Transformation de composants SmartTools en plugins Eclipse

Encadrant :
PARIGOT Didier

Etudiantes :
DUCHAMP Agnès
FAROUK HASSAM Shouhéla
MEVEL Stéphanie

EPU, Mars 2006

Remerciements

Nous tenons à remercier PARIGOT Didier, chercheur à l'INRIA, pour ses explications et sa disponibilité pendant toute la durée du projet.

Sommaire

Lexique

- **Absynt :** *Langage déclaratif utilisé pour définir une syntaxe abstraite d'un langage*
- **AST :** *Abstract Syntax Tree*
- **Cdml :** L*angage décrivant un composant*
- **Cosynt :** *Spécification décrivant la syntaxe concrète d'une grammaire et son affichage*
- **CVS :** *Concurrent Versions System*
- **DOM :** *Document Object Model*
- **DSL :** *Domain Specific Language, petits languages*
- **IDE :** *Integrated Development Environment*
- **Lml :** *Langage basé sur XML décrivant la structure de l'interface graphique*
- **PDE :** *Plugin Development Environment*
- **OSGI :** *Open Services Gateway Initiative*
- **RCP :** *Rich Client Plat-Form*
- **XML :** *eXtensible Markup Language*
- **XSL :** *eXtensible Stylesheet Language*
- **Xprofile :** *design pattern visiteur*

Notre projet « Transformation des composants SmartTools en plugins Eclipse s'est déroulé à l' INRIA sous la direction de Monsieur Didier Parigot.

L'équipe SmartTools de l'INRIA a développé un générateur d'IDE (*Integrated Development Environment*), SmartTools, basé sur une approche par fabrique logicielle.

Tout au long de ce projet, nous avons effectué une étude préliminaire consistant à proposer un mode de développement par fabrique logicielle pour développer des plugins sous Eclipse.
En effet, Eclipse est un IDE composé de plugins pouvant donc supporter ce mode de développement.

Nous allons dans un premier temps vous présenter plus en détails notre sujet.
Par la suite, nous vous décrirons notre démarche en spécifiant pour chaque étape notre motivation.
Finalement, une annexe technique décrira concrètement nos réalisations.

I. Présentation du sujet :

I.1. Contexte du projet

Avec l'essor grandissant de la société de l'information, il s'avère nécessaire de repenser fondamentalement le processus de développement logiciel. Depuis quelques années, des nouveaux concepts comme la programmation par composants ou la notion d'architecture dirigée par les services (SOA), la programmation par séparation des préoccupations, la programmation dirigée par des modèles (MDA) ont été proposées pour répondre à ces nouveaux défis. Cela s'est accompagné de l'émergence d'une multitude de technologies logicielles, proposées et soutenues par des consortiums internationaux de standardisation (W3C, OMG...) ou des fondations/alliances (Apache, Eclipse, OSGA ...) ou des grands groupes industriels (MicroSoft, IBM, SUN, BEA...).

Ces différents concepts peuvent être unifiés et regroupés dans une notion nouvelle de fabrique logicielle, concrétisée par le logiciel SmartTools crée par l'équipe SmartTools de L'INRIA.

Ce concept de fabrique logicielle est défendu depuis 2004 par les plus grands groupes informatiques du domaine, comme étant l'un des axes importants pour la recherche et le développement pour le génie logiciel. On peut citer en particulier Microsoft avec leur nouvelle version de leur environnement de programmation avec les notions de DSLs (« Domain-Specific Language »), IBM avec l'environnement de programmation Eclipse et ses évolutions vers la notion de « Rich Client Plat-Form » (RCP) et enfin SUN avec l'environnement NetBeans.

I.2. Objectifs

Le but de notre projet est de comprendre comment transformer les composant SmartTools en des plugins Eclipse. Cette étude permettra d'offrir l'approche de fabrique logicielle dans l'environnement Eclipse. Ainsi les développeurs de plugins éclipse pourront générer 90% de leur code à l'aide de la fabrique logicielle SmartTools.

I.3. Description rapide de SmartTools

À l'heure actuelle, SmartTools **est composé d'une dizaine de DSLs** (petits langages) **avec leurs outils associés.** Ceci représente environ 100 000 lignes de code java écrites pour un logiciel comportant finalement environ 1 000 000 de lignes de code après génération automatique de code (e.g., structures de données, parseurs, vues graphiques, conteneurs de composants, etc.). Cette génération de code se fait par les différents générateurs de composants de SmartTools lui-même.

SmartTools a la capacité de s'auto générer. En effet, le résultat immédiat de cette auto-utilisation est que plus de 90% du code source des applications ou des outils réalisés est produit automatiquement par la fabrique.

Les trois concepts de base de SmartTools sont les suivants :

- un développement dirigé par la spécification de langages dédiés (modélisation à l'aide de DSLs) ;
- une notion de séparation des préoccupations définies directement sur ces langages dédiés, associés au métier sous-jacent
- une architecture dirigée par les services basée sur des composants qui doivent pouvoir étendre leur interface (services) dynamiquement.

I.4. Outils utilisés

Le projet a été réalisé avec Eclipse. Nous avons utilisé les technologies JAVA et XML. Les composants de SmartTools sont sous une base CVS. Nous avons donc disposé d'un serveur CVS afin d'effectuer un travail collaboratif efficace.

II. Organisation du projet :

II.1. Directions possibles pour le projet

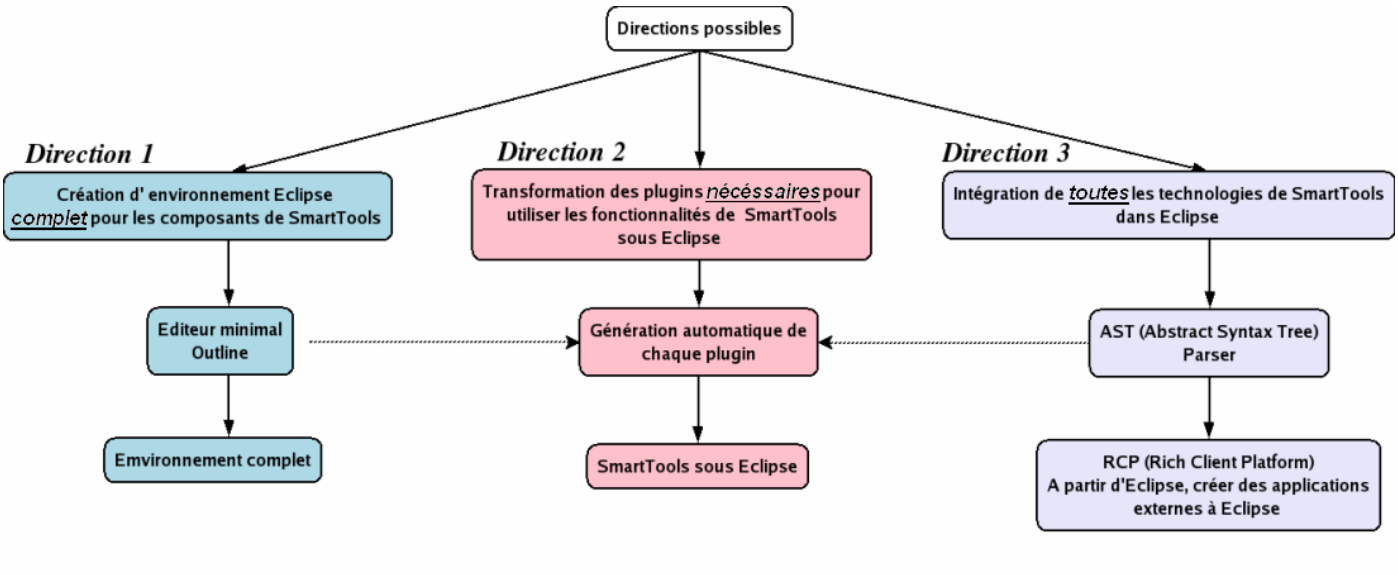


Figure 1 : Les directions possibles

Il y a trois directions possibles pour aborder le projet. Faute de temps, il était impossible de traiter toutes les directions. Nous avons choisi de nous focaliser sur la seconde direction et de faire un avancement minimal sur les deux autres directions.

La première direction correspond à la création d'environnement Eclipse pour chaque composant de SmartTools.
Cela consiste en la création d'éditeur complet pour chaque DSL de SmartTools (vue graphique, coloration, complétion, mise à jour automatique de la vue graphique, …).

Cette direction demande beaucoup de temps de travail, une programmation lourde et des connaissances avancées d'Eclipse.
De plus, il était d'abord nécessaire d'intégrer les fonctionnalités de ces DSL avant d'approfondir leur environnement

La seconde direction, sur laquelle nous nous sommes focalisées, correspond à la création de tous les plugins nécessaires pour offrir les fonctionnalités de SmartTools. Cela permettra la création automatique de plugins.

Enfin, la dernière direction consiste en l'intégration des toutes les technologies de SmartTools dans Eclipse. Nous n'avons pas choisi cette direction car il reste encore quelques questions techniques non résolues. De plus, cette direction demande de bien maîtriser le fonctionnement de SmartTools et d'Eclipse.

Il nous était nécessaire d'avancer un minimum dans les directions 1 et 3 pour une intégration minimale des composants de la direction 2.
Notre but est donc de transformer les composants SmartTools en plugins Eclipse.
Nous allons dans la prochaine partie vous expliquer la démarche que nous avons suivie.

II.2. Organisation structurelle du projet

Voici une vue d'ensemble de notre démarche :

Dans cette vue d'ensemble nous avons gardé les mêmes couleurs que precédémment afin de bien repérer les différentes directions pour chacune des étapes. Ce schéma illustre la démarche que nous avons choisie. Nous avons donc surtout travaillé dans la seconde direction en avançant au minimum dans les directions 1 et 3.

Dans les deux premières étapes, nous avons transformé manuellement les composants SmartTools en plugins Eclipse. L'étape 3 correspond à une longue réflexion sur l'automatisation de notre démarche, sur le travail en groupe et également les prochaines étapes a suivre.

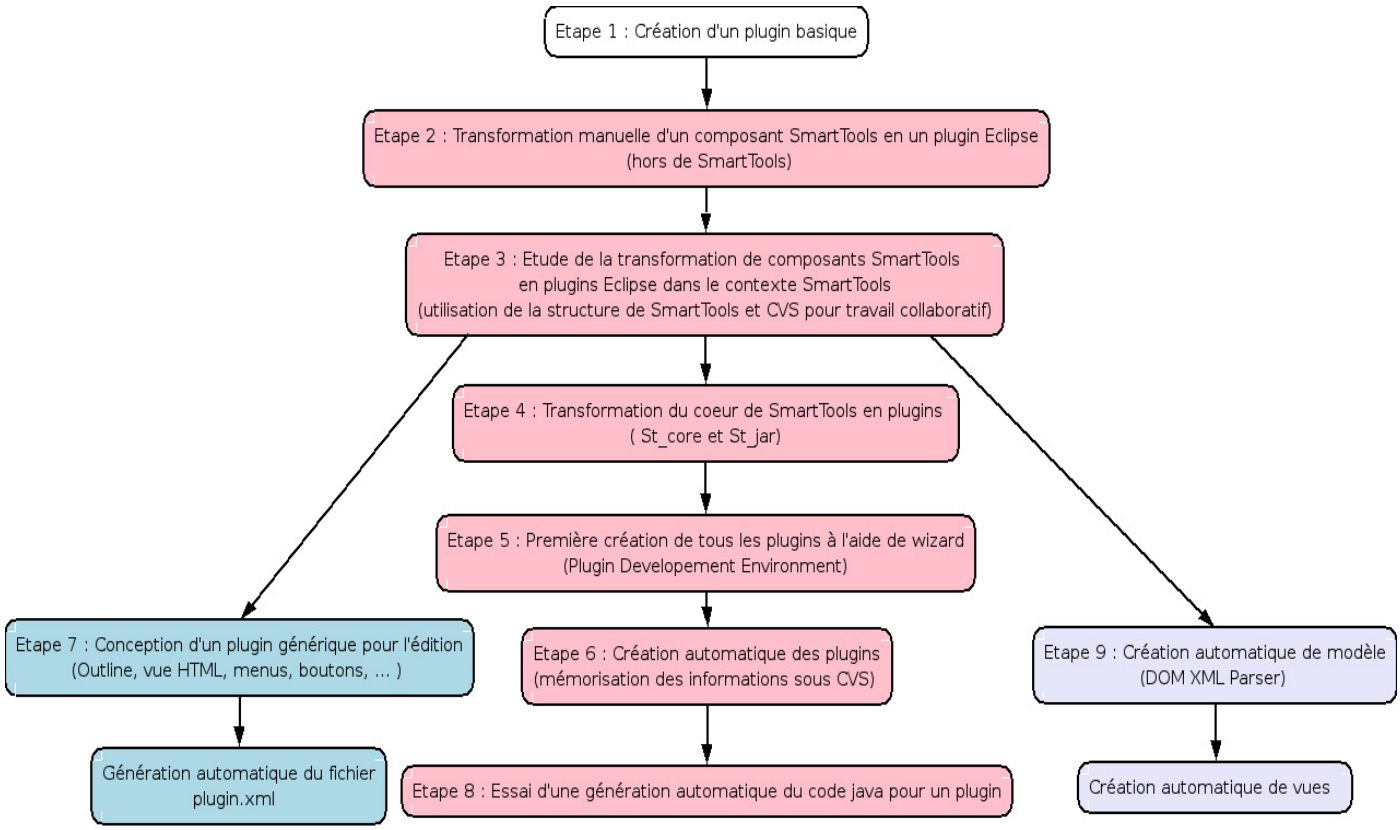


Figure 2 : Les étapes

Répartition des taches :

Nous avons travaillé ensemble pour la seconde direction.
Stéphanie MEVEL s'est ensuite concentrée sur la direction 1, Agnes DUCHAMP sur la direction 3 et Shouhéla FAROUK HASSAM aux tests et à la validation des différentes étapes.

Pour chaque étape, nous allons expliciter ci-dessous nos motivations ainsi que la démarche que nous avons suivie. Vous trouverez en annexe les parties techniques. La partie technique est sous forme HTML pour que les futurs utilisateurs puissent l'avoir comme documentation sous Eclipse.

Etape 1 : Création d'un plugin basique

Motivation :

Cette étape nous permet de créer un premier plugin basique afin de comprendre de quoi il est constitué (les fichiers de base). L'intérêt est ici de se familiariser avec les plugins Eclipse.

Démarche :

A l'aide de tutoriaux, nous avons créé un plugin simple (***HelloWorld***), nous avons étudié comment utiliser des wizards.
Une fois le plugin créé nous avons étudié sa structure.
Il est composé de :

- *classe Xplugin.java :* classe permettant de gérer le cycle de vie du plugin.
- *fichier plugin.xml :* décrit comment le plugin étend la plateforme, quelles extensions il utilise et comment il implémente ses fonctionnalités
- *fichier .classpath :* chemins nécessaires et droits d'accès
- *fichier .project :* contient les données relatives au projet
- *fichier MANIFEST.MF :* contient toutes les dépendances du plugin
- *fichier build.properties :* description des répertoires du projet contenant les fichiers nécessaires à la compilation, éxécution ...

Etape 2 : Transformation manuelle d'un composant SmartTools en plugin Eclipse par copie de fichiers (indépendants de SmartTools)

Motivation :

SmartTools est composé de plusieurs petits langages sous forme de composants (notion proche de plugin).
Nous avons donc crée un plugin avec un éditeur simple pour un premier exemple de langage. Nous avons pour cela choisi un langage simple : *LML*.
Ce plugin est créé manuellement (par copie des fichiers générés par SmartTools) et est créé hors de SmartTools (sans prendre en compte la hierarchie CVS) a l'aide du PDE.

Démarche :

Nous avons créé un éditeur associé à un langage. Nous avons choisi *LML* comme langage car c'est un langage simple.
Nous nous sommes basées sur un tutorial pour appliquer la coloration syntaxique à notre langage.
Enfin, nous avons étudié le mécanisme de la complétion des langages basés sur une syntaxe xml.

Etape 3 : Etude de la transformation de composants SmartTools en plugins Eclipse dans le contexte SmartTools

Motivation :

Nous avons appris dans les étapes précédentes comment transformer manuellement les composants SmartTools en plugins Eclipse.

Nous cherchons dans cette étape à automatiser notre démarche mais cette fois , dans le contexte de SmartTools. Il a fallu réfléchir à la manière de créer des plugins Eclipse sous la hierarchie CVS de SmartTools. Nous avons essayé de réduire le plus possible le nombre de classes (factorisation).

De plus, nous voulions faire un travail collaboratif. C'est la raison pour laquelle nous avons travaillé sous l'architecture sous CVS. Cela permet à chacun de récupérer le travail effectué par d'autres sans avoir à recréer les plugins.

Démarche :

Dans un premier temps, nous avons cherché comment intégrer le code spécifique d'Eclipse (éditeurs) dans la hiérarchie de SmartTools. En effet nous avons voulu qu'il soit toujours possible de faire la génération des composants hors du contexte d'Eclipse.

Plus précisément, les packages d'Eclipse sont inconnus lors de cette génération des composants. Ainsi le code spécifique à Eclipse (les éditeurs) devait être placé hors de la hiérarchie de SmartTools. Pour cela, nous avons placé les éditeurs dans un répertoire externe à SmartTools et créé un lien sous SmartTools pour y accéder.

Nous avons ensuite mémorisé sous la base CVS toutes les informations nécessaires aux plugins (***.project, plugin.xml***, ...) pour ensuite automatiser notre démarche (sans forcément passer à chaque fois par le PDE)

Enfin, nous avons procédé à plusieurs tests pour vérifier le bon fonctionnement de nos différents plugins. Pour cela, à plusieurs reprises nous avons repris de zero et réappliqué notre démarche étape par étape pour valider notre démarche et s'assurer que nos plugins fonctionnent. Cette étape nous a sensibilisée à la difficulté du travail en groupe.

Etape 4 : Tranformatiom du noyau de SmartTools en deux plugins Eclipse : St_core et St_jar

Motivation :

SmartTools utilise un ensemble de librairies (accessible sous forme de jar) et un cœur. Nous avons donc créé des plugins de bases (***st-jar*** et ***st-core***) qui regroupent cet ensemble de librairies. Par la suite quasiment tous les plugins que nous allons créer dépendront de ces plugins (voir l'onglet dépendance de l'environnement ***plugin.xml***).

Démarche :

Nous avons crée le plugin ***st-jar*** à partir de fichiers jar existants et ***st-core*** à partir des classes du noyau de SmartTools**.**

Etape 5 : Première création de tous les plugins à l'aide de wizards (Plugin Developpement Environment)

Motivation :

Pour la première création d'un plugin, il fallait utiliser l'environnement PDE (Plugin Developpement Environment), c'était le seul moyen.
Nous disposons alors de plugins créé hors de SmartTools (cf. Etape 2) , l'objectif de cette étape est de les créer dans le contexte Smarttools (prise en compte de la hiérarchie des fichiers).
Cette étape a pour objectif de créer sur le même modèle tous les plugins nécessaires pour utiliser sous Eclipse les fonctionnalités de SmartTools.

Il s'agit principalement des plugins :

- *Absynt (langage déclaratif utilisé pour définir une syntaxe abstraite d'un langage)*
- *Cosynt (spécification décrivant la syntaxe concrète d'une grammaire et son affichage)*
- *Cdml (langage décrivant un composant)*

Démarche :

Nous avons réitéré la démarche précédente (cf. étape 2) en utilisant le PDE et nous avons vérifié le bon fonctionnement de nos nouveaux plugins.

Etape 6 : Création simplifiée de plugin Eclipse pour les composants de SmartTools

Motivation :

Permettre la création des plugin Eclipse correspondant aux composants SmartTools sans aucune manipulation d'Eclipse (il suffit juste de charger le projet existant).

Démarche :

La création d'un plugin Eclipse demande de fournir un nombre important d'informations (renseignées avec l'aide d'un wizard). Or toutes ces données sont stockées dans des fichiers tels que ***.classpath, .project, plugin.xml, manifest.mf*** et ***build.properties***.
Nous avons donc choisi de sauvegarder sous CVS toutes ces informations pour chaque plugin. Ainsi, nous disposions chacune des sources des différents plugins pour pouvoir les modifier, sans avoir à les recréer de façon à travailler efficacement en groupe.

Etape 7 : Création de plugin générique

Motivation :

Lors de la conception des différents plugins, nous nous sommes rendues compte que l'approche Eclipse induit l'écriture de code très semblable. Plus précisément, pour chaque composant, le plugin éditeur (fourni par la plateforme) était étendu par héritage et nécessite l'écriture de classes assez semblables.

Notre principal objectif dans cette étape a été de définir un plugin générique (valide pour l'ensemble de nos composants) pour éviter de réécrire des classes similaires. Par exemple, le code nécessaire pour mettre en œuvre la vue graphique outline est dans notre cas identique pour l'ensemble de nos composants.

En suivant cette approche, d'autres fonctionnalités ont pu être factorisées. Le résultat de cette démarche est de réduire au strict minimum le code écrit à la main pour chaque plugin.

Démarche :

Nous avons crée un plugin (***st-editor***) qui contient toutes ces fonctionnalités génériques. Tous nos plugins devront dépendre de ce plugin. La démarche a consisté à regarder l'ensemble des classes contenues dans le package ***éditeur*** pour essayer identifier les classes n'ayant pas d'informations spécifiques aux langages.

Classiquement, pour créer un éditeur, il est nécessaire de créer la classe éditeur étendant ***TextEditor*** proposant les fonctionnalités voulues. Cependant, de nombreuses parties de cette classe se sont avérées être génériques dans notre cas, comme la création du modèle et l'appel du parser. Nous avons voulu factoriser ces parties de code par la création d'une sous-classe **St-*Editor*** de la classe ***TextEditor***. ***St-Editor*** est contenue dans le plugin générique. Pour rajouter des fonctionnalités spécifiques, il suffit donc d'étendre cette classe générique.

La *outline* quant à elle est totalement générique. En effet, SmartTools génère automatiquement le modèle utilise pour créer la *outline*. Ainsi, les méthodes de création de la *outline* sont génériques.
Ces méthodes sont basées sur les mêmes services (arbre DOM). C'est la raison pour laquelle nous avons créé des classes totalement génériques: ***OutlinePage, ModelContentProvider*** et ***ModelLabelProvider***. Ces classes permettent par exemple de récupérer le parent d'un nœud, action identique pour chaque langage. Nous avons donc intégré les fonctionnalités de la *outline* au plugin générique ***St-Editor***.

Dans le cadre de SmartTools, la *facade* (design pattern ***facade***) peut être générée automatiquement à partir de la description CDML. Cette classe étend la super classe ***LogicalDocument*** qui contient les services communs au composant de type *document* de SmartTools. Grâce à cette classe il a été possible de factoriser un ensemble de services (par exemple la sauvegarde en format XML, avec la méthode ***save*** sans paramètre).

Etape 8 : Génération automatique du code java dans CDML

Motivation :

Dans cette partie nous avons voulu valider le fait que la génération de code était possible en utilisant les plugins de base de SmartTools (exemple : CDML).
Nous avons voulu effectuer cette opération en restant dans l'environnement d'exécution d'Eclipse (par exemple sans utiliser de script externe comme ANT).
Pour effectuer cette opération, il fallait que les composants *absynt, cosynt, componentgenerator* et *cdml* soient des plugins Eclipse.

À partir du composant CDML, nous avons utilisé les fonctionnalités de SmartTools permettant la génération de code.
Nous avons testé cette génération sur le composant XML.

Démarche :

Pour ce faire, nous avons utilisé une fonctionnalité de SmartTools faisant appel à d'autres fonctionnalités afin de générer le code à l'aide d'un simple bouton.

Nous avons réussi à générer le composant XML. La seule difficulté rencontrée concernait la définition du *classpath*. En effet, cette génération a besoin de chercher des informations dans les deux contextes (SmartTools et Eclipse). Nous avons alors résolu ce problème en rajoutant une variable permettant de différencier ces deux contextes et ainsi permettre la génération de code dans ces deux cas.

Cette génération se fait partiellement. Il faudrait alors compléter notre démarche et l'améliorer.

Etape 9 : Création automatique de modèles

Motivation :

Lorsque on crée un plugin sous Eclipse, il faut créer son modèle et cette définition se fait manuellement par le développeur. Par contre dans SmartTools la définition de ce modèle est générée automatiquement à partir d'une description déclarative (DTD, Absynt ...). De plus SmartTools génère le parseur permettant de passer de la forme concrète vers le modèle.
L'objectif de cette étape est d'intégrer cet ensemble de code généré automatiquement par SmartTools dans l'environnement Eclipse.

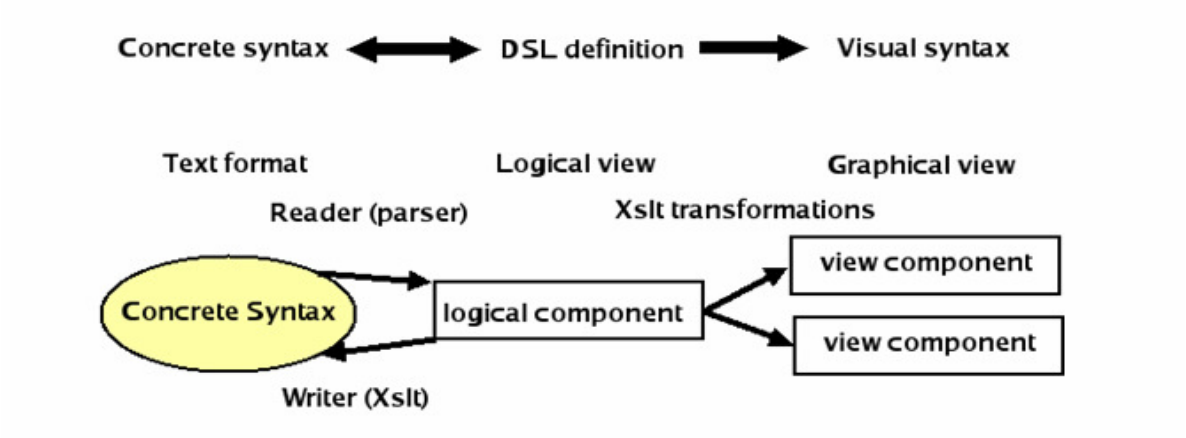


Figure 3 : Transformation d'une vue graphique

Démarche :

Eclipse dispose d'une vue graphique pour visualiser la structure logique des documents manipulés. Nous avons étudié le mode de fonctionnement de cette *outline* qui correspond à une représentation graphique du modèle du langage utilisé.
A partir d'un fichier Absynt, SmartTools est capable de générer un modèle (au dessus de DOM) et à partir d'un fichier Cosynt il est capable de générer le parseur pour une syntaxe concrète donnée.

La première étape a été de récupérer le modèle associé au texte contenu dans l'éditeur courant. Nous avons créé un bouton (*NomLangage outline*) permettant de synchroniser le texte et le modèle. L'action associée à ce bouton récupère le texte de l'éditeur qui est traité par le parseur pour obtenir le modèle.
La synchronisation s'effectue en parsant à nouveau l'ensemble du texte de l'éditeur

La seconde étape s'occupe de la synchronisation de la vue *outline* avec le texte de l'éditeur.
Pour cela, nous avions deux traitements à concevoir, les classes qui gèrent la vue graphique (*cf. étape 8*) et les classes qui gèrent la synchronisation entre l'éditeur et cette vue. À chaque fois que l'on crée un modèle (méthode ***CreateModele***), il fallait prévenir cette classe (par une méthode ***updateModele***) que le modèle était modifié.

La troisième étape s'occupe d'intégrer les possibilités de transformations d'un modèle à un autre. En effet la spécification Cosynt permet de définir différentes transformations en générant des feuilles XSL. Nous avons créé des boutons permettant d'appeler la méthode ***save*** (de SmartTools) qui prend en paramètre la feuille de style que l'on veut appliquer. Cette transformation s'effectuera sur le modèle.
Par exemple pour tous les langages nous avons fait un bouton permettant de sauver le fichier sous forme XML. Pour le langage CDML nous avons fait un bouton permettant de sauver le fichier sous forme HTML.

Conclusion

Voici tous les composants SmartTools que nous avons transformés en plugins Eclipse :

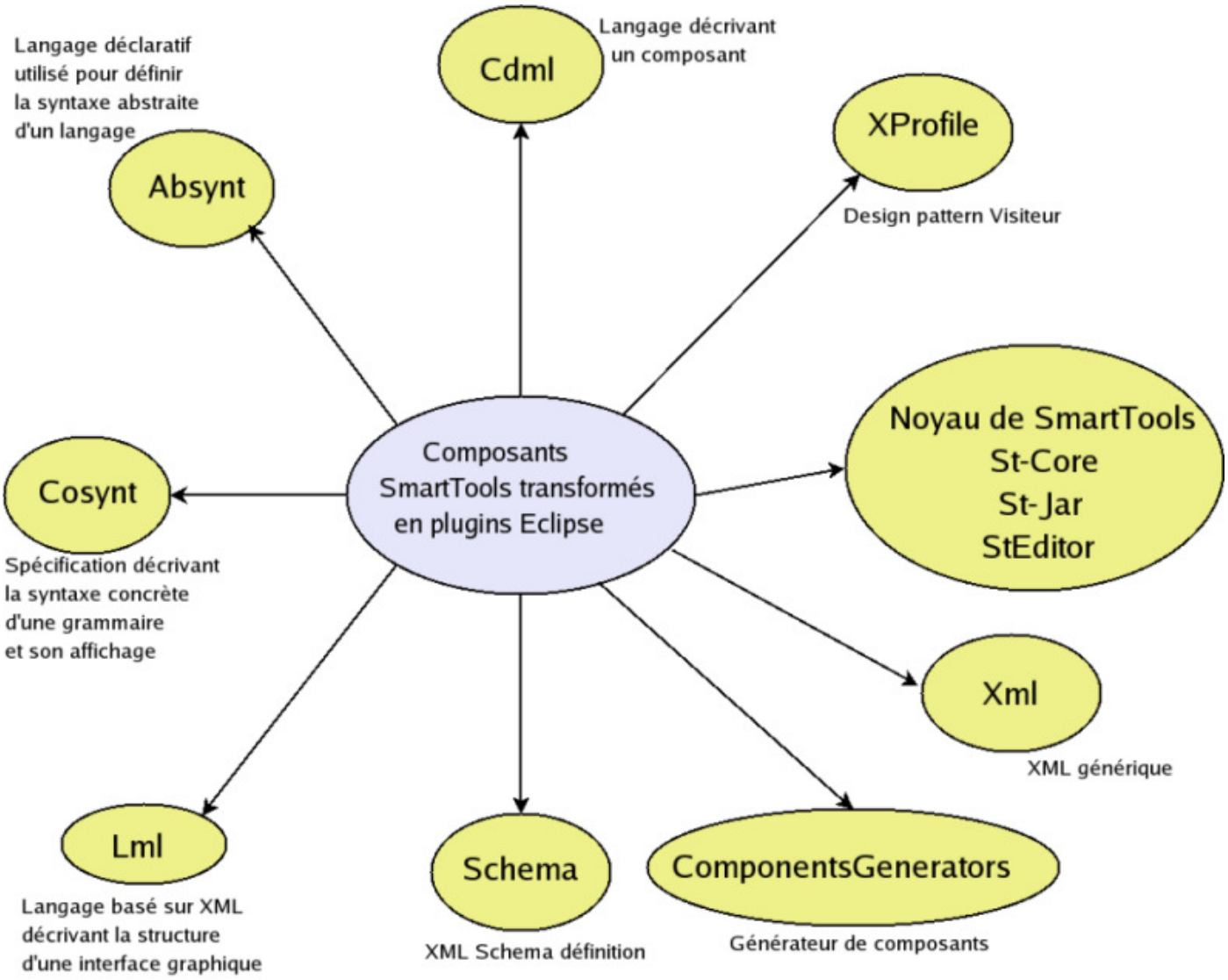


Figure 4 : Composants SmartTools transformés en plugins Eclipse

Nous avons transformé les DSL en plugins Eclipse avec des environnements minimaux. Il faudrait bien évidemment enrichir ces environnements afin d'obtenir un IDE pour chacun des langages. Chaque DSL est doté d'un compilateur, il faudrait alors envisager une gestion des erreurs pour chacun des langages.

A partir de tous ces plugins, nous avons réalisé un test de génération de composant. Nous avons donc réussi à générer le composant XML. Il faudrait compléter cette génération pour le moment incomplète.

Dans l'optique d'intégrer toutes les technologies de SmartTools dans Eclipse, nous avons crée une vue supplémentaire. Notre vue HTML permet de visualiser le modèle (généré par SmartTools) sous forme de fichiers HTML à partir de transformation XSL. Ce même principe sera utilise pour créer les autres vues (exemple : vues graphiques).

Notre plus grande difficulté était de travailler sur 2 projets existants et conséquents (SmartTools et Eclipse).

Grâce à ce projet, nous avons eu un aperçu de la programmation par composants. Ce projet nous a également sensibilisé au travail en groupe.
Notre projet correspond à une étude préliminaire (Il sera prochainement sur http://www-sop.inria.fr/smartool). Il a permis à notre encadrant de faire une proposition ODL (Opération de Développement Logiciel) qui a été accepté, à la direction nationale de l'INRIA.

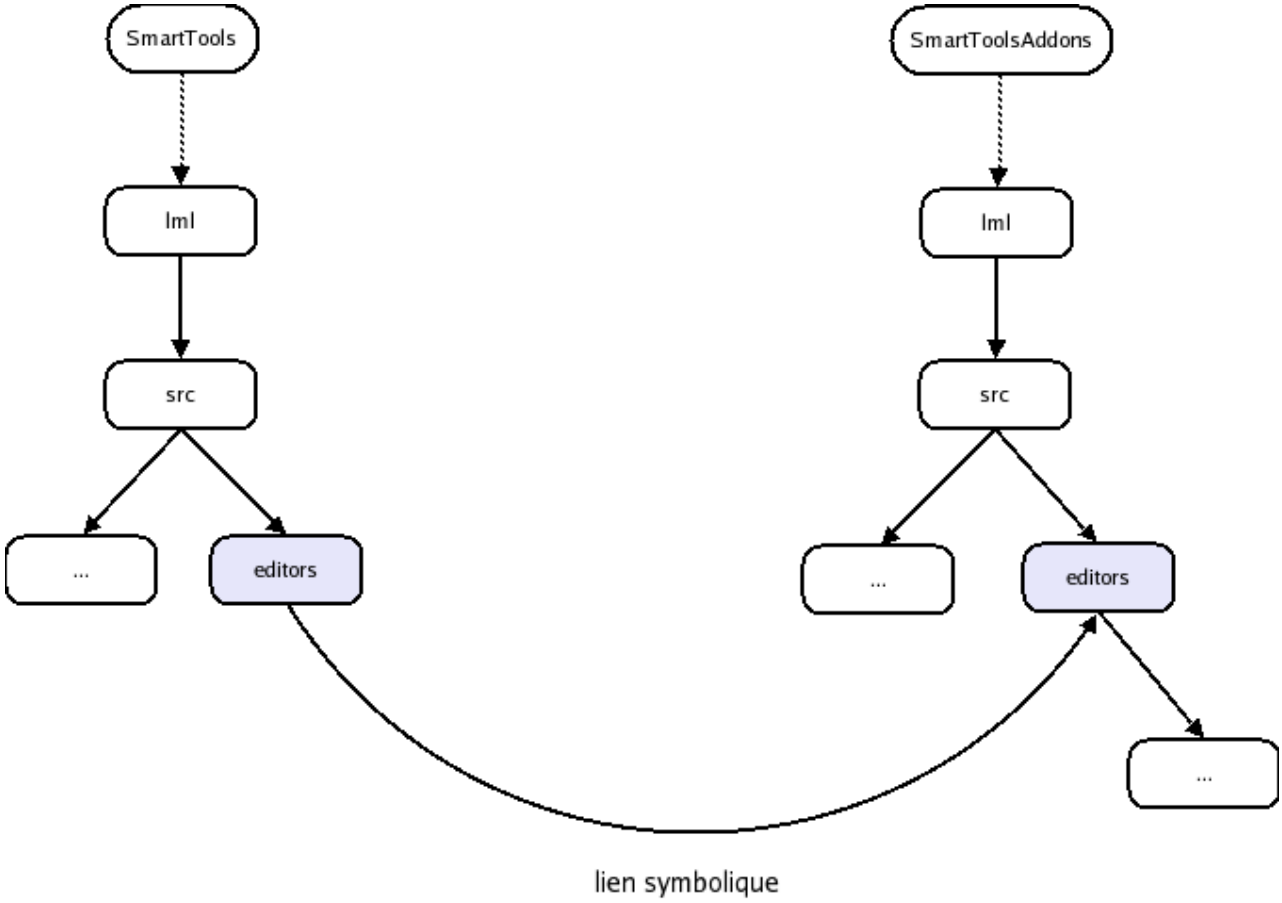
Table des illustrations

Annexes

Table des annexes

1. Procédure à suivre pour commencer le projet

- ***Récupération de SmartTools dans la base CVS :***
  cvs checkout SmartTools
- ***Régénération de SmartTools :***
  cd SmartTools/ant/developer/
  ./build cleaneclipse (Suppression des fichiers relatifs aux plugins Eclipse)
  ./build all
- ***Mise à jour :***
  cd SmartTools
  cvs update (Récupération des fichiers supprimés lors du clean)
  cd SmartTools/src/core/src
  ln -s ../fr fr
- ***Pour tous les plugins contenants un éditeur :***
  cd SmartTools/src/components/absynt/src/lml
  ln -s ../../../../../../SmartToolsAddons/src/components/lml/src/lml/editors editors



Faire de même pour les plugins suivants :

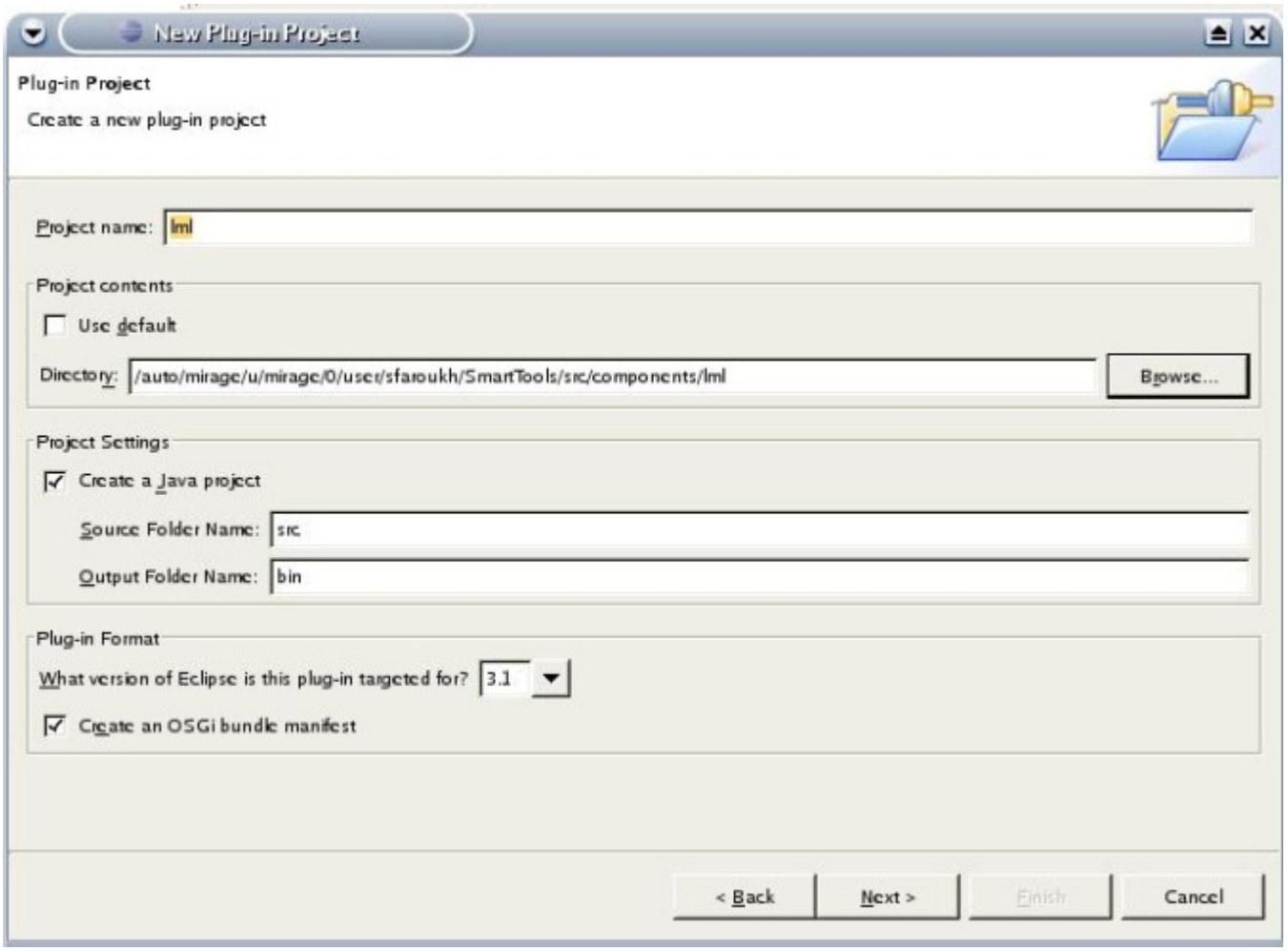
- cosynt
- absynt
- xprofile
- cdml
- xml
- schema

- ***Mise à jour :***
  cd SmartTools
  cvs update (Mise à jour des éditeurs)
- ***Précision du contexte de travail ;***
  cp -r SmartTools/ant/developer ~/eclipse/.
  cd eclipse/ant/developer
  Éditer le fichier SmartTools.properties
  Modifier : SmartTools.Eclipse=yes
  De plus, il faut préciser la variable qui précise la situation relative de SmartTools par rapport à Eclipse.
- Lancer Eclipse et importer (File->Import->Existing Project into workspace) les plugins suivants :
  - st-jar : /SmartTools/lib
  - st-core :/SmartTools/src/core
  - st-editor : /SmartToolsAddons/contrib/components/st_editor
  - lml : /SmartTools/src/components/lml
  - xml : /SmartTools/contrib/components/xml
  - schema :/SmartTools/contrib/components/schema
  - xprofile : /SmartTools/src/components/xprofile
  - componentsgenerators : /SmartTools/src/components/componentsgenerators
  - absynt : /SmartTools/src/components/absynt
  - cosynt : /SmartTools/src/components/cosynt
  - cdml : /SmartTools/contrib/components/cdml

2. Transformation d'un composant SmartTools en plugin Eclipse à l'aide de Wizard

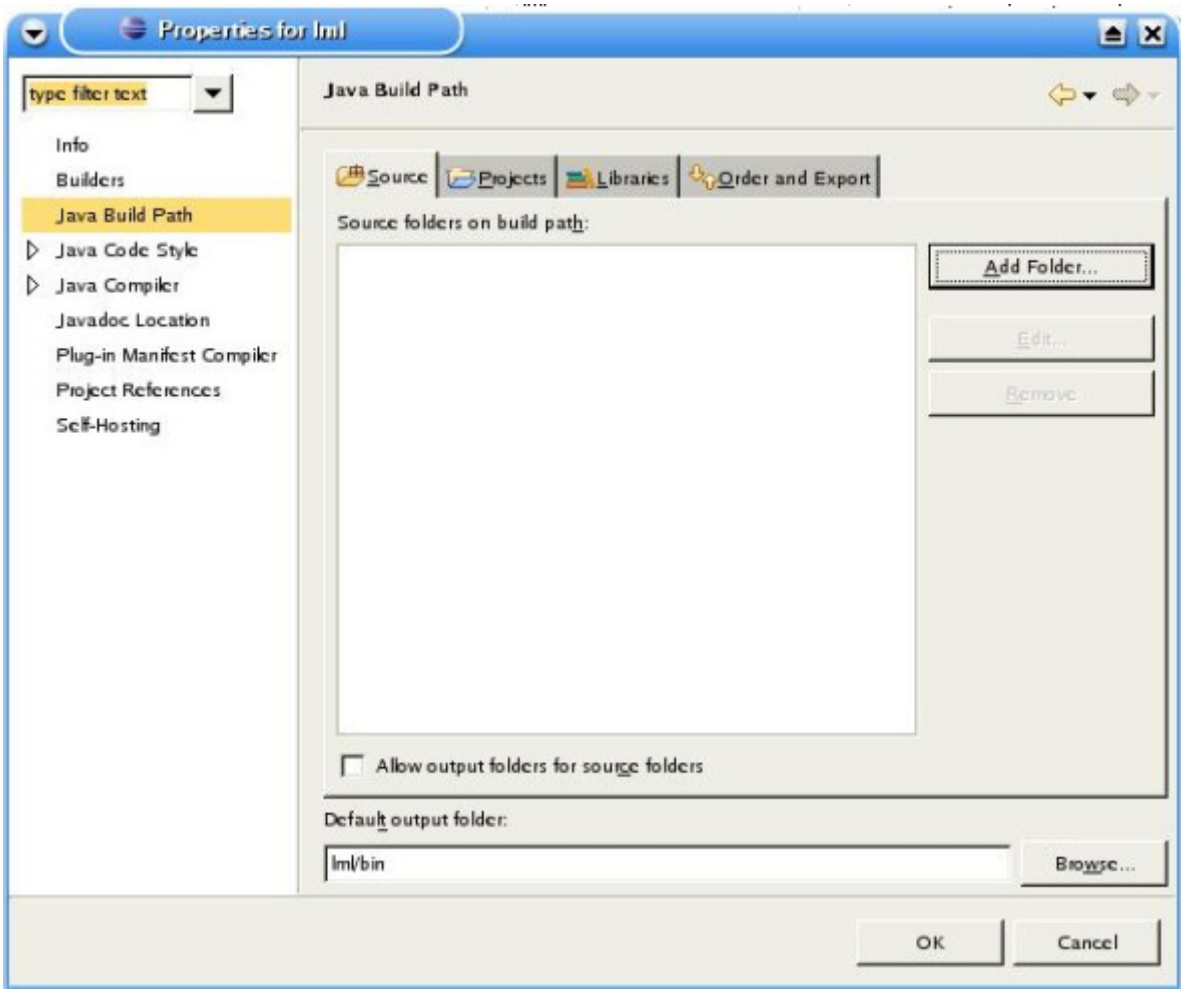
Nous allons ici prendre pour exemple la transformation du composant lml en plugin.

- Créez un nouveau plugin (File -> New -> Plug-in Project)*lml*
- Pour le contenu du projet, choisissez le répertoire correspondant au composant SmartTools
- Choisissez l'extension editor



3. Remettre à jour les packages de la hiérarchie

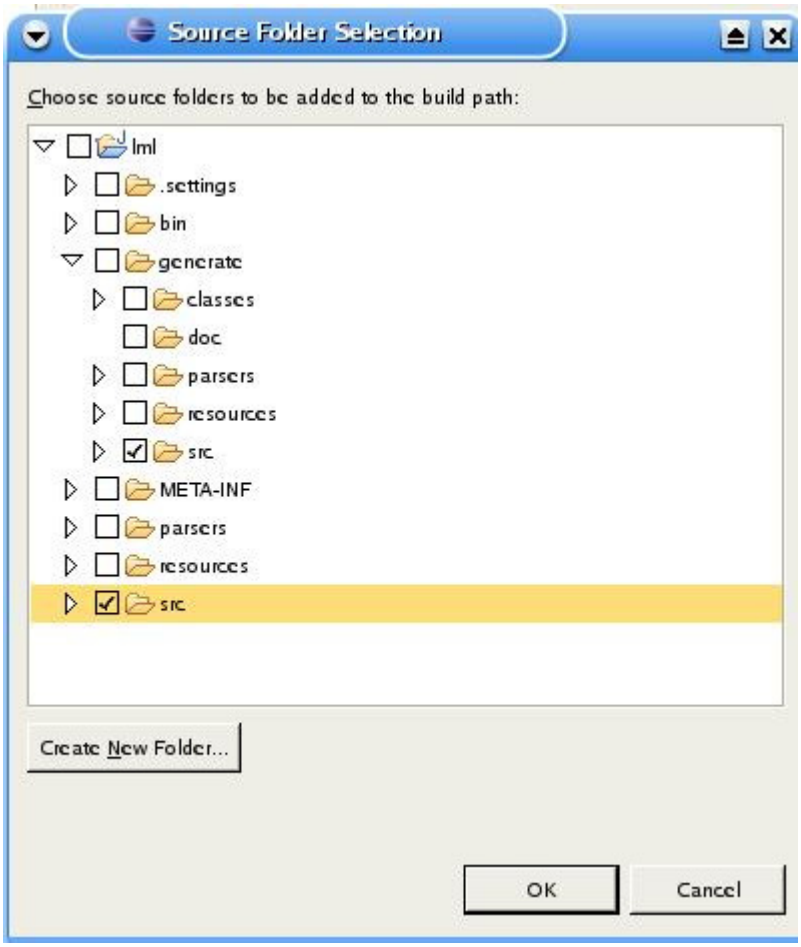
- Faites un clic droit sur le projet pour avoir ses propriétés et choisissez Java Build Path puis l'onglet Source.



- Supprimer tout ce que cet onglet contient.

4. Préciser les sources à compiler

- Ajouter alors les répertoires src et generate/src comme indiqué ci-dessous :



5. Personnalisation du plugin dans le contexte SmartTools

Copier sous SmartTools les fichiers .classpath, plugin.xml et META-INF/Manifest.mf.
A l'aide du PDE, vous devez modifier certains éléments et ne pas en modifier certains

Les éléments à ne pas modifier
Les données présentent sous les onglets dependencies,Extensions,Extension Points,Build, Manifest.mf, build.properties

Les éléments à modifier

Cela concerne les onglets Overview , Runtime et plugin.xml.

- L'onglet Overview Il renseigne des informations générales telles que le nom du plugin.
  Modifier le en prenant soin de mettre les noms relatifs à votre plugin(le nom, l'ID, la classe XPlugin.java).
- L'onglet Runtime Il renseigne les packages à exporter. Ceux précisés sont erronés. Effacez-les à l'aide du bouton REMOVE. Ensuite, grace au bouton ADD, ajouter tous les packages exportables proposés. Ces changements modifient automatiquement le fichier manifest.mf.
- L'onglet plugin.xml Dans ce fichiers, vous devez remplacer toutes les références à l'ancien plugin par le nom de votre nouveau plugin. Remplacez donc par exemple tous les lml par X. Prenez soin cependant à modifier correctement l'extension à deux endroits du fichier.
  En effet, dans certains cas le noms du plugin est différent de l'extension.
  Voici les endroits du fichier plugin.xml concernés:
  extensions="lml" <selection class="org.eclipse.core.resources.IFile" name="*.lml"/>
  Vous pouvez maintenant sauver tous vos fichiers sous cvs.

  Copier sous SmartToolsAddons la directory editors d'un plugin déjà existant dans SmartToolsAddons/src/components/X/src/X/
  par exemple celui de lml SmartToolsAddons/src/components/lml/src/lml/editors
  A ce stade, vous disposez du code java d'un autre plugin. Il faut donc modifier celles-ci en conséquence.
- Changer le nom des packages de toutes les classes
- Modifier la classe Editeur.java: Changer dans les imports celui concernant la facade.
  Mettre le type correspondant à pour la variable fFacade.
- Adapter la classe de coloration syntaxique(Scanner.java).Pour cela réfèrez vous à ce chapitre.

6. Coloration syntaxique

Motivation :

Offrir la fonctionnalité d´Eclipse de coloration syntaxique pour une meilleure visibilité du code source

Démarche :

On crée la classe Editors/Scanner.java qui étend la classe RuleBasedScanner
On définit des (tokens) à colorer par des règles (rules)
Le parseur intégré dans l´ éditeur d´Eclipse va appliquer ces règles et colorer les tokens

Technique :

Pour chaque token on choisit une couleur

```
private static Color COMMENT_COLOR = new Color(Display.getCurrent(), new RGB(0, 200, 0));
IToken commentToken = new Token(new TextAttribute(COMMENT_COLOR));
```

On implémente le constructeur :

- On va créer un tableau de règles :
  `IRule[] rules = new IRule[10];`
- on choisit une règle prédéfinie pour le colorer :
  - 1er cas : on colore toute la ligne définie par un marqueur spécifique du début de la ligne
    `rules[4] = new EndOfLineRule("//", commentToken);`
  - 2ème cas : on colore un token dont on connaît le marqueur de début et celui de fin :
    `rules[0] = new MultiLineRule("<layout",">" ,layoutToken);`
  - 3ème cas : on colore un token qui est un mot clé
    `rules[6] = new KeyWordRule("BML",viewToken );`
- À la fin on affecte les règles :
  `setRules(rules);`

7. Ajout d'un bouton associé à l'éditeur

Principe :

Pour créer un bouton, il est nécessaire d'utiliser le plugin **org.eclipse.ui.editorActions**.

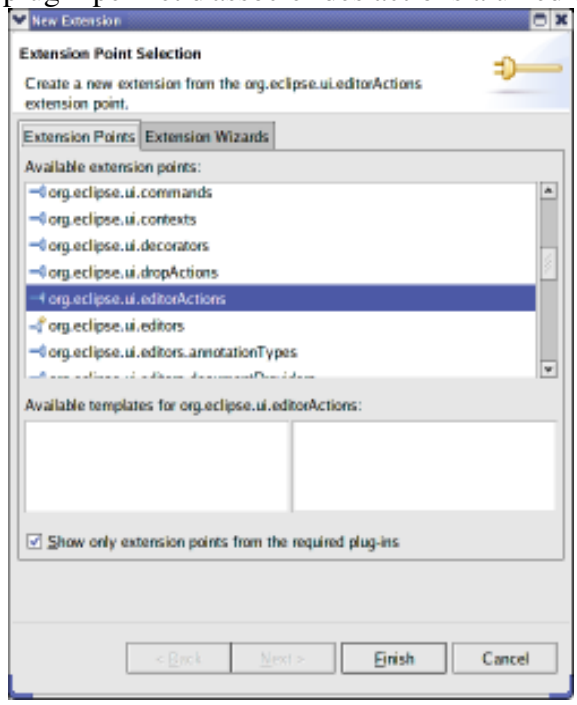
Il faut alors deux informations :

- préciser l'éditeur sur lequel est rattaché le bouton
- préciser l'action du bouton

Démarche :

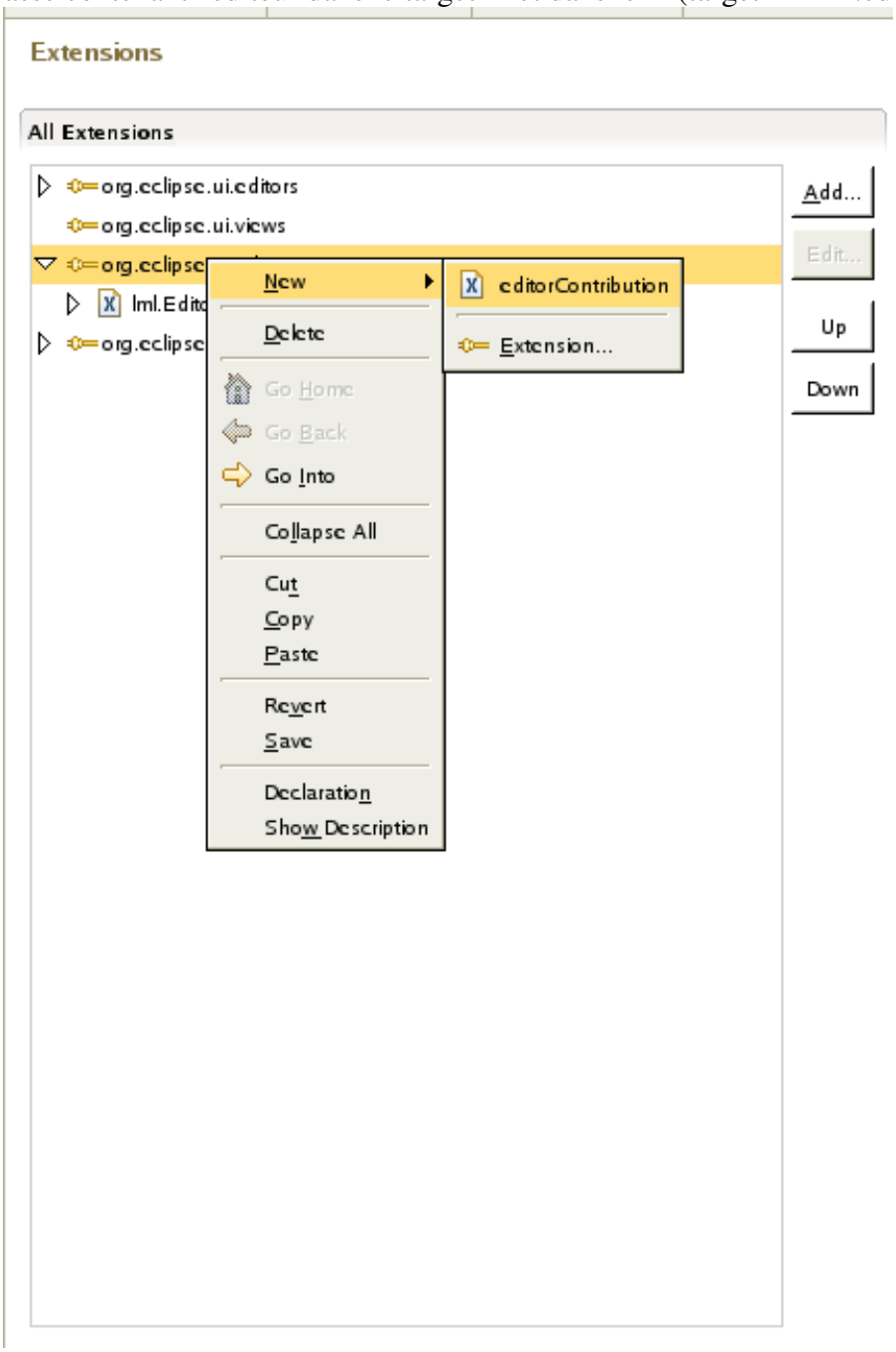
Pour créer un bouton, il faut tout d'abord aller dans l'onglet **Extensions** du plugin.

- Ajouter le plugin **org.eclipse.ui.editorActions** a partir de l'onglet ExtensionPoint.
  Ce plugin permet d'associer des actions à un éditeur

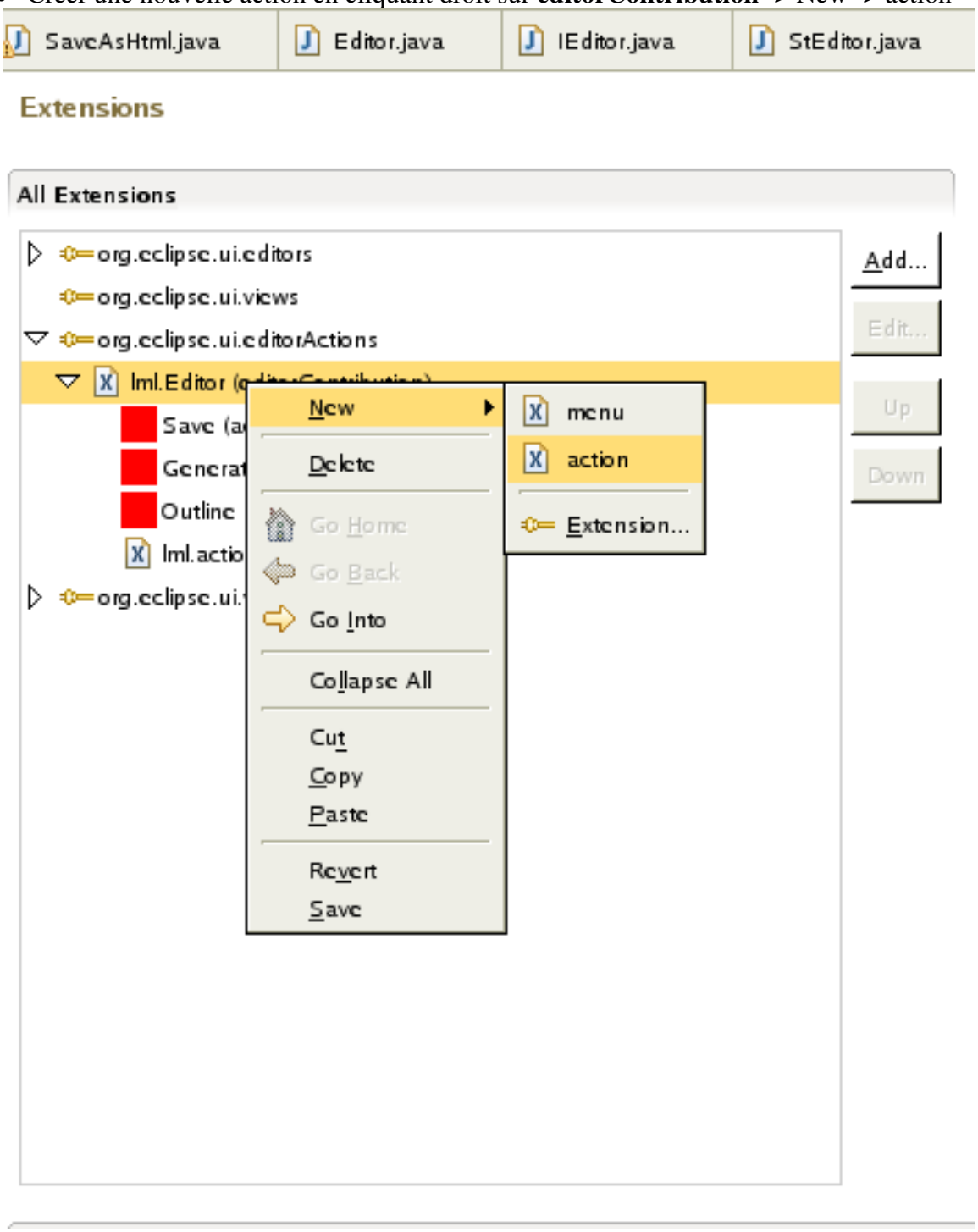


- Créer un nouveau editorContribution en cliquant droit sur org.ui.editor -> New -> editorContribution

- Choisir la classe contenant l'éditeur dans le targetID et dans le iD(targetID=lml.editors.Editor)



- Créer une nouvelle action en cliquant droit sur **editorContribution** -> New -> action



- Choisissez la classe implémentant l'action de votre bouton.
  Il faut également préciser :

  - label : le nom de votre bouton
  - toolbarPath : Le nom de votre classe éditeur (exemple : Editor)
  - icon : le chemin vers l'icône du bouton
- Implémenter la classe correspondant à l'action du bouton

Voici un exemple de classe implémentant un bouton :

```
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IEditorActionDelegate;
import org.eclipse.ui.IEditorPart;
import st_editor.*;

public class SaveAction implements IEditorActionDelegate{
   private IEditorPart editor;
   public SaveAction() {
     // do nothing
   }
   public void run(IAction action) {
     ((IEditor)editor).saveAction();
   }
   public void selectionChanged(IAction action, ISelection selection) {
     // do nothing
   }
   public void setActiveEditor(IAction action, IEditorPart editor) {
     this.editor = editor;
   }
}
```

Voici un extrait du fichier plugin.xml relatif au bouton :

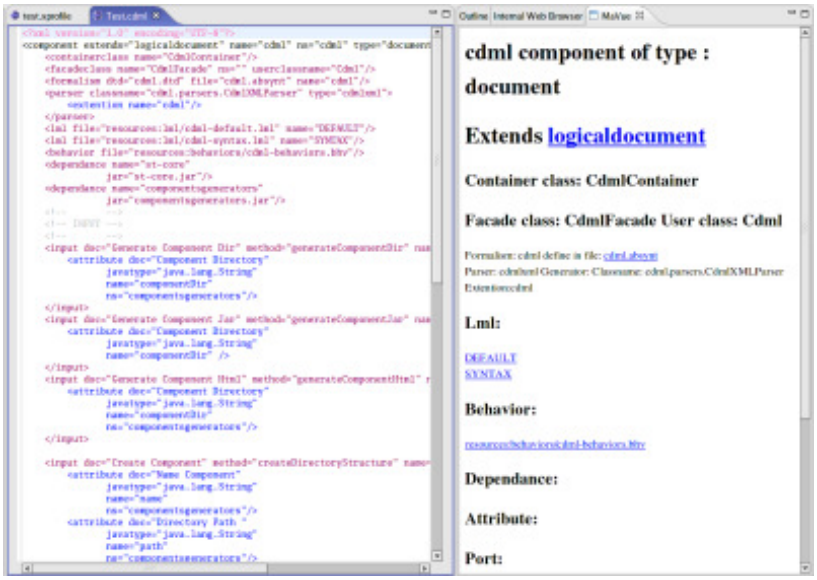
```
<extension point="org.eclipse.ui.editorActions">
   <editorContribution
   id="lml.Editor"
   targetID="lml.editors.Editor">

   <action
     class="st_editor.SaveAction"
     disabledIcon="icons/sample.gif"
     icon="icons/sample.gif"
     id="lml3"
     label="Save"
     style="push"
     toolbarPath="Editor"
     tooltip="LML Save"/>
```

8. Ajout d'une nouvelle vue

Prenons l'exemple de l'ajout de la vue HTML
Nous avons pour cela créé la classe WebView.java dans le plugin st_editor.
Cette classe doit étendre la classe **ViewPart**.

Il est nécessaire de rajouter org.eclipse.ui.views dans les dépendances.
La principale méthode a implémenté est **createPartControl(Composite parent)**.
C'est cette méthode qui renseigne le contenu de la vue.
Dans notre cas, il s'agit d'un browser. Nous avons donc rattaché le browser à la vue.
_browser = new Browser(displayArea, SWT.BORDER);



Il faut ensuite rajouter l'extension *org.eclipse.ui.views* pour spécifier

- où placer la vue dans le menu Windows->Show View. Dans notre cas, il s'agit de la catégorie "BASIC"
- quelle classe utiliser, dans notre cas il s'agit de WebView.
  Voici un extrait du fichier plugin.xml:

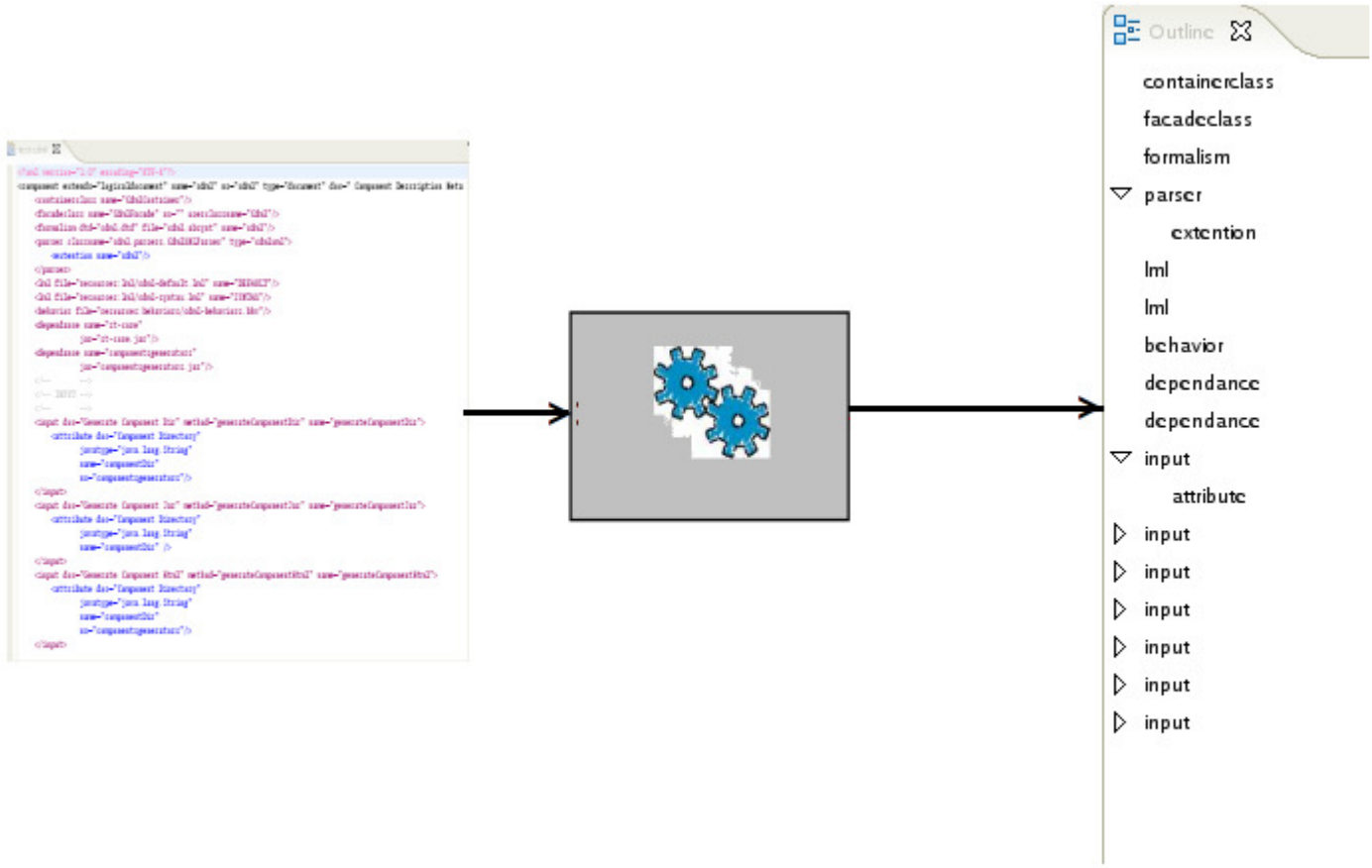
```
<extension
point="org.eclipse.ui.views">
<category name="Basic"
id="fr.improve.appli.viewCategory"/>
<view name="MaVue"
  icon="icons/sample.gif"

 category="fr.improve.appli.viewCatego
ry"
  class="st_editor.WebView"
  id="st_editor.WebView"/>
</extension>
```

9. Outline

Principe :

L'objectif est de synchroniser la outline (représentation graphique du modèle) avec le texte de l'éditeur :



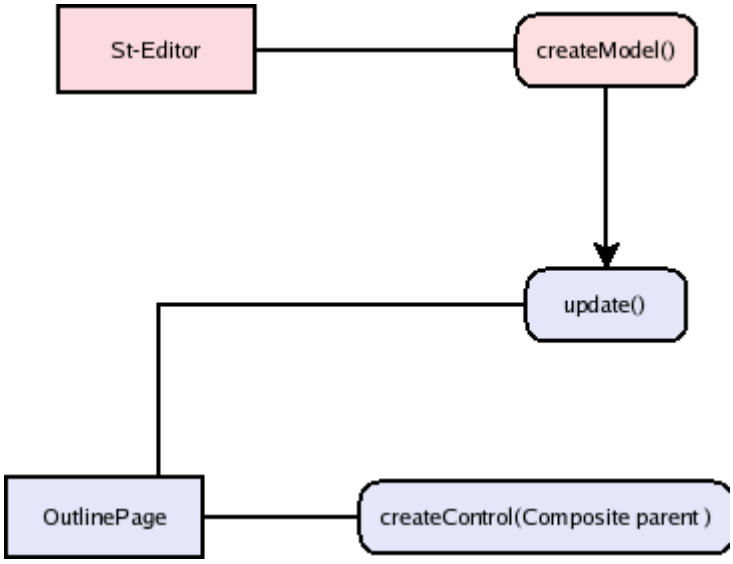
Démarche :

Il y a deux traitements à concevoir :
- gérer la vue graphique
- gérer la synchronisation entre la vue graphique et l'éditeur

A chaque fois que l'on créé un modèle (méthode createModel), il faut prévenir la classe OutlinePage avec la méthode update.
La outline est rattaché à l'éditeur grâce à la méthode **getAdapter()**

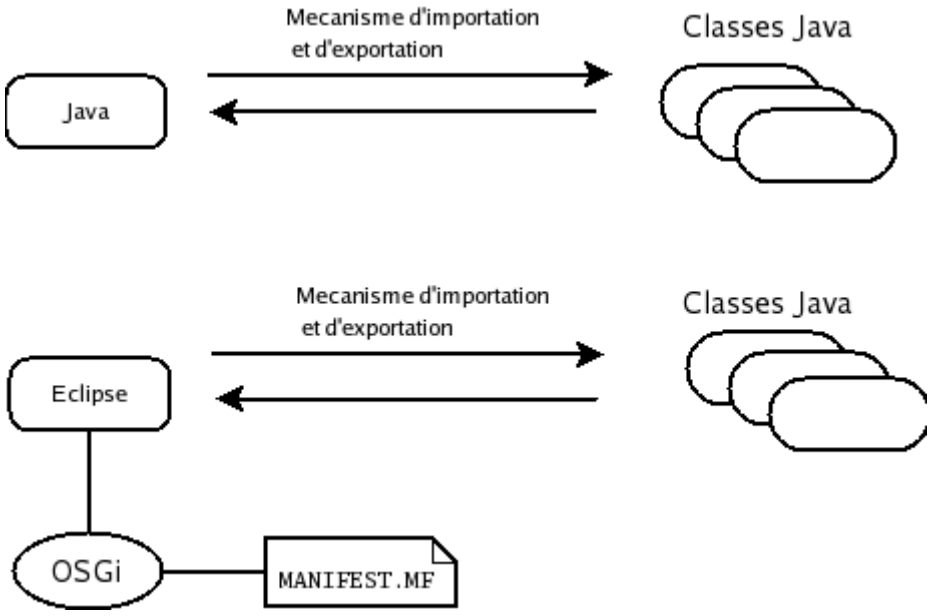
```
public Object getAdapter(Class adapter) {
if (IContentOutlinePage.class.equals(adapter)) {
if (fOutlinePage == null){
fOutlinePage = new OutlinePage(this);
}
return fOutlinePage;
}
```

Le mécanisme d'association de la outline avec l'éditeur se fait dans une des super-classes(nous n'avons pas encore compris ce mécanisme ...).



10. Gestion de l'importation et de l'exportation

Lors de l'exécution Eclipse utilise les composants OSGI qui ont des règles d'importation et exportation. Ces règles sont contenues dans le fichier manifest.mf qui est lu à l'execution(runtime). Chaque plugin doit préciser quels packages sont a exportes et a importer.
Les mécanismes d'importation et d'exportation dans Eclipse et dans Java ne sont pas les mêmes.
Il faut donc spécifier ce mécanisme dans le fichier Manifest.mf du plugin Eclipse car c'est ce fichier qui est lu par les composants OSGI d'Eclipse.



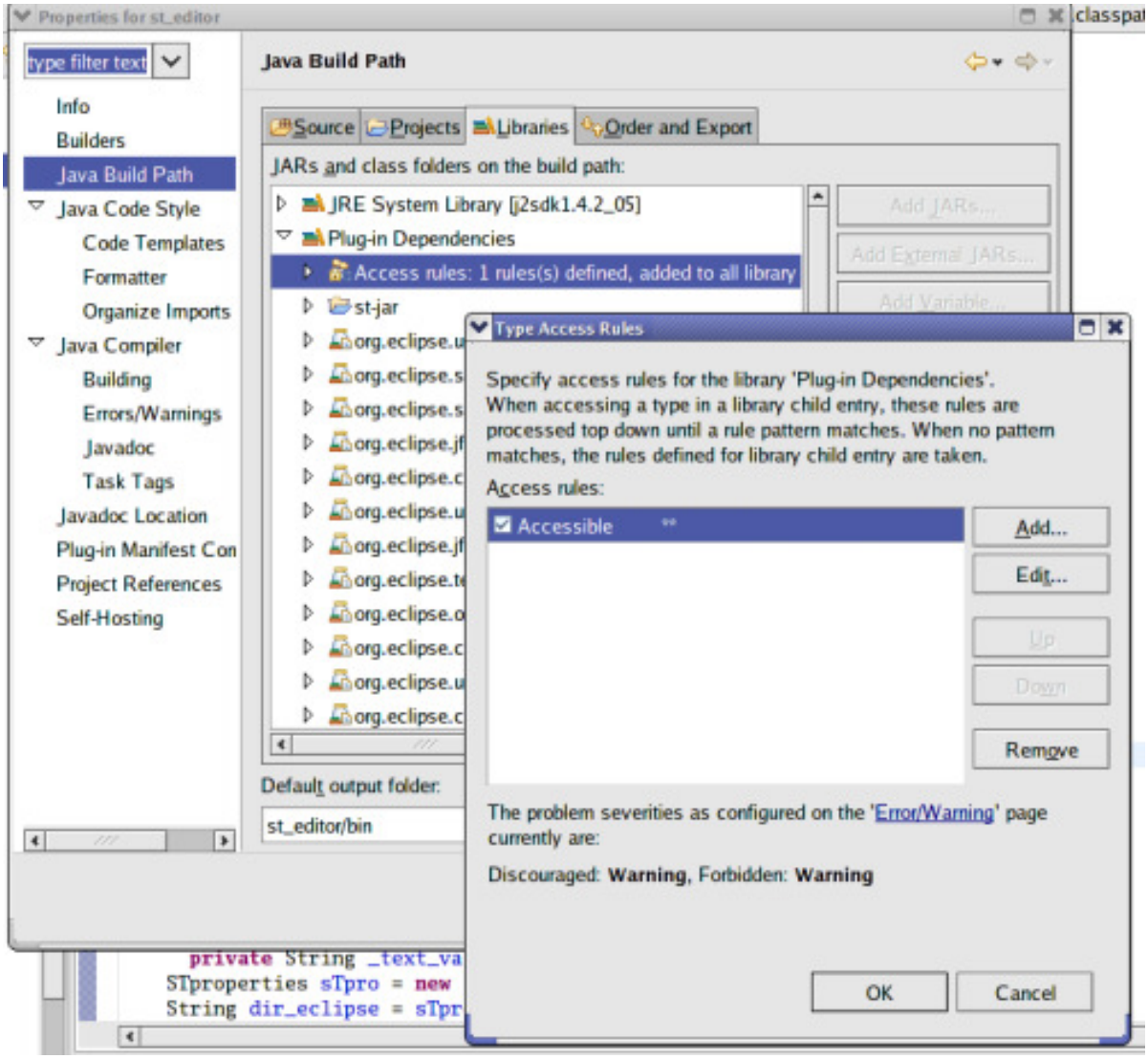
En particulier, pour le plugin coeur de SmartTools, comme nous utilisons la reflexivite, il faut preciser que ce plugin peut importer dynamiquement n'importe quel package.
D'ou la ligne DynamicImport-Package: * contenu dans le fichier manifest.mf de st-core.

11. Gestion des problèmes d'ACCESS RULES

La compilation sous Eclipse utilise des règles d'importations. Il faut préciser le droit d accès sur les packages importés. Si on ne précise pas ces droits, on obtient des erreurs à la compilation. Pour palier aux problèmes d' ACCESS RULES, il est nécessaire d'éditer le fichier .classpath.

Voici l'allure de ce fichier
<classpathentry kind="con" path="org.eclipse.pde.core.requiredPlugins" >
<accessrules >
*<accessrule kind="accessible" pattern="**"/ >*
</accessrules >
</classpathentry >

Notez qu'il est aussi possible de changer ces ACCESS RULES en utilisant :
Windows->Properties. Sélectionner Java Build Path. Choisir l'onglet Librairies. Dérouler Plug-in Dependencies et double cliquez sur ACCESSRULES pour mettre les règles de votre choix.



12. Création de vue générique

Voici comment utiliser les vues créées par SmartTools(SWING) dans Eclipse.
Les différentes vues doivent dériver de la classe générique StView de StEditor qui étend ViewPart .

Cette classe contient la méthode suivante:

```
public void createPartControl(Composite parent,String title,String transform ,String styleSheet,String
behaviors) {

        /*Récupération de l'éditeur courant*/
        fEditor = (StEditor)(getSite().getPage().getActiveEditor());
        /*Récupération du nom du fichier associé*/
        String docRef = fEditor.getFilePath();
        /*Récupération de la facade*/
        facade = fEditor.getFacade();
        displayArea = parent ;
        /*Construction de GdocView pour construire la vue */
        ged = new GdocView(title, docRef, behaviors,transform, styleSheet);
        /*Permet de récupérer les actions sur la vue*/
        GdocViewCtrl ctrl = new GdocViewCtrl(ged,facade);
        /*Ajoute la vue à la liste de vues associées à la facade pour pouvoir notifier toutes les vues lors de la
        sélection d'un élèment dans l'une d'entre elles*/
        facade.addView(ged);

        try {
                displayArea.setBounds(0, 0, 200, 250);
                /*Pour créer le pont, le composite doit être construit avec la variable SWT.EMBEDDED */
                Composite workArea = new Composite(displayArea, SWT.EMBEDDED);
                java.awt.Frame frame = SWT_AWT.new_Frame(workArea);//pont
                ged.buildContent(facade.getTree());
                frame.add(ged.getComponent());

        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

Ainsi dans chaque plugin, on peut construire une classe(par exemple CdmlSyntax pour CDML) qui étendra la classe générique StView. La Méthode createPartControl qui appelle la méthode de la classe mère avec les bons arguments.

```
public void createPartControl(Composite parent) {
        super.createPartControl(parent,"CdmlSyntax",
        "file:"+chemin+"/xsl/cdmlsyntaxbml.xsl",
        "file:"+chemin+"/css/cdmlsyntax-styles.css","file:"+dir_eclipse +
        "/contrib/components/cdml/resources/behaviors/cdml-behaviors.bhv");
}
```

Cette méthode est celle appelée par Eclipse lors de l'ouverture d'une vue.

En effet, pour chaque composant, il faut définir les vues existantes dans le fichier plugin.xml.
<extension point="org.eclipse.ui.views">
<category name="Vues CDML"
id="fr.improve.appli.viewCategory"/>
<view name="CDML XML"
icon="icons/sample.gif"
category="fr.improve.appli.viewCategory"
class="cdml.editors.CdmlXml"
id="cdml.editors.CdmlXml"/>
</extension>

Pour chaque composant doit avoir sa propre catégorie pour contenir ses vues.

Intérêt de l'utilisation de ses vues par rapport aux vues d'Eclipse:
Pour écrire un éditeur associé à un langage il est nécessaire d'implémenter un certain nombre de classes telle que Scanner.java.
Or il est fastidieux d'écrire toutes ces classes.
Or dans notre cas, nous n'avons du écrire qu'une seule ligne contenant les différents paramètres tels que la feuille de style ou la transformation(xsl)(générée automatiquement par SmartTools à partir de cdml-syntax.cosynt par exemple).

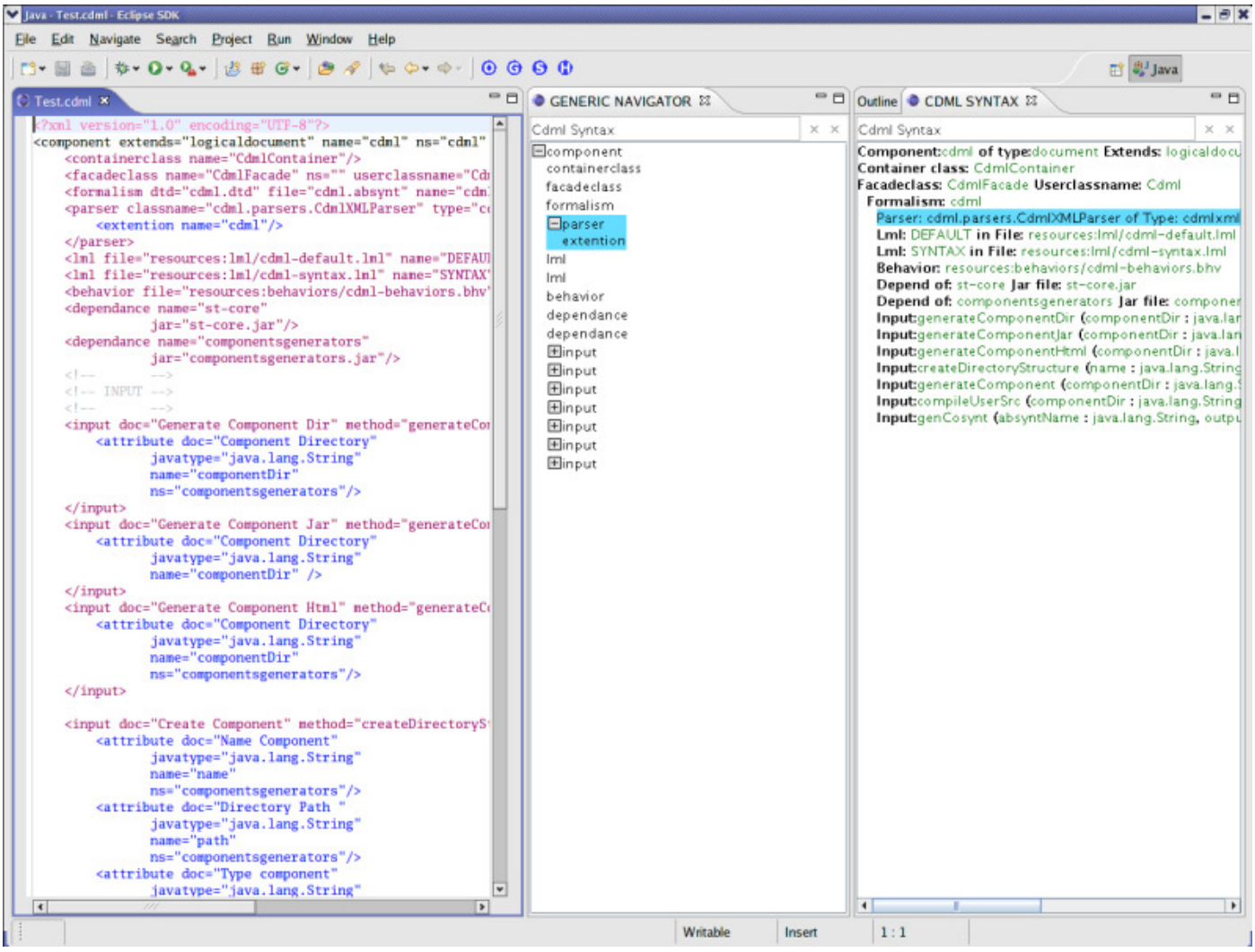
Changements effectués dans les classes de St Core:

- LogicalDocumentImpl Rajout de la notion de liste de vues.
- GabstractView && GEditorPan Rajout d'un test pour le choix des Icon selon le contexte(Eclipse ou SmartTools)
- GdocViewCtrl Rajout dans *public void processSelect(SelEvent sev)* de
  if(eclipse.equals("yes")){
  cdmlfacade.selection(cursel);
  }

Les vues actuellement disponibles:
Notez qu'il est nécessaire d'avoir sélectionné l'éditeur associé à la vue avant d'ouvrir celle-ci.
De plus, l'arbre étant construit grâce à l'appel de la outline, il est nécessaire d'appeler celle-ci avant d'ouvrir une vue.

- CDML
  - *CDML SYNTAX* Visualisation sous forme concrète.
  - *CDML XML* Visualisation sous forme XML.
  - *Browser View*Vue présentant le fichier généré par le bouton "CDML Save as HTML" de l'éditeur cdml dans un Browser.Cela nécessite donc d'avoir généré le fichier auparavant.
  - *HTML by JEDITORPAN*Vue représentant le même résultat que Browser View mais obtenu en utilisant la classe GEditorPan.
  - *GEditView* Permet de visualiser le code source de la page html visible grâce à HTML by JEDITORPAN.Possibilité d'éditer.
- ABSYNT
  - *ABSYNT SYNTAX* Visualisation sous forme concrète.
  - *ABSYNT XML* Visualisation sous forme XML
  - *Generic Navigator* Visualisation des éléments du modèle.

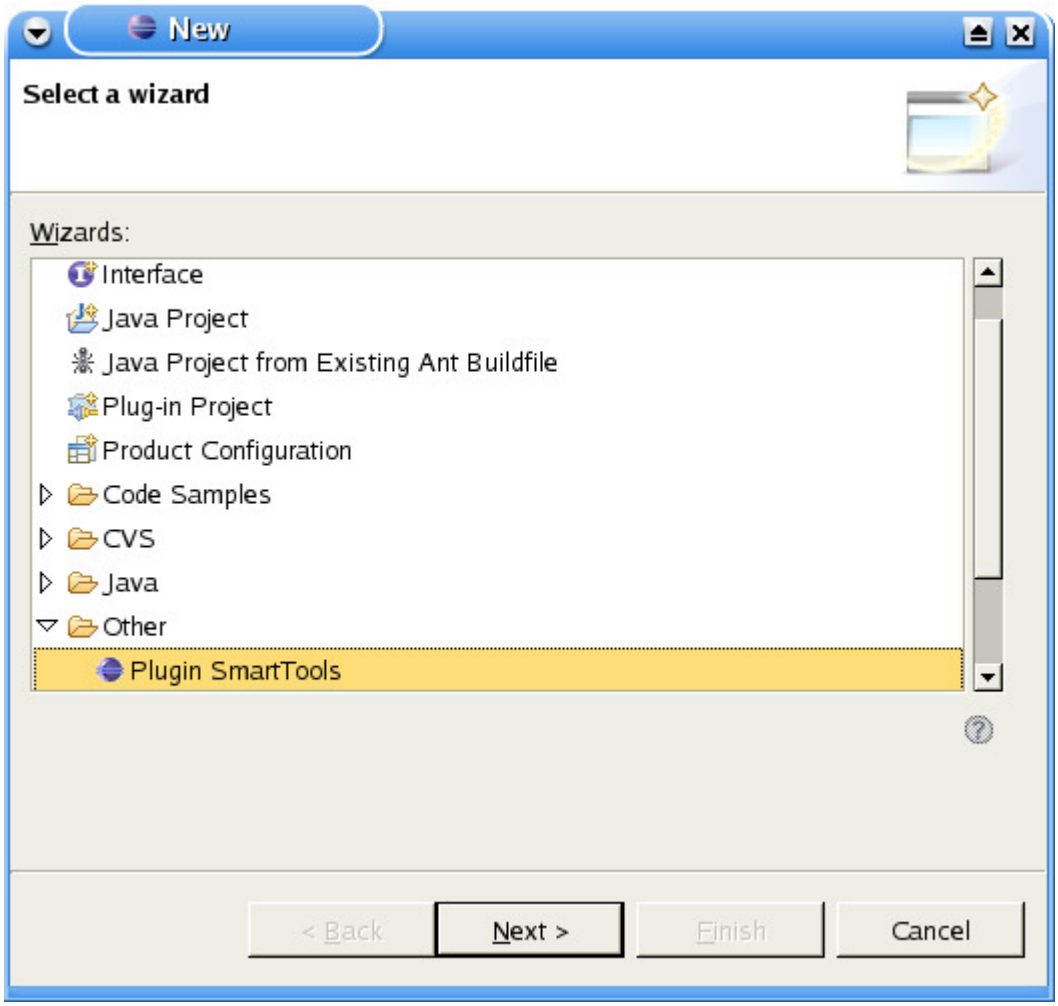
- Xml
  - XmlXml


Java - Test.cdml - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Test.cdml
GENERIC NAVIGATOR
Cdml Syntax
component
containerclass
facadeclass
formalism
parser
extention
lml
lml
behavior
dependance
dependance
input
input
input
input
input
input
input
Outline
CDML SYNTAX
Cdml Syntax
Component:cdml of type:document Extends: logicaldocu
Container class: CdmlContainer
Facadeclass: CdmlFacade Userclassname: Cdml
Formalism: cdml
Parser: cdml.parsers.CdmlXMLParser of Type: cdmlxml
Lml: DEFAULT in File: resources:lml/cdml-default.lml
Lml: SYNTAX in File: resources:lml/cdml-syntax.lml
Behavior: resources:behaviors/cdml-behaviors.bhv
Depend of: st-core Jar file: st-core.jar
Depend of: componentsgenerators Jar file: componer
Input:generateComponentDir (componentDir : java.lar
Input:generateComponentJar (componentDir : java.lan
Input:generateComponentHtml (componentDir : java.l
Input:createDirectoryStructure (name : java.lang.String
Input:generateComponent (componentDir : java.lang.
Input:compileUserSrc (componentDir : java.lang.String
Input:genCosynt (absyntName : java.lang.String, outpu
Writable
Insert
1 : 1

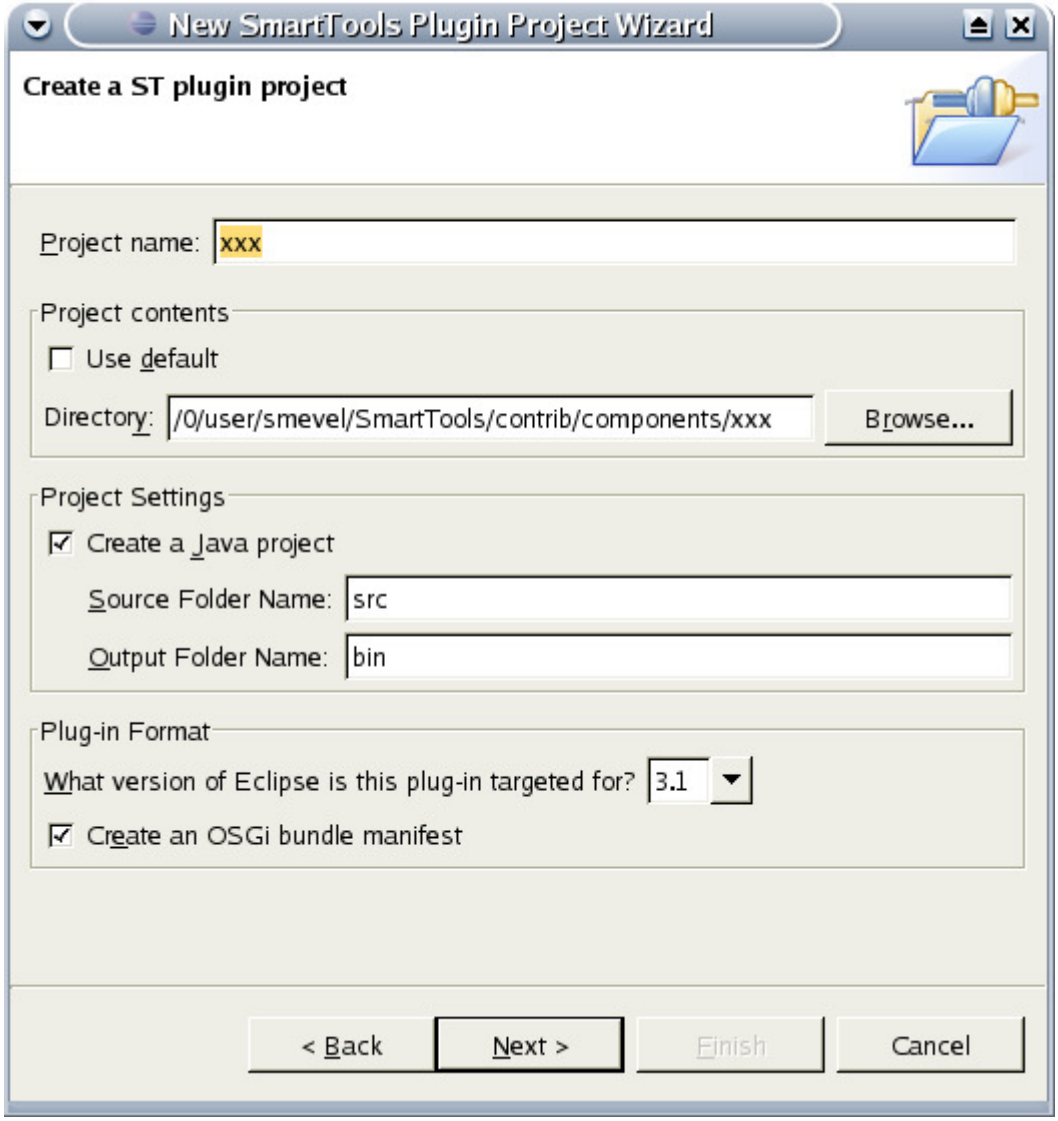
13. Extension du wizard PDE pour la fabrique SmartTools

Nous allons ici prendre pour exemple la création du composant xxx
REMARQUE : par convention, si les fichiers cibles ont pour extension xxx, alors nommer le projet xxx

- Créez un nouveau plugin (File -> New -> Other -> Plugin SmartTools)

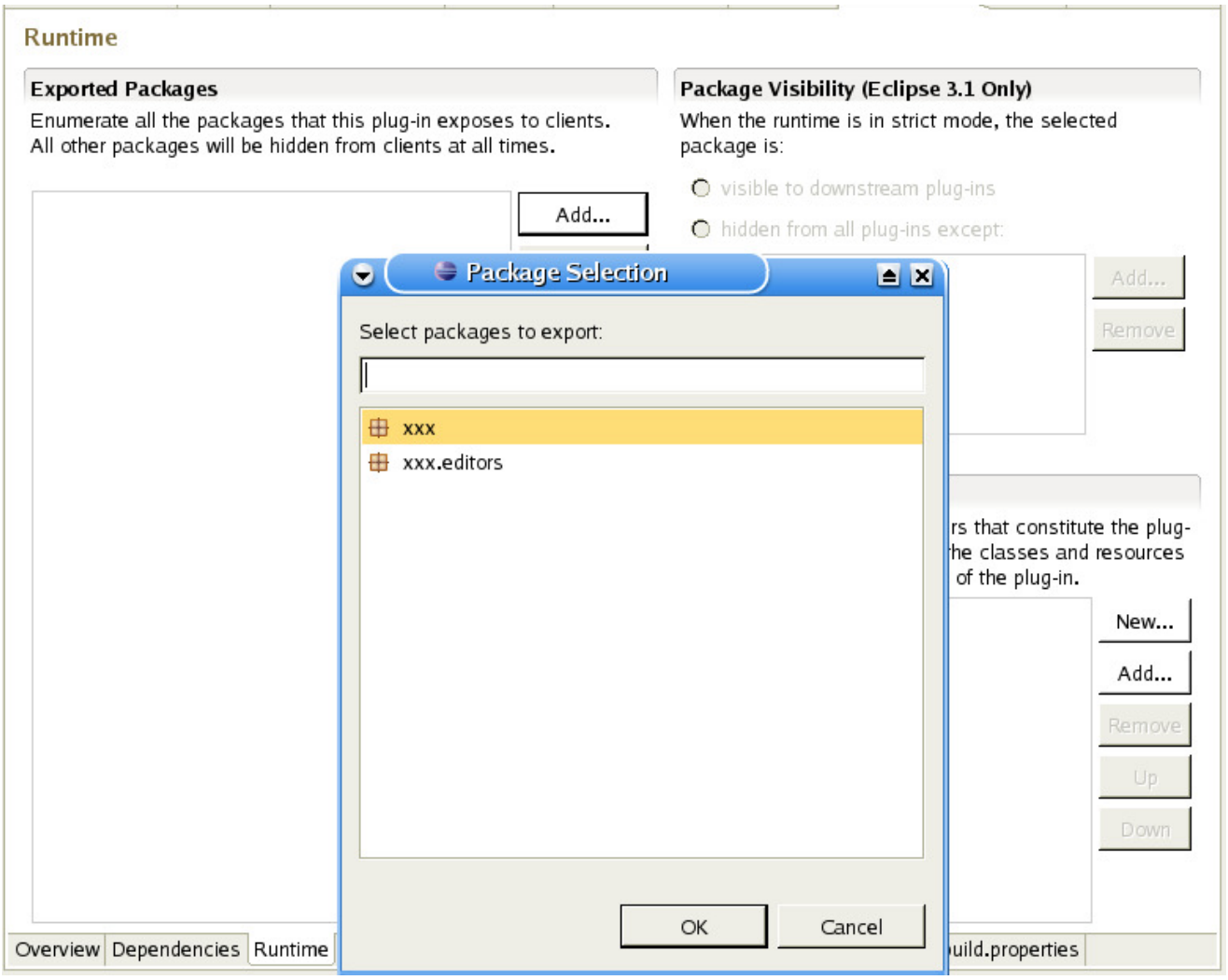


- Vous obtenez un nouveau Wizard qu'il faut remplir comme suit :



- Nom du projet : xxx
- Contenu du projet (Project contents) : /.../SmartTools/contrib/component/xxx
  ATTENTION : mettez le chemin complet en rajoutant le nom du projet xxx
- Laissez les autres options par défaut
- Cliquez sur le bouton Next, puis sur Finish

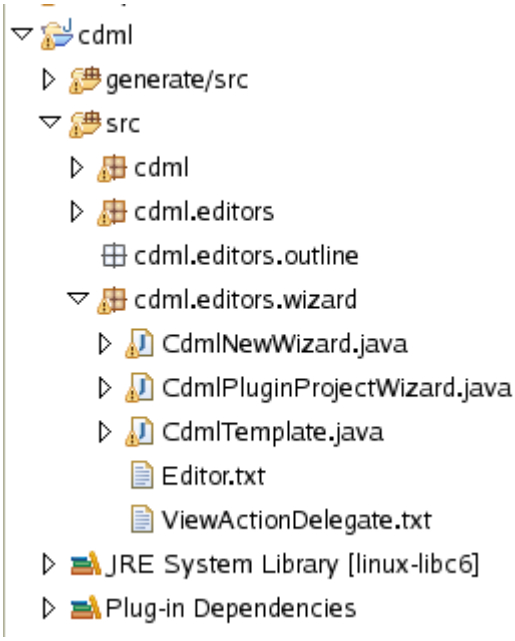
- Il faut exporter les packages du projet



- Sélectionnez l'onglet Runtime
- Cliquez sur le bouton Add pour ajouter les packages
  il faut tous les ajouter
- Sauvez le projet

Structure du package cdml.editors.wizard

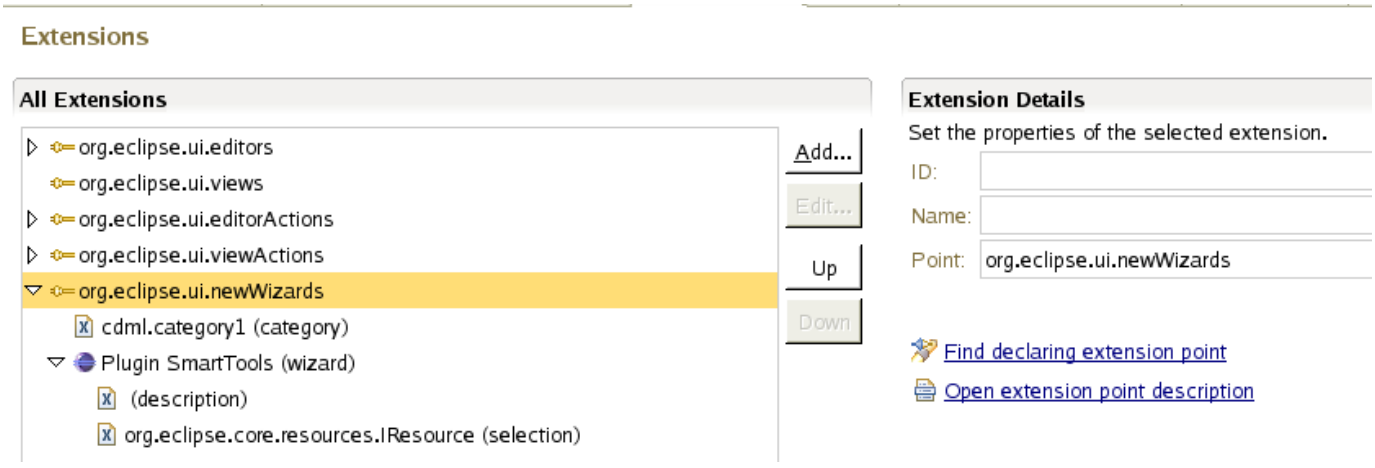
Ce package contenant les classes nécéssaires au fonctionnement du wizard



- Classe CdmlPluginProjectWizard permet de créer le wizard permettant de :
  - générer un composant SmartTools
  - importer les fichiers générés dans un projet Eclipse
  - de fabriquer un plugin à partir du composant
- Classe CdmlTemplate permet de modifier le fichier plugin.xml
- Classe CdmlNewWizard permet d'intégrer une instance de CdmlTemplate dans le wizard CdmlPluginProjectWizard
- Fichiers Editor.txt et ViewActionDelegate.txt, des fichiers génériques pour gérer l'éditeur

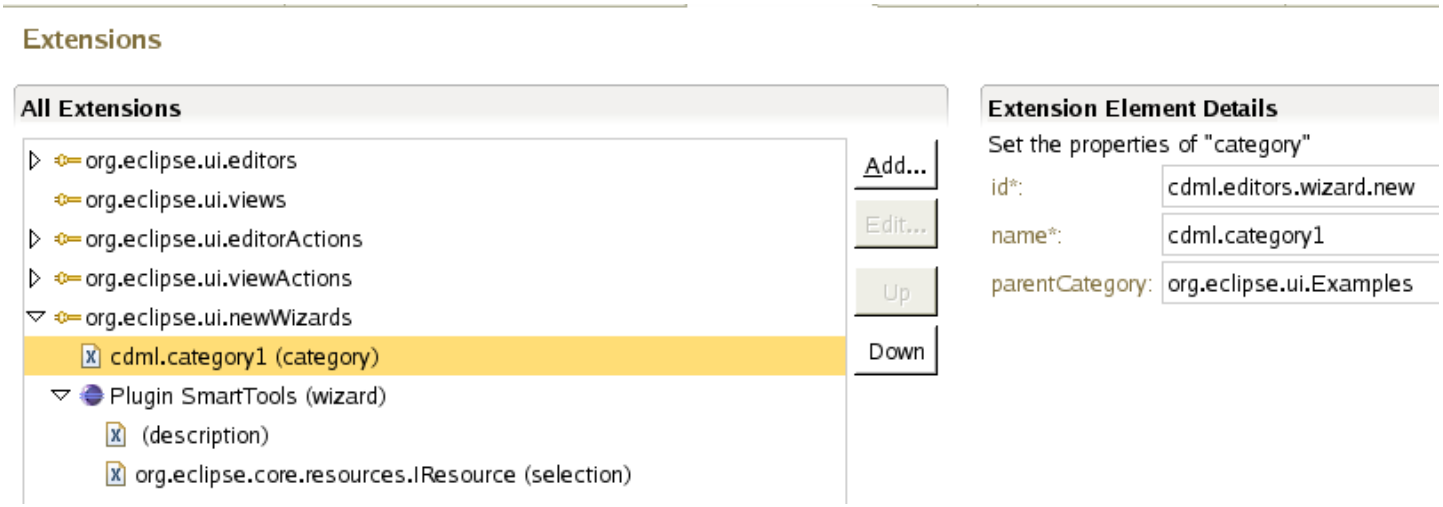
Intégration du wizard cdml.editors.wizard dans l´environnment Eclipse
On se place dans l'onglet extension du fichier plugin.xml

- 1ère étape : Ajout du point d extension org.eclipse.ui.newWizards en appuyant sur le bouton Add et en sélectionant le plugin org.eclipse.ui.newWizards dans la liste



- 2ème étape : Ajout d'une catégorie

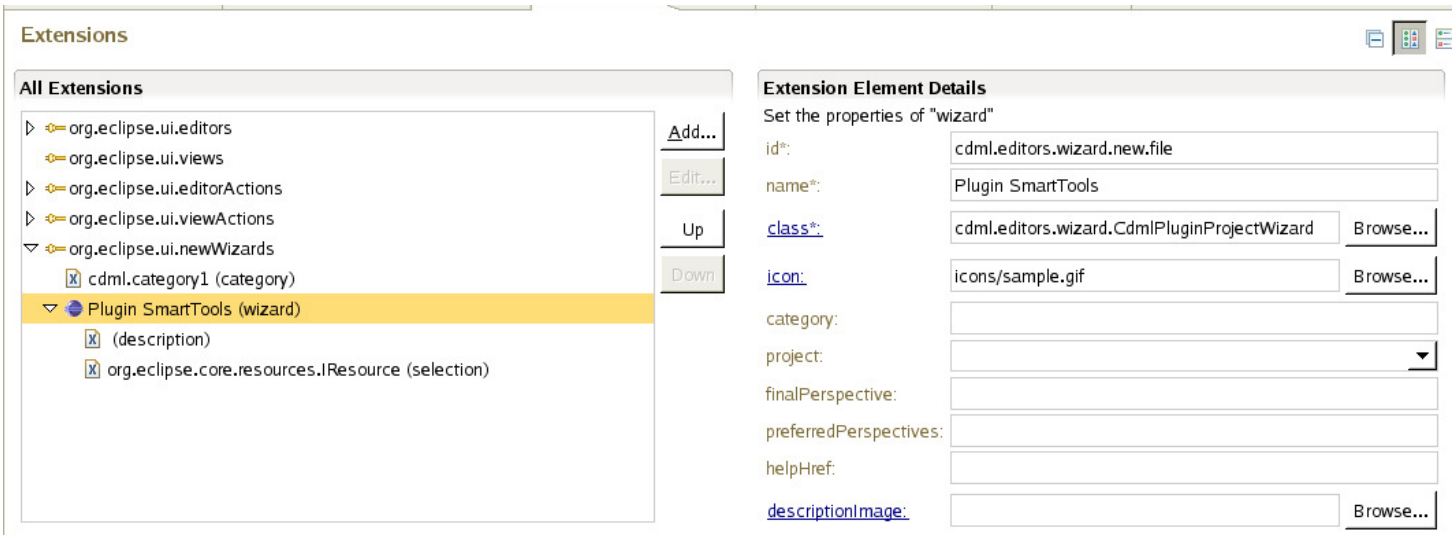
(clic droit sur org.eclipse.ui.newWizards -> category)



- 3ème étape : Ajout d'un wizard

(clic droit sur org.eclipse.ui.newWizards -> wizard)

ATTENTION : bien spécifier la classe qui va créer le wizard (ici cdml.editors.wizardCdmlPluginProjectWizard)



- 4ème étape : Ajout d'une description

(clic droit sur Plugin SmartTools (wizard) -> description)

- 5ème étape : Ajout d'une sélection

(clic droit sur Plugin SmartTools (wizard) -> selection)

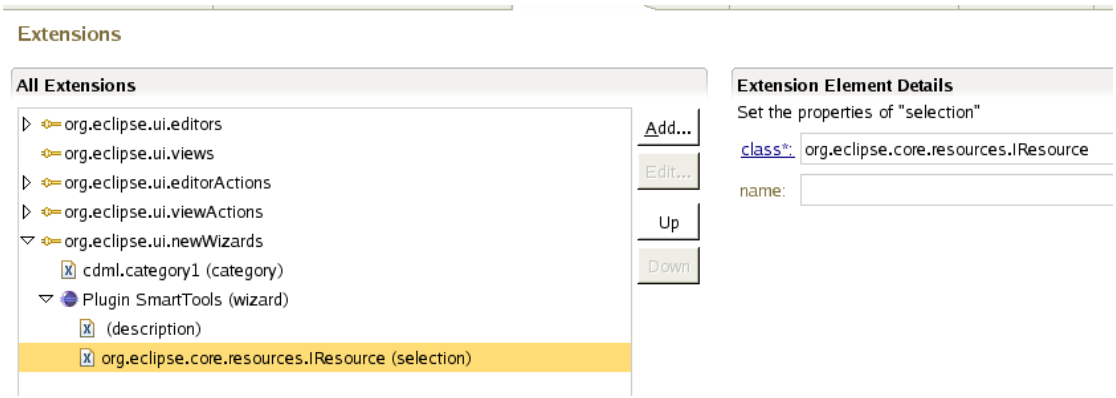


Schéma des appels des méthodes du wizard CdmlPluginProjectWizard

- L'utilisateur appuie sur le bouton Finish du wizard
- Appel de la méthode *performFinish()* de l'objet de la classe cdml.editors.wizard.CdmlPluginProjectWizard
- Création du composant SmartTools par la méthode *CreateComponent()*
  Le répertoire de ce composant contient un répertoire src
- Création du package xxx.editors sous le répertoire src
  - Création de la classe xxx.editors.Editor
  - Création de la classe xxx.editors.ViewActionContributor
- Création du projet et du plugin à l'aide d'une instance de la classe org.eclipse.pde.internal.ui.wizards.plugin.NewProjectCreationOperation
  - On fournit en paramètre une instance de la classe xxx.editors.CdmlNewWizard
  - Cette instance contient en paramètre une instance de la classe xxx.editors.CdmlTemplate
- Cette dernière instance va modifier le plugin.xml en ajoutant
  - des dépendances (dependancies)
  - des extensions

14. Petits Détails Utiles

Mise a jour de la outline :

Lorsque vous modifiez le texte dans l'éditeur et que vous recréez la outline, cette dernière doit de mettre a jour.
Pour les langages à syntaxe xml, le parser est rappelé et la outline se met a jour.
Pour certains composants comme Absynt et cosynt, il est nécesaire de recréer le parser lors de la mise a jour.
pour cela, nous avons rajouté la méthode suivante dans fr.smarttools.core.tree.parsers.AntlrParserAdapter :

```
public TypedDocument parse(Reader reader)
throws IOException, ParserConsistencyException {

	STproperties sTpro = new STproperties ();
	String eclipse = sTpro.getProperty("SmartTools.Eclipse");
	if (parser == null || eclipse.equals("yes")) {
		try {
			Class c = Class.forName(classname);
			methodParse = c.getMethod("parse", new Class[] {});
			Constructor constr = c.getDeclaredConstructor(new Class[] {Reader.class});
			parser = constr.newInstance(new Object[] { reader});
		} catch(Exception e) {
			e.printStackTrace();
			return null;
		}
	}
		try {
			doc = (TypedDocument) methodParse.invoke(parser, new Object[] {});
		} catch(Exception e) {
			e.printStackTrace();
		}
	return doc;
}
```

Mise à jour du plugin.xml:

Pour que Eclipse prenne en compte les modifications apportées dans le fichier plugin.xml,
faites un couper-coller de tout le fichier : *Ctrl a* , *Ctrl x* , *Ctrl v* et enfin *Ctrl s* .

Failure Chechsum lors d'un update:

Dans st-core, nous avons un lien symbolique de src/fr vers ../fr.
Le update s'effectue dans src/fr puis ../fr. Il y a alors une erreur "checksum update failure" car les classes ont déjà été mise a jour.