes serviront pour représenter les noeuds de ces arbres. Ainsi les concepteurs de langages se concentrent uniquement sur cette définition, non sur l'implémentation du langage.

Cosynt

Cosynt correspond à une syntaxe concrète de notre composant. Il peut y a voir plusieurs cosynt associés au même composant. Il est composé de 2 parties principales :

- BNF qui décrit la syntaxe concrète du modèle.

– une autre partie qui s'occupe de l'affichage.

Architecture

Comme vu précédemment, chaque composant doit obligatoirement comporter un fichier Absynt, définissant le langage sur lequel s'appuie le composant. De nombreuses actions sont mises en jeu lors de la création d'un composant. Le schéma ci-dessous résume l'ensemble des actions effectuées depuis la définition d'un Absynt.

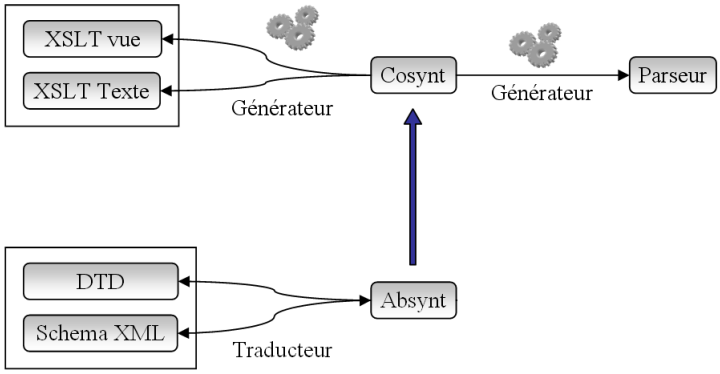

Fig. 2.3 – Ensemble d'actions effectués depuis l'Absynt

Cet Absynt peut être importé depuis les formalismes :
– DTD
– XML Schema (cf. section 4.1 - 4.2)

Pour chaque cosynt crée, un parseur Java est généré, conformément à la description Absynt du composant. De plus plusieurs XSLT sont produits :
– XSLT correspondant à la vue
– XSLT correspondant au texte

2.2 Environnement de Développement

Un des objectifs finaux du projet a été de produire un environnement de développement pour différents langages grâce à l'outil SmartTools et donc **sans écrire une ligne de code**, on peut ainsi à partir d'une définition d'un langage existant (DTD ,XSD ou schema UML) avoir automatiquement un composant associé.

L'environnement de développement permet les opérations suivantes :
– Créer la description du composant (cf. cdml) et ses repertoires associés
– Créer ou générer la syntaxe abstraite du langage (cf. absynt)
– Créer la definition graphique pour ce composant (langage lml)
– Générer le composant et produire une archive contenant tous les fichiers du composant (fichier jar)
– Ouvrir et éditer les différents documents du composant
– Créer des syntaxes concrètes pour le composant (cosynt)
– Fournir des environnements CSS et XSL qui permettent de tester la syntaxe concrete
– Créer des définitions pour le comportement des composants(Behavior)

Ci dessous une capture d'écran de la fenêtre principale de l'environnement de développement.

Nous vous expliquerons par la suite les différentes parties developpés qui nous ont permis de réaliser cet environnement.

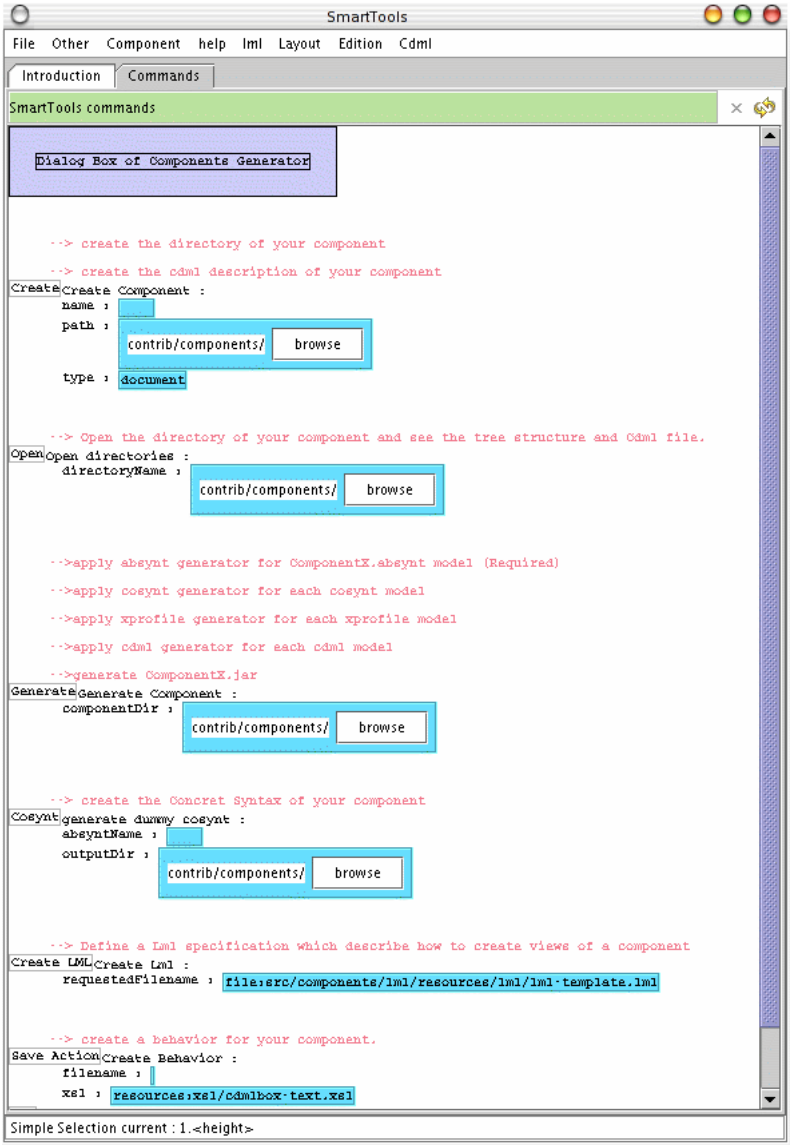

Fig. 2.4 – Fonctionnalité de l'environnement de développement

Section 3

Organisation du projet

3.1 Processus

Vous trouverez ci dessous la décomposition en tâches de notre projet et nous détaillerons par la suite chacune de ces tâches.

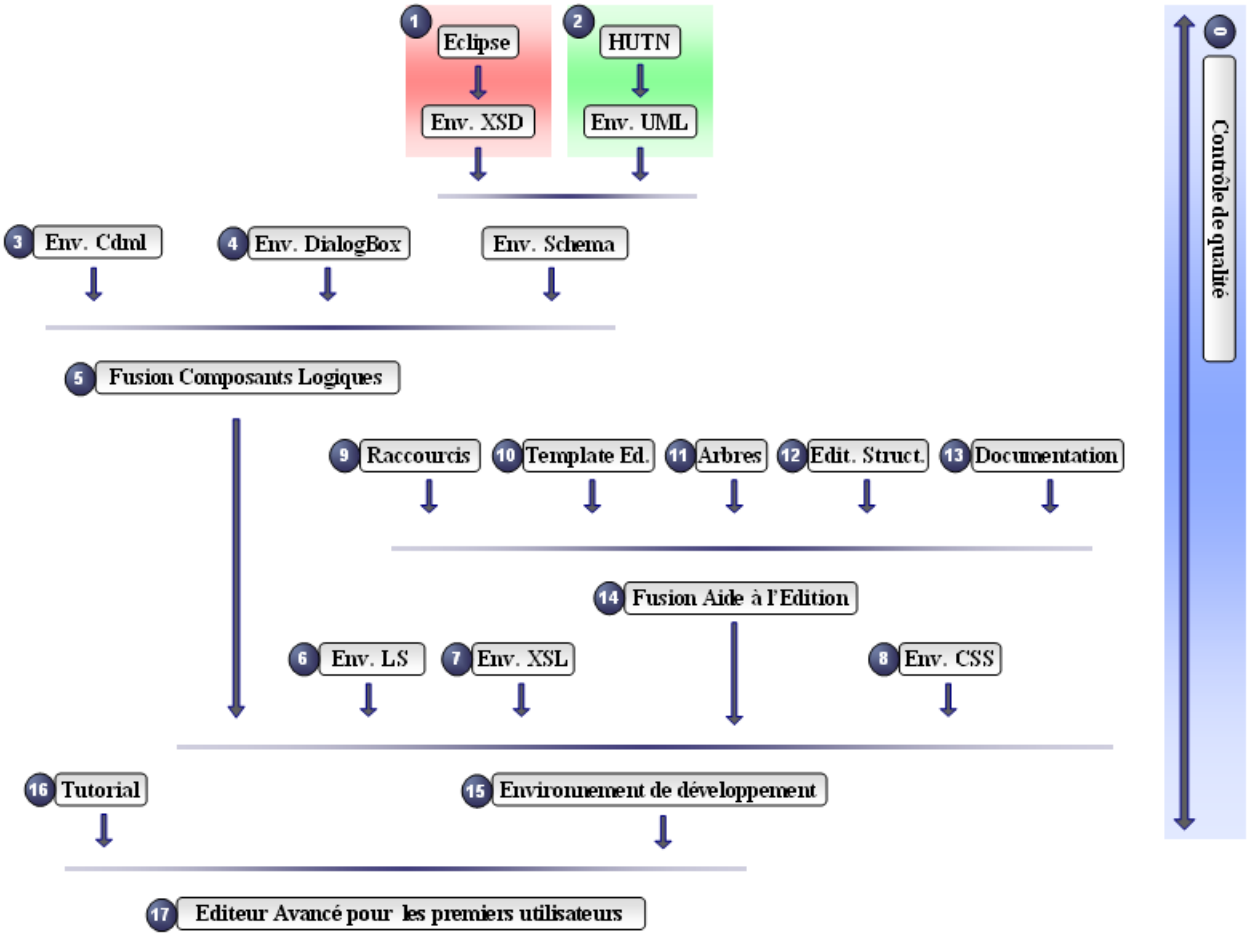

Fig. 3.1 – Organisation du projet

3.2 Organisation structurelle / Affectation des tâches

- **Contrôle de qualité (0) - Didier Parigot**

  Cette tâche se déroulait tout le long de notre projet. Monsieur Parigot contrôlait de manière régulière notre avancement, vérifiait que le résultat est correcte et mettait à jour sur le serveur CVS nos différentes modifications.

- **XML Schéma accéptés par Eclipse (1) - Benjamin Dahon**

  Cette étape à consisté à developper l'ensemble des transformations utiles afin de pouvoir aisément passer d'une sysntaxe abstraite (Absynt) vers un formalisme XML Schema, et plus précisément ceux générés par Eclipse lors de la création de modèles.

- **Autres formes d'XML Schéma (2) - Benjamin Dahon**

  Cette tâche est similaire à la précédente, cependant dans le cas présent les XML Schema aux quels nous nous sommes attachés sont différents de ceux générés par l'environnement Eclipse.

- **Environnement CDML (3) - Frédéric Ciminéra**

  Cette tâche regroupe plusieurs sous tâches différentes. Dans un premier temps nous avons créé des syntaxes abstraites pour ce composant qui permettent de le transformer en syntaxe simple, en html et en dialogbox. Dans un deuxieme temps nous avons développé un environnement d'édition qui permet de faciliter la conception de composants.

- **Environnement Dialogbox (4) - Daniel Leveneur**

  Cette tâche consiste à étudier le composant dialogbox existant et d'y apporter plusieurs améliorations afin de faciliter la prise en main de cet environnement par l'utilisateur.

- **Composants (5) - Tâche Commune**

  Dans cette étape, nous avons regroupé l'ensemble de nos composants modifiés et expliqué à chaque membre de l'équipe les modifications apportées. Cette étape est trés utile pour la réalisation de la tâche 15 c'est à dire la création de l'environnement de développement.

- **List Directories (6) - Tâche Commmunce**

  Cette partie a été développé car elle s'avérait utile et pratique. En effet ce composant (LS) permet de lister les répertoires dans SmartTools.Ainsi on peut naviguer dans notre arborescence et ouvrir le fichier voulu dans SmartTools.

- **XSL (7) - Daniel Leveneur**

  Cette tâche consite à créer un environnement XSL afin de pouvoir modifier, sauver un fichier XSL et de voir le résultat de l'application d'une feuille XSL sur notre document dans SmartTools.

- **CSS (8) - Daniel Leveneur**

  Cette tâche est identique à la précédente à la différence que cette fois ci, nous voulons modifer, sauver un fichier CSS et voir le résultat d'une feuille CSS sur notre document.

- **Raccourcis claviers(9) - Benjamin Dahon**

  Cette tâche a consisté à améliorer la navigation et l'édition en introduisant la possibilité d'incorporer des raccourcis claviers dans la définition des comportements des composants.

- **Template (10) - Frédéric Ciminéra**

  Cette tâche consiste à créer des templates (sous forme de dialogbox) qui permettent de rajouter des sous arbres prédéfinis dans différents composants pour simplifier leur édition.

- **Visualisation d'arbres (11) - Benjamin Dahon**

  Cette étape a consisté à redéfinir un composant capable de dessiner des arbres sous forme de diagramme hiérarchique.

- **Edition Structurée (12) - Frédéric Ciminera**

  Cette tâche permet d'éditer les arbres xml correspondants aux composants logiques ouverts dans l'environnement SmartTools. Cette édition propose plusieurs opérations de manipulation des noeuds de l'arbre du composant comme la copie, le collage, l'ajout après un élément etc.

- **Documentation et améliorations dialogbox (13) - Daniel Leveneur**

  Ajout d'une documentation au composant dialogbox afin de pouvoir expliquer à quoi correspond les actions des boites de dialogues.

- **Aide à l'édition (14) - Tâche Commune**

  Cette phase consiste à regrouper le résultat des différentes tâches 9, 10, 11, 12, 13 afin d'améliorer notre environnement de développement.

- **Environnement de développement (15) - Tâche Commune**

  Dans cette tâche nous avons du rassembler tout notre travail afin de créer un environnement de "prise en main" de l'outil SmartTools qui permet sans trop de diffilculté d'utiliser toutes ces possibilités.

- **Tutorial(16) - Tâche Commune**

  La tâche Tutorial a été crée afin de faciliter et aider l'utilisateur à créer son composant SmartTools. Nous avons donc crée des pages webs indiquant comment créer son composant et l'intégrer dans SmartTools.

- **Editeur avancé pour les premiers utilisateurs (17) - Tâche Commune**

  Le regroupement du tutorial et de l'environnement de développement dans SmartTools permet de fournir à l'utilisateur un point d'entrée pour construire et intégrer un composant SmartTools.

Section 4

Description du travail réalisé

4.1 XML Schéma acceptés par Eclipse (Tâche 1)

Jusqu'à présent, la seule forme d'entrée de description d'un composant était ABSYNT. La volonté de notre encadreur Didier PARIGOT était de pouvoir étendre ces formats d'entrée aux XML Schema. Nous avons isolé deux formes de XML Schema :

- les XML Schema acceptés par la plate-forme Eclipse
- les autres formes de XML Schema

L'avantage de pouvoir produire des schema acceptés par Eclipse est la possibilité de pouvoir utiliser l'ensemble des caractéristiques graphiques d' Eclipse. En effet, de par l'utilisation de EMF (Eclipse Modeling Framework), GEF (Graphical Editing Framework) et le plug-in Omondo UML, l'utilisateur peut définir sous forme graphique la syntaxe de son composant (sous forme de diagramme UML), avant de l'exporter sous forme de Schema XML.

Comme nous l'avons dit précédemment, un des avantages majeurs d'une telle démarche est de pouvoir réutiliser de nombreuses fonctionnalités d'Eclipse afin de les intégrer au processus de développement de composants.

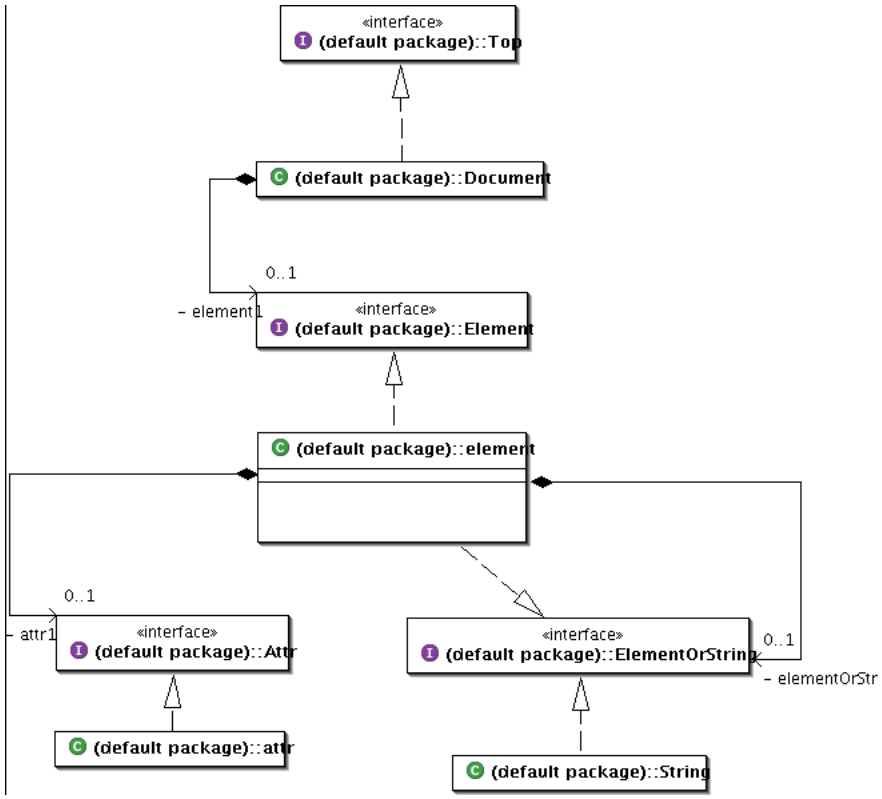

Fig. 4.1 – Création de modèles depuis Eclipse

Afin de parvenir à cette génération automatique de schema, une phase préliminaire d'étude de la génération de schema depuis Eclipse était nécessaire. De plus une grande part du temps imparti à cette tâche a été consommée dans la compréhension des différents outils entrant dans la chaîne de production de ces schema.

Ci-dessous, les mécanismes de SmartTools entrant en jeu lors des différentes transformations :

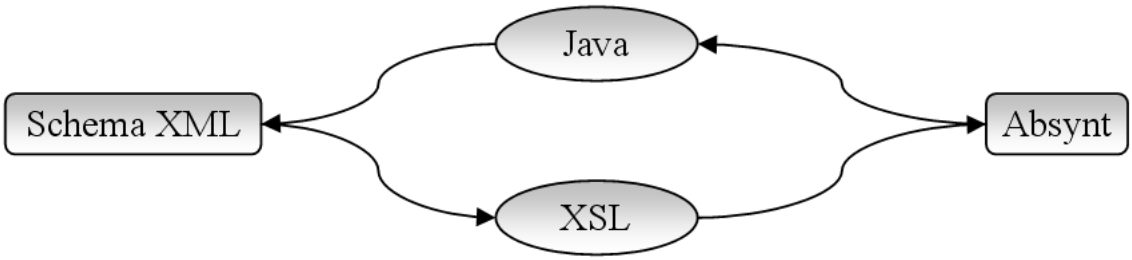

Fig. 4.2 – Mécanismes de transformation

et un exemple de fichier XML Schema généré :

```
<!-- Generating type Element Number of operators : 2-->
<xsd:complexType name="Element" abstract="true" />
```

```
<!-- Generating type Top Number of operators : 2-->
<xsd:complexType name="Top" abstract="true" />

...
 <!-- Generating operateur element -->
  <xsd:complexType name="element">
         <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="value" type="xsd:string"/>
            <xsd:element name="attrs" type="forum:Attr" minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="childs" type="forum:ElementOrString" minOccurs="0"
                         maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- Generating operateur document -->
 <xsd:complexType name="document">
         <xsd:sequence>
            <xsd:element name="root" type="forum:Element" minOccurs="1"
                         maxOccurs="1"/>
      </xsd:sequence>
   </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

4.2 Autres formes de XML Schema (Tâche 2)

Nous avons comme, il a déjà été signalé plus haut, convenu avec notre encadreur de permettre l'acceptation d'un autre formalisme de XML Schema. En effet, la plupart des gens écrivant des XML Schema, ne les écrivent pas de la même façon qu'Eclipse. Dans ce formalisme, une large utilisation des groupes est effectuée. En fait nous pouvons distinguer ces deux types de formalisme suivant la personne ou l'outil qui les génère. Les XML Schema vus précédemment sont plus communément écrits par des outils informatiques, alors que ceux-ci sont plus généralement générés par des personnes.

Dans ce cadre, nous avons été amené a travailler sur le formalisme UML d'EDF, proche de la recommandation OMG : HUTN. C'est donc dans cette optique, que nous avons développé les transformations bilatérales entre XML Schema et Absynt.

Afin de mieux comprendre le formalisme EDF, vous trouverez ci-dessous un exemple d'un tel fichier :

```
{
    Name "base pour tests"
    MaxIndex 74
}

1 baseDeRemarques {
    ISCLUSTER 1
```

```
    nom "Base pour test"
    etats 3(Etat),57(Etat),58(Etat)
    types 6(Type),7(Type),8(Type)
}

2 Etat {
    libelle "A documenter"
    definition "La fonctionnalite est correctement realisee mais elle a ete trouvee
par tests"
    remarques 23,36,37,26
}

3 Etat {
     libelle "Resolu par TNI"
     remarques 10
}
```

Avant notre arrivée, SmartTools permettait déjà l'export du formalisme ABSYNT vers XML Schema. Cependant cette transformation était incomplète (notamment ne permettait pas la gestion de l'héritage), mais surtout unilatérale. Nous avons donc été amené à remanier cette transformation afin de rendre compte de l'héritage entre types et opérateurs. Comme précédemment le mode opératoire est le suivant :

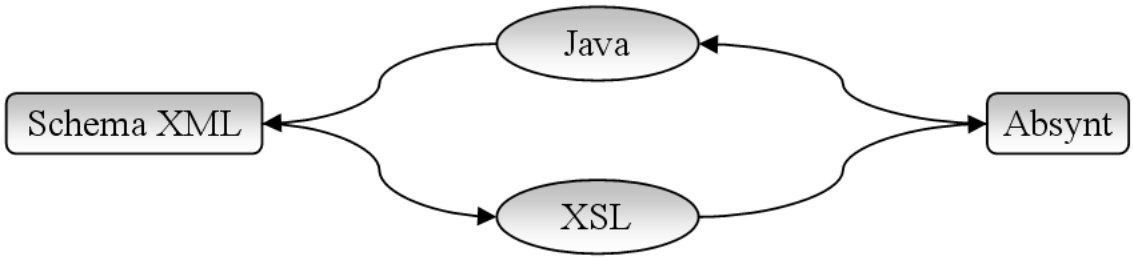

Fig. 4.3 – Mécanismes de transformation

Ci dessous un exemple de fichier XML Schema généré depuis SmartTools.

```
<!-- Group (type) definitions-->
 <xsd:group name="Element">
   <xsd:choice>
      <xsd:element ref="element"/>
   </xsd:choice>
</xsd:group>

<xsd:group name="Top">
   <xsd:choice>
      <xsd:element ref="document"/>
   </xsd:choice>
</xsd:group>

...
```

```
 <!-- Element definitions -->
<xsd:element name="Element" type="ElementType"/>

<xsd:element name="Top" type="TopType"/>

...

<!-- ComplexType definitions -->
 <xsd:complexType name="elementType">
   <xsd:complexContent>
         <xsd:sequence>
            <xsd:group ref="Attr" minOccurs="0" maxOccurs="0"/>
            <xsd:group ref="ElementOrString" minOccurs="0" maxOccurs="0"/>
         </xsd:sequence>
         <xsd:attribute name="name" type="java.lang.String" use="required"/>
         <xsd:attribute name="value" type="java.lang.String" use="required"/>
   </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="documentType">
   <xsd:complexContent>
         <xsd:sequence>
            <xsd:group ref="Element" minOccurs="0" maxOccurs="0"/>
         </xsd:sequence>
   </xsd:complexContent>
</xsd:complexType>
```

4.3 Cdml (Tâche 3)

Le langage Cdml (Composant description markup language) permet de décrire les différents containers qui font partie d'un composant. Cette description utilise le formalisme XML. Pour chacun de ces services on peut fournir un nom logique, la méthode à appeler et ses arguments ainsi que la façade de son interface (la façade d'un composant est l'unique point d'entrée pour un composant). Un composant (non abstrait) doit avoir entre autres dans sa description les éléments suivants :

- Son nom
- Le nom du container
- Le nom de sa classe façade
- Les noms et catégories de service avec leurs méthodes et arguments qui interagissent avec la façade.

Le composant Cdml de SmartTools permet de créer des composants. Ses services principaux sont les suivants : Create component...

C'est un composant important dans l'éditeur car c'est par lui que l'on va passer pour commencer la création d'un composant.

Pour ce composant nous avons défini par l'intermédiaire de fichiers cosynt des transformations qui permettent d'avoir différentes vues de ce composant :

Génération Cdml vers DialogBox qui permet de créer un fichier (cf. Partie environnement DialogBox) qui dans la vue appropriée permet de créer des boites de dialogues correspondant aux services

du composant. On pourra ainsi appeler ses services par l'intermédiaire d'une interface graphique sans avoir écrit une seule ligne de code.

Génération Cdml vers Behavior qui permet de créer un fichier Behavior qui va permettre de décrire le comportement du composant en fonction de ses services.

Génération Cdml vers HTML Cette transformation permet de visualiser un composant sous forme HTML, elle est utile pour avoir une vue plus compréhensible du composant et pour générer des documentations

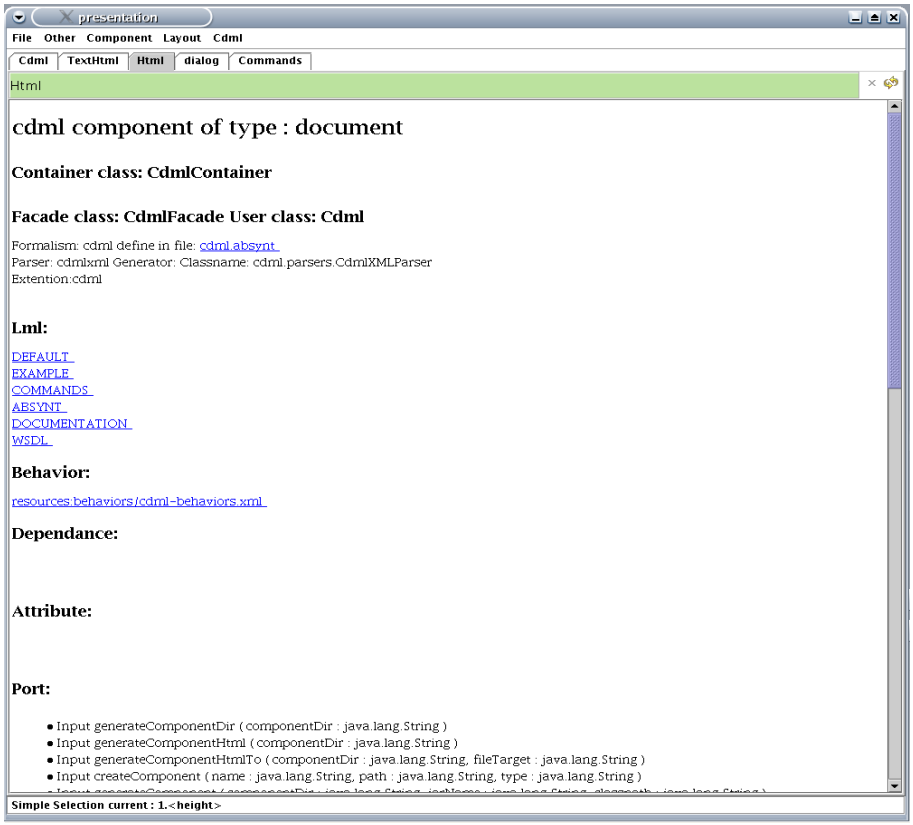

Fig. 4.4 – Représentation en HTML du composant CDML

Génération Cdml vers Syntaxe simple qui permet de générer une vue du composant non plus sous forme XML mais sous une forme simplifiée et plus lisible. Cette transformation est utile lors de l'édition d'un composant. Lorsque l'on modifie la vue simplifiée, les modifications sont répercutées sur le composant logique et donc rend son édition plus facile.

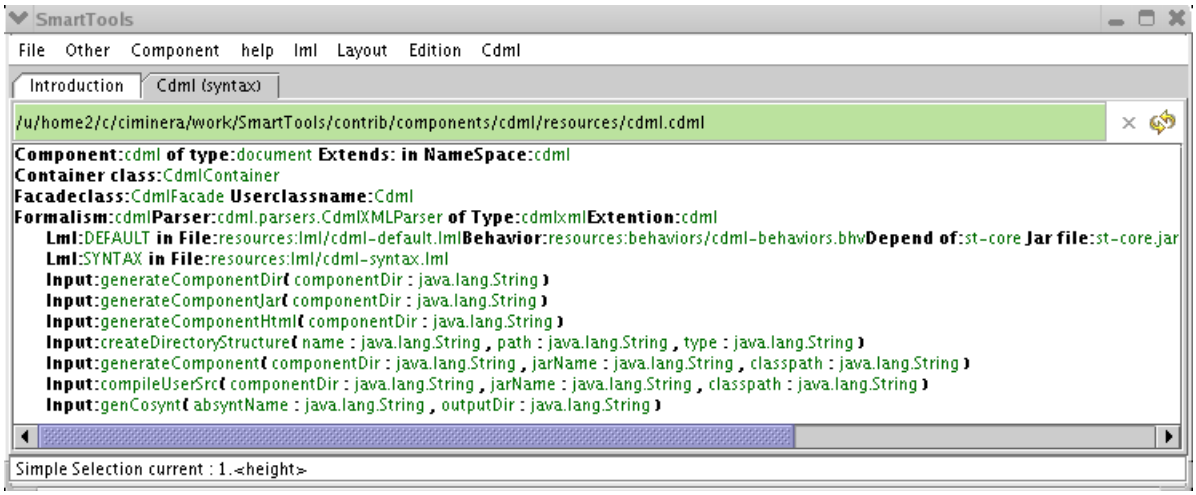

Fig. 4.5 – Représentation en syntaxe simple du composant CDML

4.4 Environnement dialogbox (Tâche 4)

L'environnement dialogbox est un composant qui est capable de créer des boites de dialogues grâce à un fichier d'entrée .dlb (fichier xml). Voici ci-dessous un exemple graphique d'une boite de dialogue :

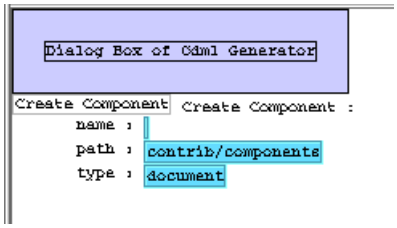

Fig. 4.6 – Exemple de boite de dialogue dans SmartTools

Voici le fichier dlb dont se servait le composant dialogbox de SmartTools pour produire le résultat de la figure ci dessus :

```
<?xml version="1.0" encoding="UTF-8"?>
 <buttons name="Dialog Box of Cdml Generator">
  <button name="Create Component">
  <action name="Create Component">
      <msg name="createComponent">
        <msgattr name="name" value=""/>
        <msgattr name="path" value=""/>
        <msgattr name="type" value=""/>
      </msg>
  </action>
  </button>
  <labels>
    <label name="name" value=""/>
    <label name="path" value="contrib/components/"/>
```

```
    <label name="type" value="document"/>
</labels>
</buttons>
```

SmartTools séparait l'affichage graphique avec la balise labels de la partie action (ce que va réellement faire le composant) avec la balise action.

Dans cette phase il fallait permettre à SmartTools de ne plus séparer ces 2 modes et de proposer ainsi à la fois une vue graphique et l'envoi d'évènements. Plus clairement SmartTools devait prendre en entrée ce type de fichiers et obtenir le même résultat que précedemment :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <actions name="DialogBox of Cdml Generator">
<action name="Create Component">
      <msg name="createComponent">
        <msgattr name="name" value="contrib/components/"/>
        <msgattr name="path" value=""/>
        <msgattr name="type" value="document"/>
      </msg>
</action>
</actions>
```

Les labels apparaissent ici implicitement, le fichier est ainsi plus simple et l'utilisateur peut creér sa boite de dialogue plus facilement.
Nous avons donc dû :

- Etablir un cosynt et absynt permettant de valider ce nouveau fichier dlb.
- Créer un fichier xsl permettant de gérer à la fois l'action des boutons et l'affichage graphique de la boite de dialogue.
- Modifier le compilateur en conséquence afin que le xsl géneré par cosynt fasse ce qu'il faut.

Afin de valider ce composant j'ai dû tester les fichiers .dlb que me fournissait Frédéric en entrée(cf Tache 3 - CDML).

4.5 Environnement Xsl(Tâche 7)

Le xsl devient souvent difficile à comprendre surtout lorsque la taille du fichier est importante.

Nous avons, dés notre intégration au projet, développé des feuilles xsl mais nous devions à chaque fois compiler, lancer SmartTools afin de voir le résultat.
Ce mécanisme nous faisait perdre pas mal de temps d'où l'intêret de cet environnement.

Grâce à cet environnement, nous pouvons ainsi voir en "temps réel" le résultat du xsl sur notre document.

Comment fonctionne cet environnement ?
Dés l'ouverture d'un fichier XSL, un arbre représentant le XSL est crée. Lorsque l'on veut voir le résultat de l'application du fichier XSL sur notre document, on clique sur le bouton *proceed* et le xsl pris en compte n'est plus le fichier XSL mais le contenu de la zone de texte.
On pourra par la suite sauver son fichier XSL dès que le résultat nous satisfait.
Voici ci-dessous un aperçu de cet environnement :

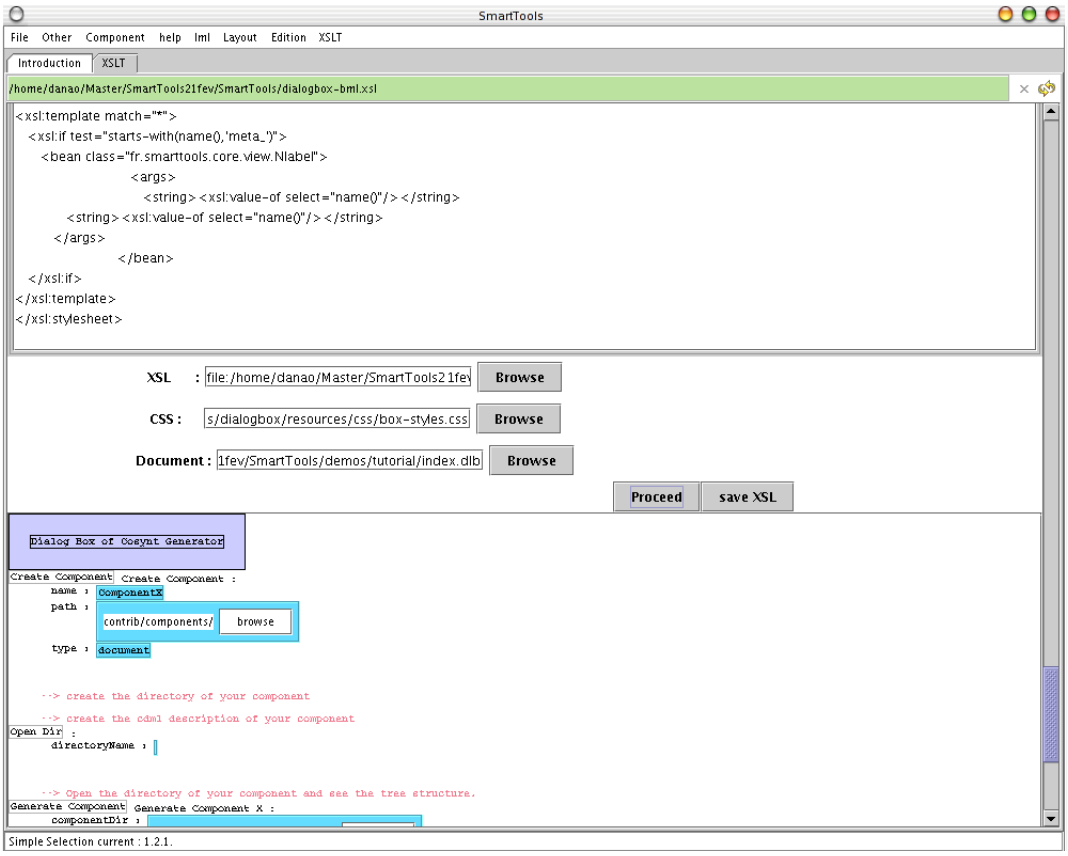

Fig. 4.7 – Exemple de l'environnement XSL

4.6 Environnement CSS (Tâche 8)

Cet environnement est similaire à l'environnement xsl mais cette fois ci nous modifions le css. Comme pour l'environnement XSL, dès l'ouverture d'un fichier CSS, un arbre représentant le CSS est crée.

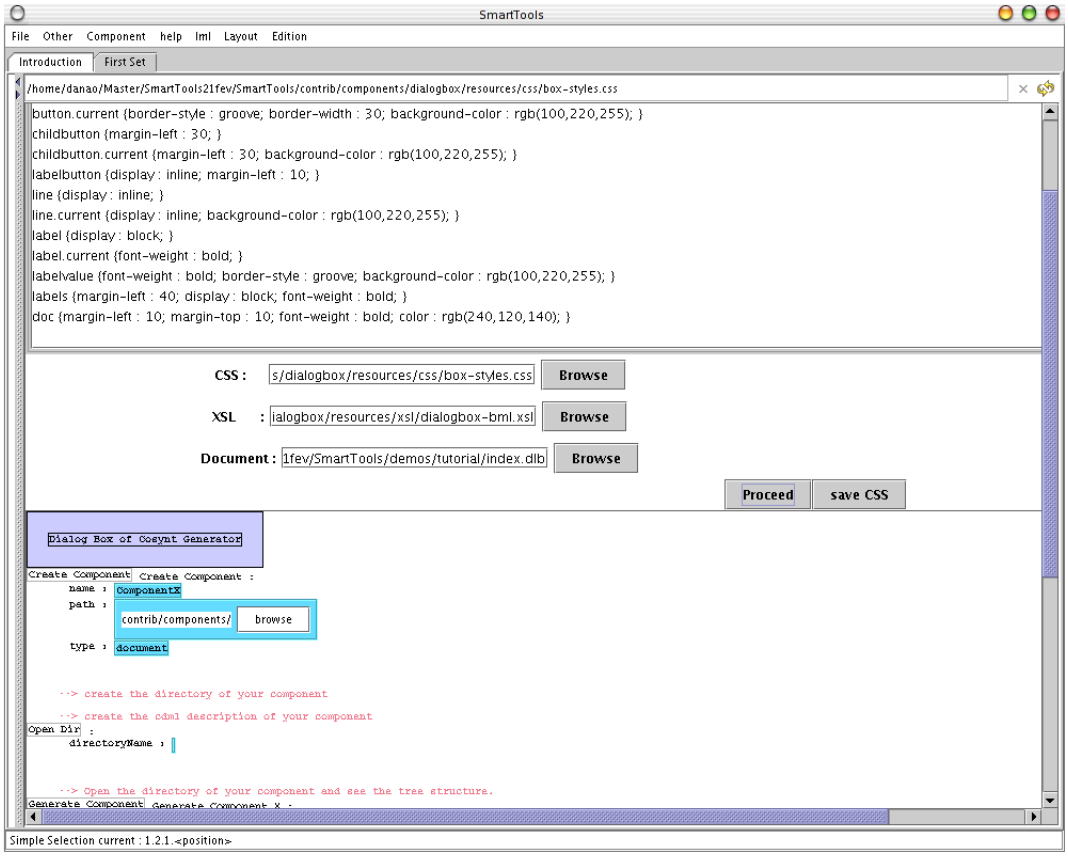

Fig. 4.8 – Exemple de l'environnement CSS

Comme dans la tâche précédente nous pouvons également sauver notre CSS lorque le résultat nous convient grâce au bouton *Save CSS*.

4.7 Raccourcis (Tâche 9)

Afin d'améliorer la navigation et l'édition il nous a été demandé de rendre possible la création de raccourcis claviers associées aux actions présentes dans les menus.

Notre travail s'est appuyé sur les points suivants :
- fichier behavior
- code Java
- fichier XSLT

Fichiers behavior
Chaque composant décrit un comportement (behavior) enregistré dans une instance de fichier XML. Une balise <keyb> était déjà présente dans ces fichiers mais inutilisée. Notre travail s'est donc dans un premier temps axé sur l'évolution de ces fichiers pour prendre en compte la présence de raccourcis claviers. Afin de mieux comprendre, vous trouverez ci-dessous, un exemple de fichier behavior.

```
<?xml version="1.0" encoding="ISO-8859-1"?> <behaviors>
```

```
<actions>
  ...
</actions>
<menus>
     <menu name="Edition">
       <item action="Copy"/>
         <item action="Cut"/>
    ...
     </menu>
</menus>
<toolbar> </toolbar>
<keyb>
  <shortcut menu="Edition" action="Copy" sequence="ctrl C" />
  <shortcut menu="Edition" action="Cut" sequence="ctrl X" />
</keyb>
<mouse> </mouse>

</behaviors>
```

Ainsi comme vous pouvez le voir sur l'exemple précédent, l'élément <keyb> possède un ensemble d'éléments <shortcut>. Cette structure peut être schématisée sous forme de diagramme UML de la façon suivante :

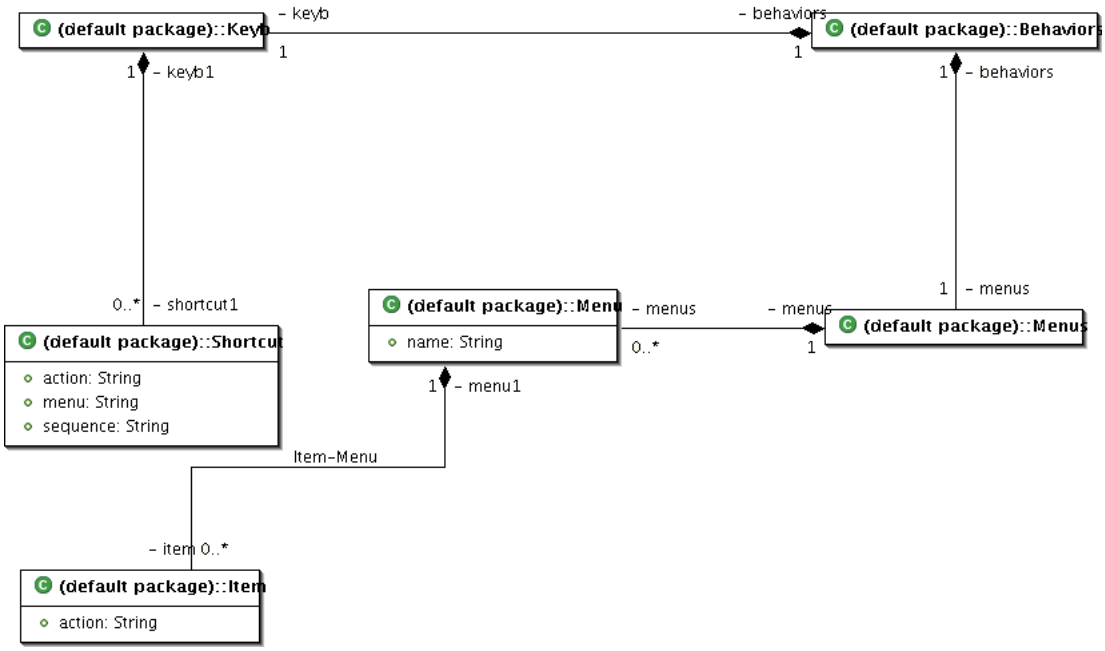

Fig. 4.9 – Diagramme représentant le behavior

Ainsi à chaque élément shortcut, un nom de menu, une action et une séquence sont associés. L'attribut action pointe vers l'item correspondant dans le menu dont le nom est défini par la valeur de l'attribut menu. Cette façon d'associer à chaque <shorcut> un <item>, permet de ne pas introduire d'ambiguïté et donc pas d'effets de bord lorsque deux raccourcis possèdent un item de même nom.

Code Java

L'écriture de code Java permettant de prendre en compte les changements liés aux raccourcis claviers, à induit une longue phase de compréhension de l'architecture des différentes classes actrices de l'affichage. Notamment, la compréhension de la gestion des évènements fut très longue.

Fichier XSLT

Enfin la dernière phase a consisté en l'écriture d'un fichier XSLT, prenant en compte les changements effectués dans les fichiers behavior. Afin de mieux comprendre en quoi la réécriture des fichiers XSLT était importante, nous allons vous présenter brièvement les différents processus qui entrent en jeu lors de l'affichage des vues d'un composant.

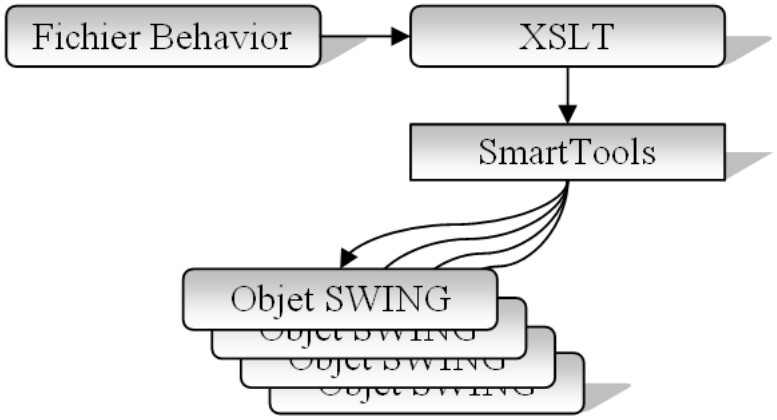

Fig. 4.10 – Création de l'interface graphique

Ci-dessous, le rèsultat final de notre travail concernant la gestion des raccourcis claviers :

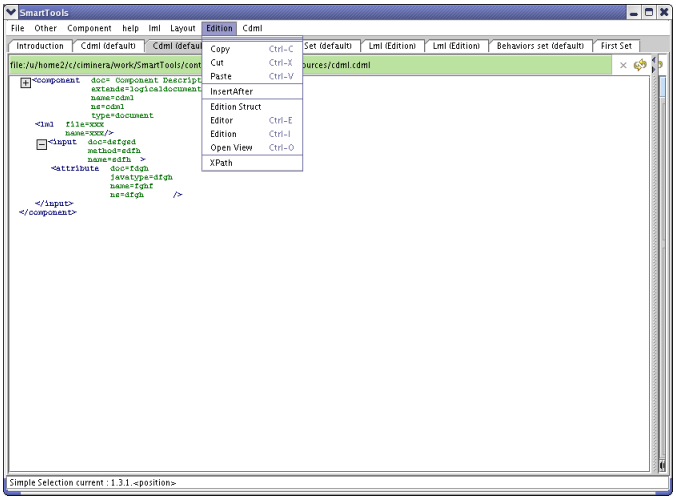

Fig. 4.11 – Affichage des raccourcis

4.8 Template (Tâche 10)

Nous avons aussi écris des fichiers dialogbox pour avoir une interface graphique permettant l'édition sous leur forme XML de certains composants. (Cdml, DialogBox, Behavior, World)

Ces interfaces permettent de rajouter par simple clic des templates respectant la syntaxe abstraite du langage dans le composant. Ces templates sont rajoutés sous forme de sous arbres xml et l'utilisateur n'a plus qu'à modifier les noms ou attributs qu'il désire dans ces templates. Ces templates sont insérés en utilisant les fonctionnalité de l'édition structurée(cf. tâche 12).

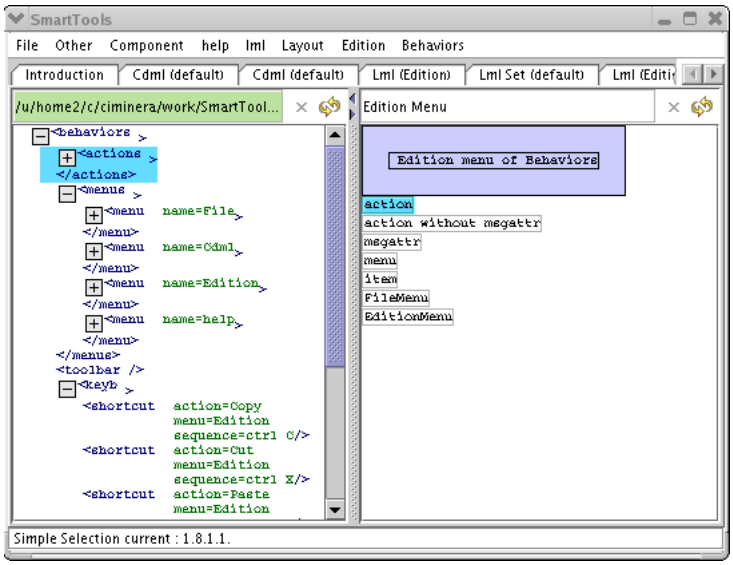

Fig. 4.12 – Edition du behavior

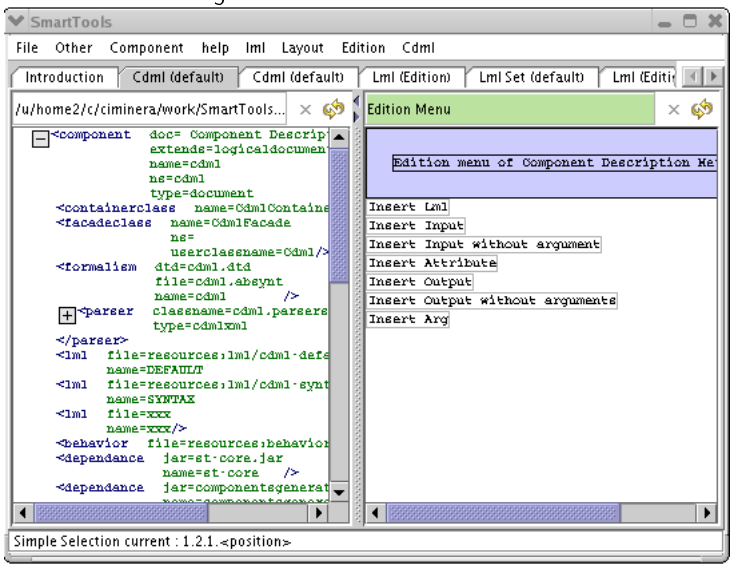

Fig. 4.13 – Edition du cdml

4.9 Visualisation d'arbres (Tâche 11)

Dans une version précédente de SmartTools, un composant permettait l'affichage de données sous forme d'arbre, cependant les évolutions successives de SmartTools ont rendu ce composant inopérant. Notre travail s'est donc inscrit dans le cadre de l'adaptation de cette entité pour la version actuelle de SmartTools. Cependant la complexité de ce composant à entraîné une réécriture complète suivant des spécifications moins coûteuses. A l'heure actuelle, la visualisation d'arbres, n'est pas encore totalement intégrée à la plate-forme SmartTools, néanmoins l'effort d'adaptation est mineur et ne nécessitera que très peu de temps.
Lors de l'intégration, ce composant fournira à l'interface graphique de SmartTools, la vue suivante :

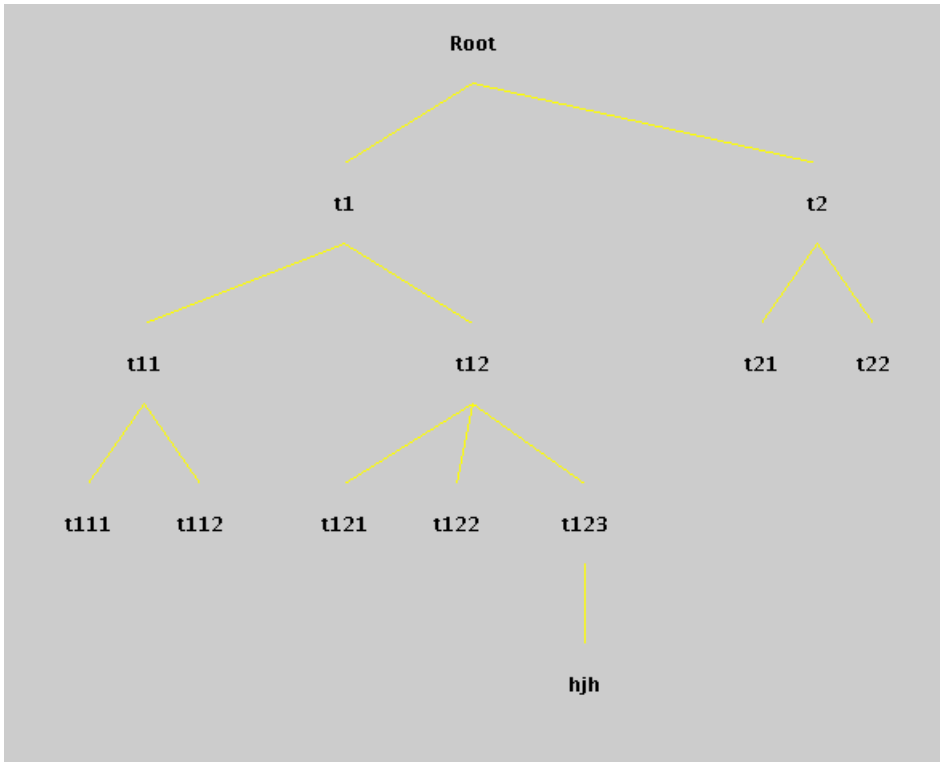

Fig. 4.14 – Visualisation d'arbres

4.10 Documentation et améliorations dialogbox (Tâche 13)

Une fois l'environnement dialogbox fonctionnel et testé, nous voulions faciliter l'utilisation de cette boite de dialogue.
Nous avons donc convenu d'une notation pertinante et établi un cosynt correspondant. Nous avons ensuite rajouter des tags documentation à notre langage défini dans la section 4.4. Nous y avons également apporté des améliorations diverses comme la création d'un AlabelTextChooser qui nous permet de naviguer afin de choisir le fichier souhaité. Notre fichier dlb ressemble à présent à :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <actions name="DialogBox of Cdml Generator">
<action name="Create Component">
```

```
      <msg name="createComponent">
        <msgattr name="name" value="contrib/components/"/>
        <msgattr name="path"  filechooser="fr.smarttools.core.util.chooser.OpenDirChoose
        <msgattr name="type" value="document"/>
      </msg>
</action>
 <doc>
   <but but ="--> create the directory of your component"/>
   <but but="--> create the cdml description of your component"/>
 </doc>
</actions>
```

et le résultat obtenu :

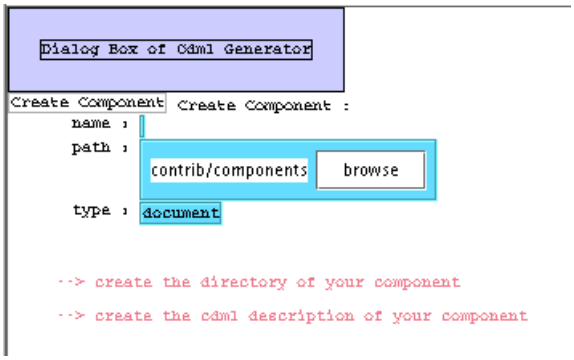

Fig. 4.15 – Nouvelle boite de dialogue

4.11 Environnement de développement (Tâche 15)

Cette tâche, qui était notre objectif final, a permis de réaliser un environnement d'aide au développement ainsi qu'une documentation sous forme HTML (cf Tutorial - Tache 16). Le problème principal pour cet éditeur a été de rendre SmartTools plus utilisable car les fonctionnalités étaient plus ou moins déja présentes et la plupart des composants dont nous avions besoin pour implémenter cet environnement de développement existaient déja dans SmartTools. Cependant ce qui nous pris le plus de temps a été de les connecter entres eux et de faire une interface plus pratique.

Nous disposions pour cela d'un avantage non négligeable : nous avons utilisé SmartTools pour enrichir SmartTools ce qui nous a permis de générer du code très rapidement et à chaque fois qu'un nouvel élément de l'environnement de développement a été créé nous pouvions aussitôt l'utiliser.

Pour créer un composant avec cet environnement de développement il faut d'abord créer la syntaxe abstraite du langage, on peut soit en créer une nouvelle soit la générer à partir d'une DTD, d'un XSD ou d'un schema UML existant (cf tâche 1 et 2). L'interface d'edition est générée automatiquement à partir du composant cdml : on génère à l'aide d'une syntaxe concrète un modèle dlb (cf tâche 3) qui sera ensuite interprété dans une vue graphique (cf tâche 4).

Ensuite il faut décrire le composant sous forme CDML qui doit respecter la syntaxe abstraite definie précédemment (cf tâche 3). C'est à ce moment là que l'on peut décrire les attributs du composant. L'éditeur propose une édition structurée du composant (cf tâche 12) et propose des patrons de conception XML correspondant à diférentes parties du composant en insérant des sous arbres XML (cf tâche 10). Une fois la description du composant terminée on peut générer le fichier jar correspondant.

Une fois le composant généré on a la possibilité en utilisant l'environnement LS (cf tâche 6) de visualiser tous les fichiers décrivant le composant.

On peut par la suite avec cet environnement de développement créer des syntaxes concrètes (cosynt). Ces syntaxes permettent de transformer le composant dans différents modèles. SmartTools générera ensuite automatiquement un parseur pour la syntaxe ainsi qu'un fichier XSL pour effectuer la transformation sous forme graphique (BML) ou textuelle.

On peut aussi créer des fichiers de description (LML) qui décrivent les vues utilisables pour un composant. Pour qu'elles soient accessibles il faut qu'elles apparaissent dans la description CDML du composant.

L'environnement permet aussi de créer des fichiers Behaviors qui décrivent le comportement du composant. Ils peuvent aussi être générés directement à partir de la description CDML.

Bien que l'environnement génère automatiquement des fichiers XSL pour une syntaxe concrète il dispose aussi de deux éditeurs, un pour éditer des fichiers XSL (cf tâche 7) et un pour les fichiers CSS (cf tâche 8). Ces deux éditeurs permettent de voir en temps réel le résultat des modifications.

Section 5

Conclusion Globale

Le but de notre projet était de développer une application d'aide aux premiers utilisateurs pour leur permettre de créer des composants avec l'outil SmartTools.

Malgré le temps qui nous a été imparti, l'objectif a tout de même été atteint. Cependant il faut savoir que SmartTools est un projet complexe et qu'il est difficile d'en appréhender tous les aspects.
En effet le principal problème que l'on a rencontré a été de comprendre le principe de fonctionnement de SmartTools mais une fois cela compris, le développement avec cet outil nous a fait permis de gagner du temps sur l'implémentation car la plupart du code a été généré automatiquement.

En définitive ce projet peut-être considéré comme très enrichissant pour tous les membres de l'équipe, nous y avons découvert des concepts nouveaux comme la programmation générative, la programmation par aspect et l'approche MDA.

En trois mois il nous était impossible d'aborder tous ces aspects, nous en avons simplement eu un aperçu et c'est pourquoi nous avons travaillé chacun sur des petites parties différentes de SmartTools afin de pouvoir, à terme, réaliser notre objectif.

Section 6

Annexes

DTD de CDML

```
<!ENTITY % FORMALISM 'formalism'>
<!ENTITY % PARSER 'parser'>
<!ENTITY % CONTAINER 'containerclass'>
<!ENTITY % FACADE 'facadeclass'>
<!ENTITY % DOCUMENTATION 'documentation'>
<!ENTITY % ATTRIBUTE 'attribute'>
<!ENTITY % INPUT 'input'>
<!ENTITY % OUTPUT 'output'>
<!ENTITY % INOUT 'inout'>
<!ENTITY % ARG 'arg'>
<!ENTITY % BINDING 'binding'>
<!ENTITY % LML 'lml'>
<!ENTITY % DEPENDANCE 'dependance'>
<!--                                -->
<!ELEMENT component ((%FORMALISM;)?, (%PARSER;)*, (%CONTAINER;)?, (%FACADE;)?,
(%LML;)*, (%DEPENDANCE;)*, (%DOCUMENTATION;)?, (%ATTRIBUTE;)*, (%INPUT;)*,
(%OUTPUT;)*, (%INOUT;)*)>
<!ATTLIST component
    name CDATA #REQUIRED
    type CDATA #REQUIRED
    extends CDATA #REQUIRED
    ns CDATA #REQUIRED
>
<!ELEMENT formalism EMPTY> <!ATTLIST formalism
    name CDATA #REQUIRED
    dtd CDATA #IMPLIED
>
<!ELEMENT parser EMPTY> <!ATTLIST parser
    type CDATA #REQUIRED
    extention CDATA #REQUIRED
    classname CDATA #REQUIRED
>
<!ELEMENT containerclass EMPTY> <!ATTLIST containerclass
    name CDATA #REQUIRED
```

```
>
<!ELEMENT facadeclass EMPTY> <!ATTLIST facadeclass
    name CDATA #REQUIRED
>
<!ELEMENT lml EMPTY> <!ATTLIST lml
    name CDATA #REQUIRED
    file CDATA #REQUIRED
>
<!ELEMENT dependance EMPTY> <!ATTLIST dependance
    name CDATA #REQUIRED
    jar CDATA #REQUIRED
    url CDATA #REQUIRED
>
<!ELEMENT documentation (#PCDATA)> <!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    doc CDATA #IMPLIED
    javatype CDATA #IMPLIED
    name CDATA #IMPLIED
    ref CDATA #IMPLIED
    ns CDATA #IMPLIED
>
<!ELEMENT input ((%ATTRIBUTE;)*, (%BINDING;)*)>
<!ATTLIST input
    doc CDATA #IMPLIED
    method CDATA #REQUIRED
    name CDATA #REQUIRED
    ns CDATA #IMPLIED
>
<!ELEMENT output ((%ARG;)*)>
<!ATTLIST output
    doc CDATA #IMPLIED
    method CDATA #IMPLIED
    name CDATA #REQUIRED
    ns CDATA #IMPLIED
>
<!ELEMENT inout ((%ARG;)*)>
<!ATTLIST inout
    doc CDATA #IMPLIED
    method CDATA #REQUIRED
    name CDATA #REQUIRED
    output CDATA #REQUIRED
    ns CDATA #IMPLIED
>
<!ELEMENT arg EMPTY> <!ATTLIST arg
    doc CDATA #IMPLIED
    name CDATA #IMPLIED
    type CDATA #IMPLIED
    javatype CDATA #IMPLIED
    ref CDATA #IMPLIED
>
<!ELEMENT binding ((%ARG;)*)>
```

```
<!ATTLIST binding
    toMethod CDATA #REQUIRED
>
```

BNF de Cosynt

```
COSYNT        - "Cosynt for" NAME "is" CONCRETE LAYOUT CONCRETRE
- "Concrete Syntaxe {" BNF PARSER LEXER "} BNF           - "BNF {"
BNF_RULE+? "}"
                              ("{" REF_ATTR ("," REF_ATTR)* "}")? ":" BNF_TOKEN+ ";"
BNF_RULE      - NAME "(" (NAME ("," NAME)*)? ")" BNF_TOKEN     -
SUGAR | REF_ATTR | REF_ARG | REF_LEXER | ITER | OPTIONAL | CASE
REF_ATTR      - "@" NAME REF_ARG       - "#" NAME
REF_LEXER     - "%" NAME
SUGAR         - """ NAME """ ITER          - ("+" | "*") "["
SUGAR* REF_ARG SUGAR*
                ("separator" SUGAR+)?
                      ("beforeList" SUGAR+)?
                          ("afterList" SUGAR+)?
                              "]"
OPTIONAL      - "{" SUGAR* REF_ARG SUGAR*
                ("beforeList" SUGAR+)?
                ("afterList" SUGAR+)?
                      "}"
CASE          - "case (" REF_ATTR ")" ":" "(" (VALUE ":"
BNF_TOKEN+ ";")+ ")" PARSER        - "Parser[k = " NAME "{"
PARSER_RULE* "}" PARSER_RULE   - LEXER         - "Lexer[k = " NAME
", attributes = " NAME "] {" LEXER_RULE+ "}" LEXER_RULE    - NAME
" = <" REGEXPR ">;" LAYOUT        - "Layout {" STYLES TRANSFO
OUTPUT "}" STYLES        - "Styles[default = " NAME ", sugars = "
NAME
                 ", attributes = " NAME
                            "] {" STYLE_RULE* "}";
STYLE_RULE    - TRANSFO       - "Transformation Rules {"
TRANSFO_RULE* "}" TRANSFO_RULE  - NAME ":" TRANSFO_TOKEN+
TRANSFO_TOKEN - (NAME ":")? (REF_CHILD | "(" TRANSFO_TOKEN+ ")")
OUTPUT        - "Output {" OUTPUT_RULE+ "}" OUTPUT_RULE   - NAME
"[default = " NAME ", sugars = " NAME ",
                attributes = " NAME "] {" (NAME ":" REF_OP+)* "}"
```

Références

- La page officielle de SmartTools
  http://www-sop.inria.fr/smartool/SmartTools/

- Didier PARIGOT, Carine COURBIS, Pascal DEGENNE, Alexandre FAU, Claude PASQUIER
  L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools.
  L'Objet- numéro spécial XML et les objets, Juin 2002.

- Carine COURBIS, Pascal DEGENNE, Alexandre FAU, Didier PARIGOT
  Un modèle abstrait de composants adaptables. Systèmes à composants adaptables et extensibles, Juin 2003.

- Composantes SmartTools comme des Web Services
  http://www.essi.fr/~kowalski/contexte.html

- Carine COURBIS.
  Contribution à la prgrammation générative, Thèse, Université de Nice Sophia-Antipolis, Décembre 2002.

- Didier Parigot, Carine Courbis.
  Domain-Driven Development : the SmartTools Software Factory.

- Contribution à la programmation générative.
  http://www-sop.inria.fr/smartool/Didier.Parigot/

- Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.
  http://www.redbooks.ibm.com/