Une SOA pour les RCP Eclipse

Etudiants :

LOUIS Romain
VICARD Sébastien

Encadrant :

PARIGOT Didier

Vendredi 2 Mars 2007

Remerciements

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

Introduction

Notre projet « Une SOA pour les RCP 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*) nommé SmartTools et basé sur une approche par fabrique logicielle.

Ce dernier permet notamment d'intégrer ses composants en plugins sur l'IDE Eclipse.

Tout au long de ce projet, nous avons effectué des expériences préliminaires pour contribuer à intégrer la SOA SmartTools au sein d'Eclipse.

Dans un premier temps, nous allons vous présenter plus en détails notre sujet et nos objectifs.
Nous détaillerons ensuite le fonctionnement des plates-formes que nous avons utilisées ainsi que le problème posé et notre idée pour le résoudre.
Nous verrons enfin toutes les étapes de notre travail puis nous terminerons par une conclusion sur ces quatre mois de stage à l'INRIA.

Une annexe technique décrivant concrètement nos réalisations est également présente à la fin du rapport.

Table des matières

Remerciements 2
Introduction 3
Table des matières 4
Lexique 5
I. Présentation du sujet 6
1. Contexte du projet 6
2. Objectif 6
3. SmartTools 6
4. Outils utilisés 7
5. Vocabulaire 7
II. Analyse de l'existant 8
1. OSGi 8
2. SmartTools 8
3. Eclipse 9
4. OSGi et SmartTools 10
5. SmartTools et Eclipse, situation actuelle 11
III. Idée 12
IV. Organisation du projet 13
Etape 1 : Création d'un plugin Eclipse à partir de SmartTools 13
Etape 2 : Création d'une application RCP basique associée à SmartTools 14
Etape 3 : Utilisation des containers des composants SmartTools "à la main" 15
Etape 4 : Utilisation des containers des composants SmartTools "générés" 16
Etape 5 : Adaptation aux documents, ou composants dits "logiques" 18
Etape 6 : Elagage et exportation de l'application Rcp_Basic 19
Etape 7 : Mise en oeuvre des composants logiques sur Rcp_Db 20
Etape 8 : Mise en place de l'architecture complète 21
Conclusion 22
Table des illustrations 23
Annexes 24

Lexique

- **Cdml** : Langage décrivant un composant
- **DSL** : Domain Specific Language, petit langage
- **GUI** : Graphical User Interface
- **IDE** : Integrated Development Environment
- **Lml** : Langage basé sur XML décrivant la structure de l'interface graphique
- **MDA** : Model Driven Architecture
- **OMG** : Object Management Group
- **OSGi** : Open Services Gateway Initiative
- **RCP** : Rich Client Platform
- **SOA** : Service Oriented Architecture
- **XML** : eXtensible Markup Language
- **Svn** : Subversion, gestionnaire de versions de fichiers utilisant une base de données
- **W3C** : World Wide Web Consortium

I. Présentation du sujet

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 proposés pour répondre à ces nouveaux défis. Cela s'est accompagné de l'émergence d'une multitude de technologies logicielles, proposées et soutenues par des « consortia » internationaux de standardisation (W3C, OMG, etc...) ou des fondations/alliances (Apache, Eclipse, OSGA, etc...) ou des grands groupes industriels (MicroSoft, IBM, SUN, BEA, etc...).

Ces différents concepts peuvent être unifiés et regroupés dans une notion nouvelle de fabrique logicielle, concrétisée par le logiciel SmartTools créé 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 DSL (Domain Specific Language), IBM avec l'environnement de programmation Eclipse et ses évolutions vers la notion de RCP (Rich Client Platform) et enfin SUN avec l'environnement NetBeans.

2. Objectif

Notre objectif est d'instancier une approche orientée services de SmartTools au dessus d'Eclipse. Avant de préciser concrètement cet objectif et en particulier l'approche orientée services de SmartTools, nous devons présenter rapidement les concepts de base en terme de composants de SmartTools, du consortium OSGi (les bundles), et enfin de l'IDE Eclipse (les plugins).

3. SmartTools

À l'heure actuelle, SmartTools est **composé d'une dizaine de petits langages appelés DSL 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 ce dernier (structures de données, parsers, 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 DSL)
- 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

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 SVN. Nous avons donc disposé d'un serveur SVN afin d'effectuer un travail collaboratif efficace.

5. Vocabulaire

Dans ce rapport, nous parlerons de ***bundles OSGi***, de ***composants SmartTools*** et de ***plugins Eclipse*** afin de respecter les normes d'appellation de chacun.

II. Analyse de l'existant

1. OSGi

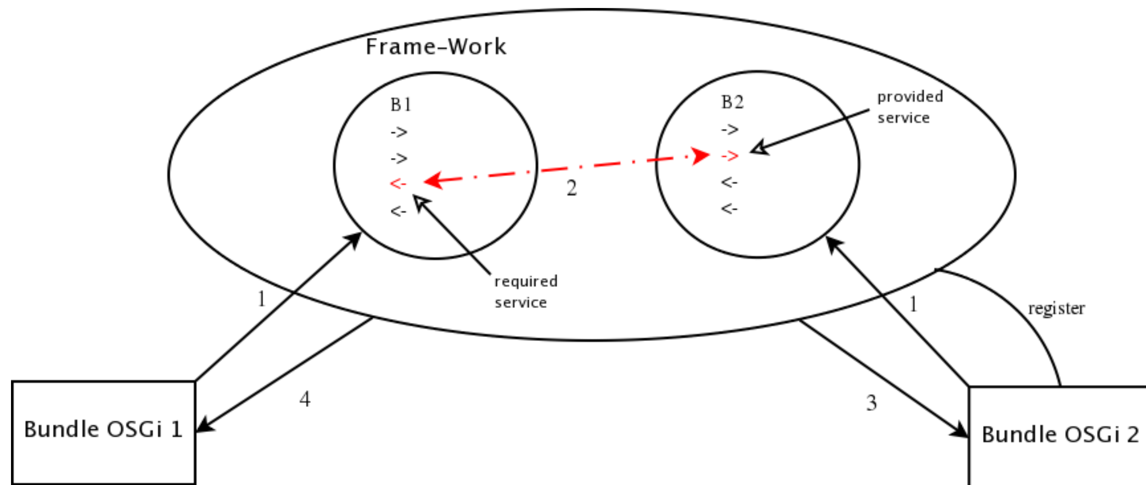

Figure 2.1 : Communication sous OSGi

Sous OSGi, il existe un framework qui joue, en particulier, le rôle d'annuaire de services. Comme le montre la figure 2.1, lorsqu'un bundle est installé, il doit enregistrer (déclarer) au framework ses services fournis et requis (**1**). Ces derniers sont décrits à l'aide d'interfaces Java.
Le framework s'occupe de la communication entre les services des bundles, par l'intermédiaire de messages de notification.
En effet, pour pouvoir communiquer entre eux, les bundles OSGi sont capables de fournir des services. Ainsi, on appellera « service fourni » (**->**) un service fourni par le bundle et destiné à être utilisé par les autres bundles, et « service requis » (**<-**) un service que le bundle demande à l'extérieur.
Le framework, quant à lui, établira la communication entre deux bundles en fonction des services requis par l'un et fournis par l'autre. Pour cela, il identifiera d'abord les présences respectives des deux services (**2**). Il ira ensuite chercher le service fourni via la récupération du contexte (**3**). Enfin, il préviendra le bundle qui requiert le service de la présence de ce dernier (**4**).

Pour conclure, un bundle fournit des services au framework susceptibles d'être utilisés par n'importe quel autre bundle rattaché à ce framework.

2. SmartTools

Tout comme sous OSGi, il y a sous SmartTools la présence d'un framework appelé **Components Manager** (CM) comme le montre la figure 2.2. Ce dernier sert à initialiser les composants, ainsi qu'à établir une communication entre deux composants.
Pour cela, chaque composant possède des services à fournir et requiert d'autres services en retour.
Le premier composant signalera au CM son désir de communiquer avec un second.

Le CM récupère donc les spécifications de chacun des deux composants et établit la communication entre eux en fonction des services requis et fournis de chacun.
Dès que le code métier est modifié, le container est informé et informe tous les autres containers communiquant avec lui par une procédure suivant le principe du multicast.

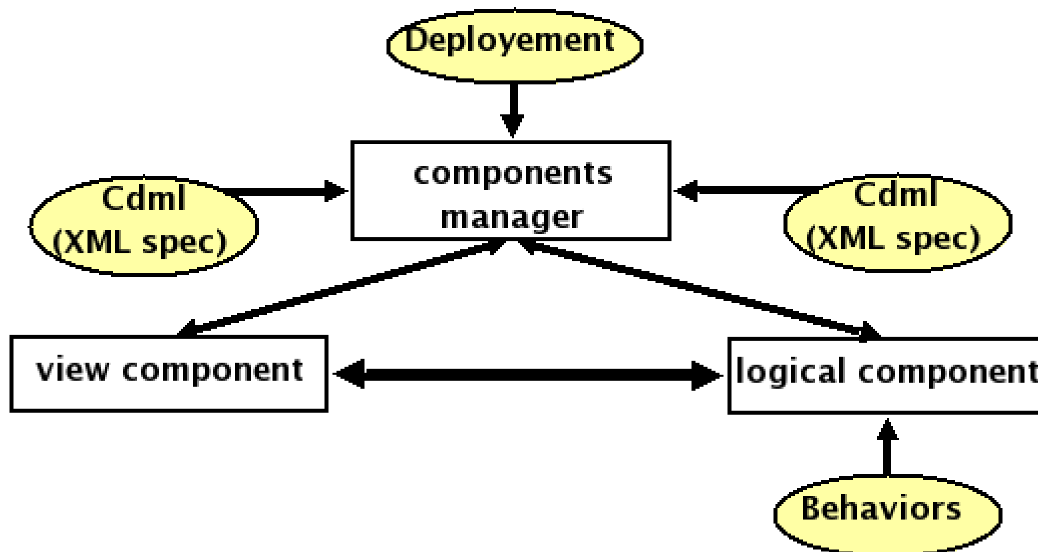

Figure 2.2 : Instanciation et communication sous SmartTools

3. Eclipse

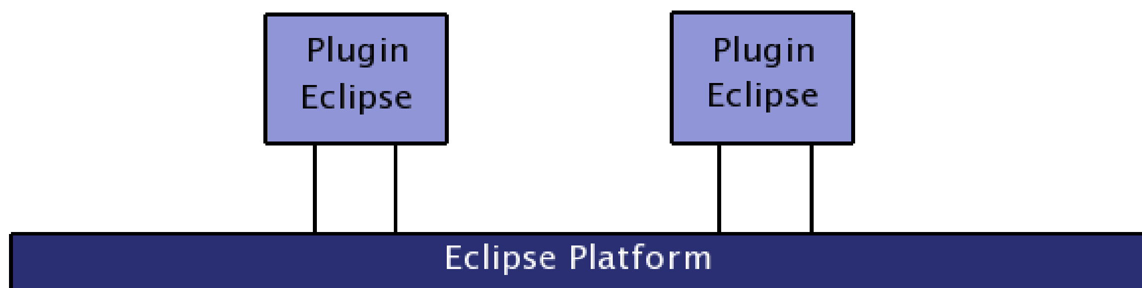

Figure 2.3 : Les plugins sous Eclipse

Sous Eclipse, l'interaction entre les plugins et la plate-forme se fait par des points d'extension. Même si la plate-forme Eclipse utilise le framework OSGi (depuis la version 3 seulement), la notion de point d'extension a été introduite (avant) pour mettre en oeuvre un mécanisme d'enrichissement de la plate-forme. Mais ce mécanisme n'établit pas explicitement de communication (ceci reste au niveau de la programmation et se fait par appel de méthodes codées en Java).

4. OSGi et SmartTools

SmartTools a été instancié « au dessus » de OSGi. La version actuelle utilise les fonctionnalités de base de OSGi (start, stop et install) et enrichit le framework avec les capacités du CM de SmartTools (service spécifié dans des descriptifs neutres).

La première différence entre les communications OSGi et SmartTools est l'établissement de cette communication.
En effet, sous OSGi, lorsqu'un bundle est déclaré, ce dernier renseigne le framework en lui donnant les services qu'il fournit et ceux qu'il attend. Ainsi, le framework agit de la même façon qu'un annuaire en listant en fonction des bundles actifs tous les services requis et fournis par chacun. Ainsi, dès qu'un nouveau bundle est susceptible de fournir les services requis par le premier, le framework établit la communication entre les deux bundles via ce service.
Pour résumer, un bundle précise ce par quoi il veut communiquer, mais pas avec qui.

C'est là que se fait la différence avec SmartTools. En effet, ce dernier se base sur le principe d'une communication pair à pair. C'est-à-dire que le composant ne dit pas comment il veut communiquer, mais avec qui. Cette information est envoyée au CM, qui récupère de son côté les spécifications du composant demandeur, celles du composant demandé, et il établit les communications sur tous les services fournis et requis compatibles entre les deux composants.

La seconde différence concerne l'implication du programmeur dans l'établissement de cette communication.
On a vu que la communication OSGi était établie selon les besoins du bundle. Le problème est que les services sont décrits à l'avance dans les interfaces Java des deux bundles. Ceci reste donc la tâche du programmeur. Cela signifie que ce dernier doit connaître à l'avance quel type de message il s'attend à recevoir et que les classes communicantes doivent porter le même nom.

À l'inverse, sous SmartTools, la communication est pair à pair. Cela signifie que le CM établit la communication entre deux composants précis selon une description neutre (fichier cdml) de chacun. L'avantage de cette méthode est le dynamisme qu'elle permet. En effet, toutes les communications entre services sont établies. Si un service n'a pas son équivalent « en face », alors la communication restera « floue ».
De même, le programmeur a juste besoin de noter dans les spécifications les actions dont il a besoin, et c'est le moteur qui s'occupera de les lier et de générer le code correspondant.

5. SmartTools, Eclipse et situation actuelle

Comme le montre la figure 2.4, notre but est d'instancier une approche orientée services de SmartTools au dessus d'Eclipse. Ainsi, nous nous servirons d'Eclipse et de sa GUI tout comme nous nous servons déjà de la GUI propre de SmartTools. De plus, nous n'aurons pas à nous soucier de la communication entre un plugin Eclipse et la plate-forme.

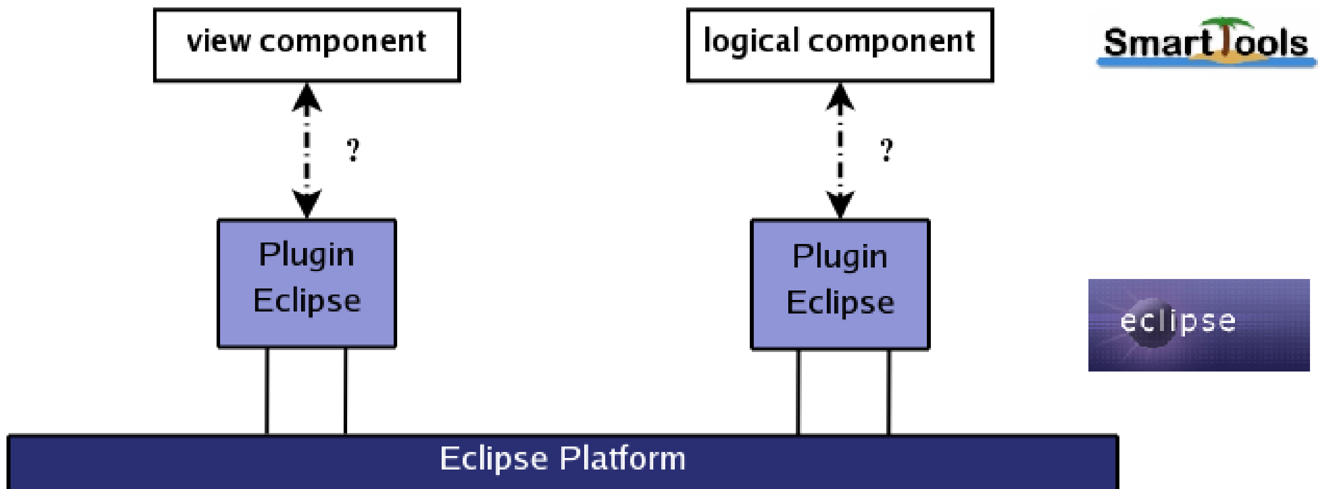

Figure 2.4 : Problème de communication actuel

À l'heure actuelle, l'équipe a déjà effectué une transformation des composants SmartTools en plugins Eclipse, mais dans ce contexte la communication entre les composants SmartTools et leur vue Eclipse correspondante n'existe pas. En effet, seule la partie métier des composants a été intégrée en oubliant totalement les aspects de communication (la SOA SmartTools).

Nous devrons donc dans un premier temps tenter d'établir une communication entre ces derniers en se basant sur la partie des composants qui leur permet déjà de communiquer entre eux.

III. Idée

L'objectif du projet est de trouver le moyen de faire communiquer le monde SmartTools avec le monde Eclipse. Plus précisément, trouver un moyen pour qu'un composant SmartTools puisse à travers son container communiquer avec la plate forme Eclipse. Par exemple, pour un composant graphique de SmartTools, on souhaiterait que le résultat de son évolution soit visible dans la GUI d'Eclipse. Pour cela, il est nécessaire de définir l'objet Eclipse (une vue Eclipse) qui sera instancié à la manière Eclipse (mécanisme de points d'extension) et sera le support pour visualiser le résultat.

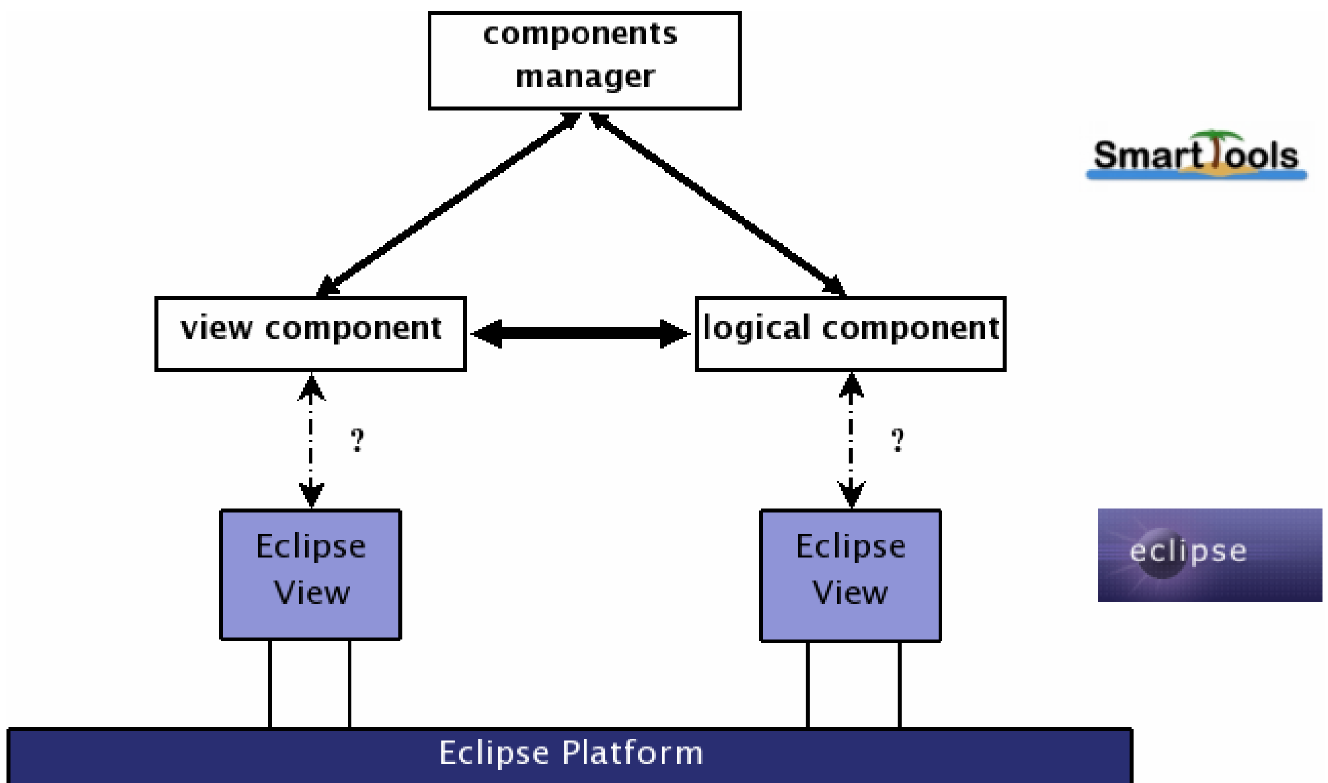

Figure 3.1 : Architecture de communication proposée

L'idée de base est donc de considérer que chaque type de composant SmartTools sera statiquement associé à une vue Eclipse. Cette association sera naturelle, et définie par l'activeur associé à ce type de composant. Ce dernier est un service OSGi fourni à notre « components manager » et son rôle est de gérer le cycle de vie du composant. Pour ce faire, le CM sera préalablement lancé, ainsi lors de la création d'un composant SmartTools par l'activeur, il faudra d'une part trouver la vue Eclipse associée à ce type de composant et puis établir une communication dans les deux sens pour l'échange d'informations comme le montre la figure 3.1.

IV. Organisation du projet

Nous allons vous décrire dans ce chapitre la démarche que nous avons suivi pour l'élaboration de notre application.

Dans un premier temps, nous avons fait le choix de travailler sur une RCP Eclipse pour sa facilité de mise en oeuvre. En effet, la plate forme Eclipse permet des développements d'application dans ce sens, avec une organisation très facile à gérer. Nous avons donc commencé à approcher les RCP par une démonstration fournie par Eclipse. A partir de cela, nous avons introduit nos premiers composants SmartTools.
Nous avons donc d'abord commencé par instancier deux composants simples à la main, c'est-à-dire dans le code en principe généré par SmartTools. Nous avons ensuite effectué des ajouts de façon à ce qu'ils soient générés pour tout composant. Puis nous avons suivi le même ordre d'idée pour les composants logiques. Enfin, nous avons étendu cela aux composants graphiques, dernier type de composant non traité. Néanmoins, ces derniers n'étant pas générés, toutes nos modifications ont été faites à la main, dans le noyau de SmartTools.

Etape 1 : Création d'un plugin Eclipse à partir de SmartTools

Motivation

Cette étape nous permet de créer un plugin à partir de SmartTools afin d’analyser comment il est généré.
L’intérêt ici est de mieux comprendre non seulement le fonctionnement de SmartTools mais aussi la tranformation du composant en plugin.

Démarche

À l'aide des annexes techniques du projet "Transformation de composants SmartTools en plugins Eclipse" de l'année dernière, nous avons pu créer à partir du composant SmartTools (Tiny) le plugin correspondant.
Pour se faire, nous avons ajouté et paramétré les fichiers suivants :

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

Etape 2 : Création d'une application RCP basique associée à SmartTools

Motivation

L'objectif ici est d'utiliser la technologie SmartTools, afin d'intégrer deux composants (l'un jouant le rôle d'éditeur et l'autre de vue) et leur vue dans une application RCP basique.
Les vues sont établies par copie simple (aucune utilisation des containers des composants). La communication entre les deux composants se faisant via SmartTools avec leur container.
L'intérêt ici est de mieux comprendre la communication entre composants grâce à SmartTools, et d'étudier les liens avec les vues.

Démarche

À l'aide des tutoriels **http://www.eclipse.org/articles/Article-RCP-3/tutorial3.html** (Site Eclipse) et **http://wiki.eclipse.org/index.php/Rich_Client_Platform#Tutorials**, nous avons créé une application RCP basique.
Nous avons effectué ensuite les opérations suivantes :

- Ajout de deux vues (*Cmp1View.java* et *Cmp2View.java*) que nous avons intégrées à la perspective (*Perspective.java*)
- Ajout des paramètres des vues dans *plugin.xml*
- Ajout et paramétrage du fichier *Rcp_Basic.product*
- Ajout et paramétrage du fichier *config.ini*
- Association de deux composants (*cmp1* et *cmp2*) aux deux vues par leurs fichiers d'application (*Cmp1App.java* et *Cmp2App.java*)

Etape 3 : Utilisation des containers des composants SmartTools "à la main"

Motivation

On souhaite maintenant ajouter un système de communication entre les différents éléments de notre application RCP, c'est-à-dire nos vues Eclipse et nos composants SmartTools. On suppose que chaque type de composant SmartTools connaît le type de vue à laquelle il sera associé.

Démarche

Dans un premier temps, il nous faut instancier le composant (sa partie métier appelée **façade** et son **container**) et la **vue** Eclipse correspondante qui sera enregistrée parmi d'autres vues. Pour cela, le CM va appeler l'activeur pour instancier le composant. L'activeur fera ensuite une recherche de la vue pour l'instancier. Il nous faudra enfin établir la communication entre elle et le container du composant.

Le problème se sépare donc en deux points :

- la recherche de la **vue** Eclipse correspondant au **container**
- la communication entre la **façade** et la **vue** (pour la mise à jour de la **vue**) :
  - la communication entre la **façade** et le **container**
  - la communication entre le **container** et la **vue** Eclipse du composant dans l'application

Pour cela, nous avons mis en place cette communication "à la main".
En effet, nous avons modifié directement les classes en principe automatiquement générées par SmartTools (les containers et les façades).
La communication que nous souhaitions obtenir se devait de pouvoir ressembler à un dialogue, c'est-à-dire devait pouvoir se faire "dans les deux sens". De plus, pour respecter une indépendance entre SmartTools et Eclipse, nos containers ne doivent pas connaître, par exemple, les détails d'implentation des vues Eclipse. Pour cela, nous avons choisi deux techniques les relations suivantes également présentes sur la figure 4.1 :

- la **vue** référence un **container** pour lui parler et l'écoute grâce à un listener
- le **container** référence la **façade** pour lui parler et l'écoute grâce à un listener

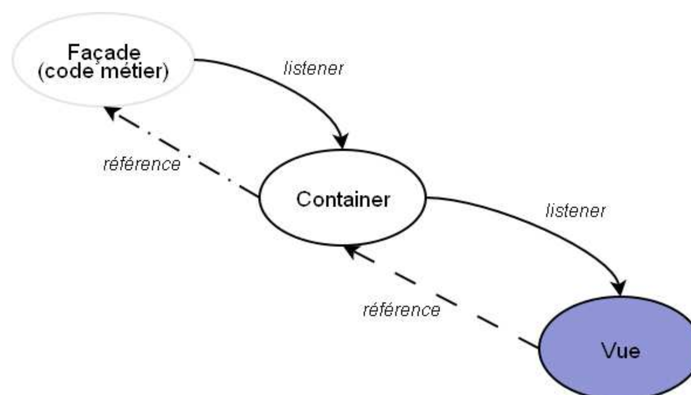

Figure 4.1 : Communication entre un composant et sa vue

Etape 4 : Utilisation des containers des composants SmartTools "générés"

Motivation

On souhaite maintenant faire en sorte que cette amélioration marche dans le cadre de tous les composants simples de SmartTools. Il nous faut donc modifier les fichiers qui génèrent ceux que nous avons adaptés dans l'étape 3 et surtout sans faire de modifications dans le code généré par SmartTools.

Démarche

Nous avions donc à mettre en place les deux communications. Pour cela, nous avons dû procéder de deux façons différentes. En effet, nous savions que les classes des listeners devaient se trouver dans les packages des membres écoutés.

Nous avions ainsi le listener entre :

- la **façade** et le **container** dans le package de la façade
- le **container** et la **vue** Eclipse dans le package du container

Dans le premier cas, nous avons fait en sorte que ces listeners soient générés automatiquement par le générateur de composants de SmartTools (Components Generator). Pour cela, nous avons rajouté le service (updateComponent) dans le descriptif du composant (fichier cdml) :

```
<output name="updateComponent" method="updateComponent" doc="update Component">
     <arg name="message" doc="" javatype="javax.Swing.JPanel"/>
</output>
```

Dans le second cas, nous avons défini ce listener dans le coeur de SmartTools, c'est-à-dire dans la classe de base des containers (AbstractContainer du composant noyau de SmartTools nommé st_core). Pour les vues Eclipse, tout a été défini dans une classe commune que toutes les vues doivent étendre.

De plus, tous les changements apportés dans le code généré ont été répercutés vers les classes qui servent de base à la génération.

La figure 4.2 montre plus en détails le processus global.

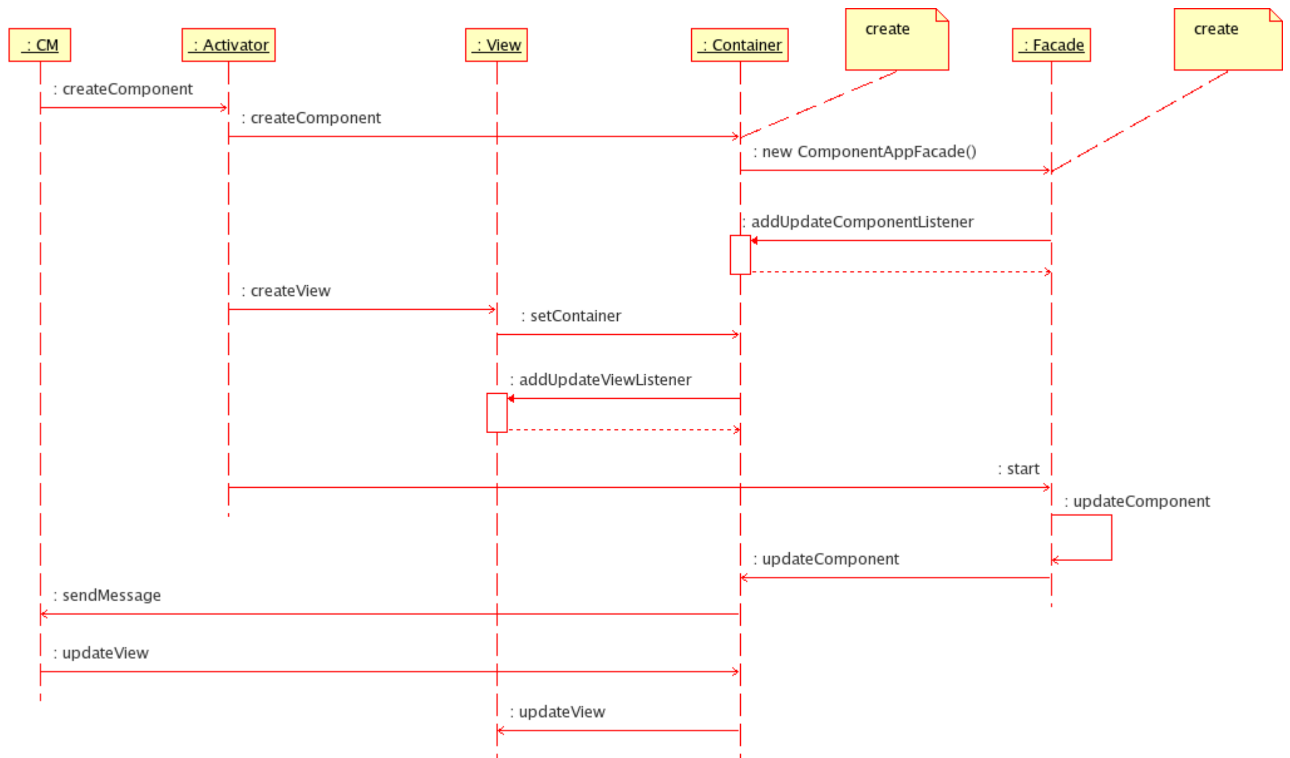

Figure 4.2 : Processus de communication

Etape 5 : Adaptation aux documents, ou composants dits "logiques"

Motivation

Parmi les composants de SmartTools, il existe des "documents", ou composants dits "logiques". Après avoir testé nos améliorations sur des composants génériques et simplifiés, nous allons tenter de porter ce mécanisme à ces types de composants.

Démarche

Deux solutions s'offraient à nous :

- soit nous modifions seulement le cdml (comme jusqu'à présent)
- soit nous travaillions sur le code lui-même

Nous avons dans un premier temps choisi de vérifier le fonctionnement de nos implémentations en passant par le fichier cdml. Nous avons donc suivi les étapes décrites plus haut pour le rajout de code et nous avons spécifié le fichier cdml de la même façon que pour un composant simple.

Cependant, un petit ajout de code au coeur de SmartTools a été nécessaire, afin de récupérer la façade du composant, chose qui n'était pas possible avant dans le cadre d'un "document".

Etape 6 : Elagage et exportation de l'application Rcp_Basic

Motivation

Une RCP doit pouvoir être transportable et lançable n'importe où sur l'ordinateur. Le nettoyage de Rcp_Basic était donc nécessaire afin d'alléger et de transformer notre application Rcp_Basic de test en application Eclipse indépendante visible sur la figure 4.3.

Démarche

La première étape consiste à élaguer l'application en supprimant tous les fichiers récupérés initialement par le tutoriel et n'ayant plus aucune utilité.

La phase d'exportation se déroule ensuite de la manière suivante :

- Exportation de l'application par le lien "Eclipse Product export wizard" du fichier *Rcp_Basic.product*
  Les dossiers *configuration* et *plugins* sont créés ainsi que les fichiers suivants :
  - eclipse
  - eclipse.ini
  - startup.jar
- Ajout du fichier *st.xml* à la racine du répertoire d'exportation
- Modification du fichier *eclipse.ini* à partir des arguments de lancement de l'application présents dans le fichier *Rcp_Basic.product*

Remarques :

- Le lancement de cette application RCP indépendante se fait par le shell grâce à la commande "./eclipse".
- Certains des plugins exportés dans le dossier du même nom peuvent présenter certaines erreurs. Dans ce cas, il faut les exporter manuellement par le lien "Export" du menu contextuel du plugin dans Eclipse. Sélectionner ensuite "Plug-in Development" puis "Deployable plug-ins and fragments" pour spécifier le type d'exportation.

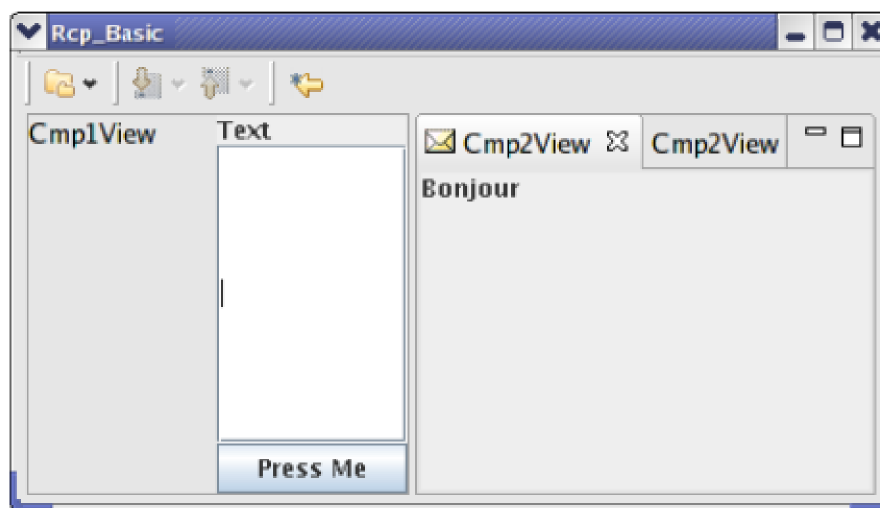

Figure 4.3 : Rcp_Basic

Etape 7 : Mise en oeuvre des composants logiques sur Rcp_Db

Motivation

Rcp_Db est une application destinée à la communication avec une base de données. L'intérêt de cette application est de mettre en pratique nos modifications sur le composant logique stUsersDb qui permet à l'utilisateur d'intéragir visuellement avec la base de données où le composant db traite la partie JDBC.

Démarche

Dans un premier temps, nous nous sommes basés sur l'application Rcp_Basic. Nous avons donc :

- adapté les vues et la perspective
- modifié le fichier st.xml (pour les liens entre composants)
- adapté les plugins des composants pour qu'ils connaissent leur vue

Il nous a ensuite fallu regénérer le composant, de sorte à qu'il soit conforme aux modifications effectuées jusqu'à l'étape 6.

Le challenge de cette application a été la communication entre un composant simple **db**, et un composant logique, **stUsersDb**. Une fois que les fichiers sont bien configurés, l'application se lance correctement. Les trois champs de texte et les trois boutons correspondant (init, launch query, launch update) présents sur la figure 4.4 et qui composent la partie graphique de StUsersDb s'affichent dans la perspective. L'inconvénient est que le résultat de nos requêtes SQL s'affiche en console or nous souhaiterions le faire afficher dans un éditeur au sein de notre application RCP.

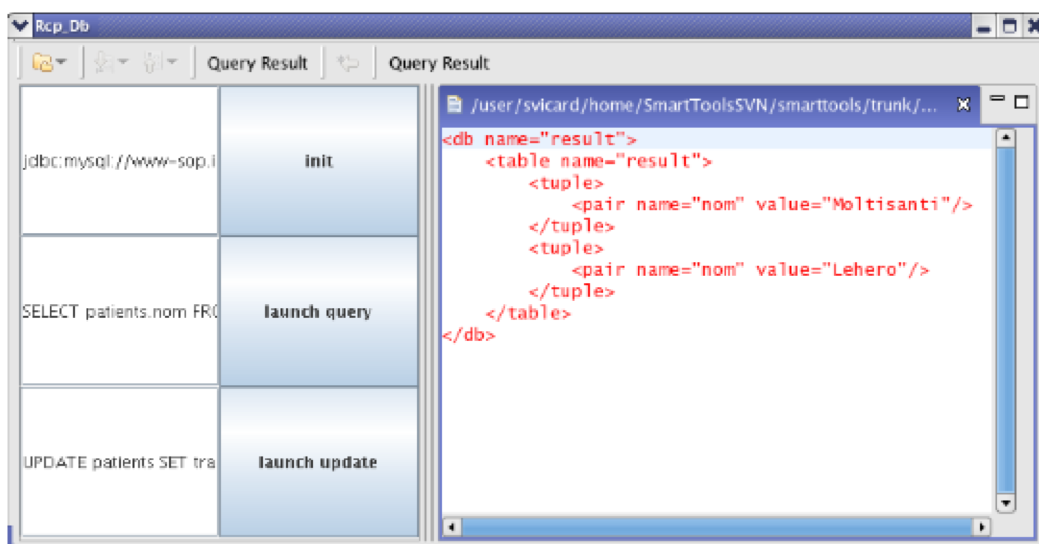

Figure 4.4 : Rcp_Db

Etape 8 : Mise en place de l'architecture complète

Motivation

Rcp_Db nous permet à l'heure actuelle de communiquer simplement avec une base de données à travers les boutons de la vue Eclipse. Cependant, nous n'avons aucune possibilité d'agir sur notre components manager pour, par exemple, créer une nouvelle instance d'un composant pour gérer l'ouverture d'un nouveau type de fichier.
Plus précisément, lors d'une action effectuée dans l'interface Eclipse (par exemple, une ouverture de fichier), il faut communiquer au CM ce message (ici, "ouvrir le composant associé"). Pour cela, la manière la plus simple est de rendre active en terme de services l'interface SmartTools (appelée Glayout) qui communique avec le CM.
Ainsi lors d'une action dans le monde Eclipse (un fichier ouvert) qui a besoin d'être transmise à un composant, il faudra passer par l'object Eclipse associé au composant Glayout.

Démarche

Pour cela, il fallait que le composant Glayout soit actif pour la RCP. On a donc utilisé la même méthode que pour SmartTools (hors d'Eclipse), c'est-à-dire mettre une instruction de connection entre le composant Glayout et le CM. Il fallait également associer au composant Glayout une vue Eclipse qui allait jouer le rôle de passerelle entre le monde Eclipse (interaction dans la GUI d'Eclipse) et le monde SmartTools (en particulier le composant Glayout).

Plus précisément lors de l'ouverture d'un éditeur pour un fichier de type X, il faudra émettre le message équivalent au bouton "openfile" de l'interface SmartTools au composant Glayout.
Ce dernier va déclencher un ensemble d'actions (par des messages au CM) qui va aboutir à la création d'une instance d'un composant X qui recherchera ensuite la vue Eclipse qui, dans ce cas, est l'éditeur pour X.

Conclusion

Dans le cadre de ce projet de fin d'études à l'INRIA, au sein de l'équipe SmartTools, une mission précise nous a été définie, à savoir établir une communication entre le monde SOA de SmartTools et la plate-forme Eclipse. C'est dans la continuité d'un projet réalisé dans le même cadre l'année dernière consistant à porter les composants SmartTools en plugins que nous avons travaillé.

En établissant ces communications, nous avons donc réussi à développer un modèle standard d'application RCP dans Eclipse se basant sur une architecture orientée services.

Ce stage nous a également apporté de nombreuses connaissances concernant notamment OSGi, les SOA, la programmation par composants et les plugins Eclipse dont nous avions seulement entendu parler. De plus, le fait de participer au développement d'un programme tel que SmartTools au sein de l'INRIA nous a offert une expérience très intéressante quant au fait de prendre part à un projet ayant débuté 4 ans auparavant et de s'adapter rapidement à une architecture existante.

Pour finir, notre solution se trouvera dans la prochaine version de SmartTools sous Eclipse. La suite du travail sera d'approfondir la communication entre les mondes SmartTools, Eclipse (les points d'extension), et le monde OSGi (les services de base).

Table des illustrations

Communication sous OSGi 8
Instanciation et communication sous SmartTools 9
Les plugins sous Eclipse 9
Problème de communication actuel 11
Architecture de communication proposée 12
Communication entre un composant et sa vue 15
Processus de communication 17
Rcp_Basic 19
Rcp_Db 20

Annexes

Lexique

Gforge : Système de gestion de développement collaboratif de logiciels via une interface web utilisant Svn pour la gestion des versions de projet. Le projet SmartTools est installé sur le site Gforge de l'INRIA (**http://gforge.inria.fr/projects/smartTool**).

Utilisation de Svn

Copier un projet à partir d'une base Svn

Créer un répertoire et y ouvrir un shell.
Exécuter la commande « svn checkout *x* » où x est l'adresse du projet sur la base Svn.

Afficher le contenu d'un répertoire d'un projet sur une base Svn

Se placer dans le répertoire du projet désiré et y ouvrir un shell.
Exécuter la commande « svn ls ».

Mettre à jour un projet à partir d'une base Svn

Se placer dans le répertoire du projet désiré et y ouvrir un shell.
Exécuter la commande « svn update ».

Mettre à jour un projet sur une base Svn

Se placer dans le répertoire du projet désiré et y ouvrir un shell.
Exécuter la commande « svn commit -m *message* » où *message* est les commentaires sur les modifications que vous avez apportées par rapport à la version précédente.

Ajouter un fichier ou répertoire à un projet sur une base Svn

Se placer dans le répertoire du projet désiré et y ouvrir un shell.
Exécuter la commande « svn add *chemin* » où *chemin* est le chemin du fichier ou du répertoire.

Remarque :

Toujours effectuer une mise à jour du projet à partir de la base après ces deux dernières opérations.

Installation d'Eclipse

Télécharger « Eclipse SDK 3.2.1 » sur **http://www.eclipse.org/downloads**.
Décompresser l'archive.

Remarque :

Pour une installation sur Linux, il est conseillé d'ajouter un alias vers l'exécutable de l'application nomme « eclipse » et d'actualiser le PATH en conséquence.

Téléchargement de SmartTools

On peut utiliser et télécharger SmartTools depuis une base Svn de deux manières différentes qui sont en tant que simple utilisateur ou développeur.

Utilisateur

Créer un répertoire « SmartToolsSVN » et y ouvrir un shell.
Exécuter la commande « svn checkout svn://scm.gforge.inria.fr/svn/smarttools ».

Développeur

Installer et configurer un client Svn et le logiciel Ssh.
Générer une paire de clés (publique et privée) pour les connexions via Ssh :
Exécuter la commande « ssh-keygen -t -rsa ».
Valider à la première question sans spécifier un nom de fichier.
Entrer à la seconde question une phrase simple à retenir qui servira de phrase de passe (passphrase) pour les accès à la base Svn.

La clé publique est créée dans le fichier « ./.ssh/id_rsa.pub ».
La clé privée est créée dans le fichier « ./.ssh/id_rsa ».

Copier la clé publique.

Se connecter sur le site « **http://gforge.inria.fr/projects/smartTools** » :
Créer un compte Gforge en cliquant sur le lien « nouveau compte ».
Joindre le projet SmartTools en cliquant sur « Request to join ».
Attendre que votre requête soit validée par l'administrateur du projet.

Après validation, votre nom sera inclus dans la liste des développeurs.

Cliquer sur l'onglet « Ma Page ».

Votre liste de projets doit contenir « SmartTools Software Factory ».

Cliquer sur le lien « Gestion du compte ».
Cliquer sur le lien « Éditer les clés » et coller la clé publique avant de valider.
Attendre au moins une heure avant que votre clé publique soit publiée sur Gforge.

Remarque :

Si le lien « Éditer les clés » n'est pas présent, aller à la page **https://gforge.inria.fr/account/editsshkeys.php**.

Créer un répertoire « SmartToolsSVN » et y ouvrir un shell.
Exécuter la commande « svn checkout svn+ssh://*login*@scm.gforge.inria.fr/svn/smarttools » où *login* est votre nom de développeur enregistré sur Gforge.
Entrer à la question votre phrase de passe que vous avez retenue.

Génération et installation des composants SmartTools

Aller dans le répertoire « SmartToolsSVN/smarttools/trunk/SmartTools/ant/developer ».
Exécuter la commande « *./build all* ».

Génération manuelle de composants

Aller dans le répertoire « SmartToolsSVN/smarttools/trunk/SmartTools/ant/developer ».
Exécuter la commande « *./build user.component* ».
Entrez le nom du composant à générer.

Utilisation de SmartTools dans Eclipse

Génération des composants

Aller dans le répertoire « SmartToolsSVN/smarttools/trunk/SmartTools/ant/developer ».
Exécuter la commande « *./build user.component* ».
Entrez le nom du composant à générer.

Importation des composants

Ouvrir Eclipse.
Aller dans le menu « File ».
Cliquer sur « Import ».
Cliquer sur « General » dans la nouvelle fenêtre qui s'ouvre.
Cliquer sur « Existing Projects into Workspace ».
Choisir le répertoire « SmartToolsSVN/smarttools/trunk » pour importer tous les projets.

Les composants ST importés sont représentés dans Eclipse sous forme de projets.

Détails de la communication composant ST - vue Eclipse

La communication entre les plugins Eclipse et les composants SmartTools va se faire grâce à un système de listeners. Un activeur OSGi permettra de l'instancier. En effet, les plugins étant basés sur le principe OSGi tout comme les composants, c'est le CM qui va établir la communication entre eux.

Principe

- Le CM de SmartTools utilise 2 services OSGi pour communiquer avec la partie statique pure OSGI (activeur) des composants.
- L'activeur permet d'instancier des composants avec leur container.
- Il peut donc instancier aussi l'objet Eclipse qu'on souhaite associer à ce composant.
- Pour éviter une dépendance entre le container et l'objet Eclipse, la communication du container vers cet objet Eclipse se fera par un listener. Dans l'autre sens, l'objet Eclipse aura un accès direct (par un attribut) au container.
- Quand l'objet Eclipse veut informer le composant, il l'indiquera directement via un appel de méthode au container qui va ensuite modifier la partie métier.
- De la même manière que dans le programme SmartTools, cette dernière va avertir d'un changement ce même container qui va ensuite indiquer la modification aux containers des autres composants ayant établi une communication avec lui.

Étapes d'instanciation

- Le CM appelle la méthode de création du composant SmartTools C1 de l'activeur.
  La communication du code métier C1m vers le container C1c se fera par un listener. Dans l'autre sens, le container aura un accès direct au code métier.
- Le CM instancie le plugin Eclipse P1 correspondant (référencé dans l'activeur) en lui fournissant en attribut le container.
  La communication du container vers le plugin se fera par un listener. Dans l'autre sens, le plugin aura un accès direct (par un attribut) au container.
- Le CM répète les étapes 1 et 2 pour un autre composant C2.
- C1 informe le CM qu'il veut se connecter à C2.
  Le CM récupère les spécifications (fichier cdml référencé dans l'activeur) de C1 et C2. On suppose qu'elles sont compatibles. Le CM connecte donc les deux containers qui communiquent par des listeners dans un sens comme dans l'autre.
- C1 et C2 sont connectés.

Étapes de fonctionnement

Supposons que P1 vienne d'être modifié ...

- P1 informe C1c des modifications par accès direct.
- C1c informe C1m des modifications par accès direct.
  C1m enregistre les modifications.
- C1c est averti des modifications effectuées sur C1m par un listener.
- C2c est averti des modifications effectuées sur C1c par un listener.
- On répète les étapes 2 et 3 pour C2.
- P2 est averti des modifications effectuées sur C2c par un listener.
  P2 enregistre les modifications.