INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

Transformation de l'architecture orientée services de SmartTools au-dessus du frame-work OSGi

Responsable :
Didier PARIGOT

Etudiant :
DANG Hoang Vu

Sophia Antipolis, Juillet 2006

Table de matières

1. Introduction 4
2. SmartTools 5
  2.1. Modèle de composants de SmartTools 5
  2.2. Architecture orientée services de SmartTools 6
3. La plate-forme OSGi 7
4. Objectif du travail 9
5. Demarche 9
  3.1. Phase 1 10
    3.1.1. Expérimentation 1 10
    3.1.2. Expérimentation 2 11
    3.1.3. Expérimentation 3 11
    3.1.4. Expérimentation 4 12
    3.1.5. Expérimentation 5 14
    3.1.6. Expérimentation 6 26
6. Conclusion et perspectives 36
7. Références 37
8. Annexe 38
  6.1. Guide d'utilisation OSCAR 38
  6.2. Expérimentation 1 - réalisation le bundle cmp2 38
  6.3. Expérimentation 1 - réalisation le bundle cmp1 40
  6.4. Expérimentation 2 - réalisation le bundle cmp2 42
  6.5. Expérimentation 2 - réalisation le bundle cmp1 42
  6.6. Expérimentation 4 - réalisation le bundle cmp2 45
  6.7. Expérimentation 4 - réalisation le bundle cmp1 48
  6.8. La liste des fichier modifiés 52

Table des illustrations

Figure 1: Description du composant Graph... 6
Figure 2: Modèle graphique du composant Graph[1] 6
Figure 3: Fonctionnement du Component Manager [1] 7
Figure 4: Exemple d'application basé sur OSGi [3].. 8
Figure 5: Architecture de la plate-forme OSGi [3] 8
Figure 6: Communication entre deux bundles.. 10
Figure 7: Communication entre deux bundles avec Service Binder................................. 12
Figure 8: Communication entre 3 bundles avec ServiceListener et des observateurs de service 13
Figure 9: Modèle de communication asynchrone entre des bundles 14
Figure 10: Modèle des classes de cmp1 17
Figure 11: Modèle des classes de cmp2 20
Figure 12: Modèle des classes de cmp2 21
Figure 13: Modèle de communication des composants SmartTools dans OSGi.............. 26
Figure 15: Une vue graphiques des liaisons entre des instances des composants........... 34
Figure 16: Une version simple de SmartTools au-dessus OSGi...................................... 35

1. Introduction

Aujourd'hui, avec l'évolution d'Internet, la conception et le développement d'applications complexes doivent impérativement prendre en compte les aspects de répartition, déploiement et réutilisation de code. Cette évolution a conduit un changement radical dans la programmation des application. Avec ces bouleversements, les anciens paradigmes de programmation ont présenté des limitations, et ces limitations ont conduit à l'émergence d'une nouvelle l'approche de programmation - l'approche à composants orientés services (SOA) afin de mieux séparer les aspects de communication et les parties fonctionnelles.

La plate-forme **OSGi** (Open Services Gateway Initiative) –définie par OSGi Alliance en 1999- est une plate-forme dynamique de service qui est soutenu par des grandes projets informatiques comme Eclipse, Apache, etc. OSGi définit un ensemble de mécanismes permettant de déployer, de découvrir et de gérer des services à l'intérieur de la plate-forme (appelé le frame-work). Cependant tous les notions liés aux services d'OSGi sont encore très élémentaires Cela ne permet pas à la plate-forme OSGi de devenir une vrai plate-forme orientée services.

Chez l'équipe **SmartTools** de l'INRIA, on a construit un atelier logiciel nommé SmartTools, C'est un générateur d'outils pour les langages de programmation ou métiers qui s'appuie fortement sur les technologies objets et XML. L'architecture de SmartTools est fondé sur l'approche à composants orientés services. SmartTools possède son propre modèle de composants qui est facilement transposable vers d'autres technologies de composants et qui est bien spécifique aux contraintes de conception de SmartTools. L'interaction entre des composants est réalisée en se basant sur une architecture orientée services (SOA) qui rend SmartTools plus dynamique et flexible.

L'objectif du document est de présenter une démarche pour transformer l'architecture orientée services de SmartTools au-dessus OSGi. Ce travail permet d'ajouter à OSGi les notions de l'architecture orientée services qui est spécifique de SmartTools en profitant de tous les avantages (la fiabilité, la dynamité, les standards) de OSGi. L'objectif à plus long terme du travail est d'intégrer SmartTools dans **Eclipse** (l'Eclipse utilise maintenant la plate-forme OSGi comme support d'exécution de ses plug-ins).

Ce document se compose 8 chapitres

- Chapitre 1: C'est une petite introduction sur le travail.
- Chapitre 2: La présentation sur SmartTools, son modèle de composant et son architecture orientée services.
- Chapitre 3: Introduction sur la plate-forme OSGi.
- Chapitre 4: L'objectif du travail.
- Chapitre 5: Présenter la démarche pour proposer le modèle de transformation de l'architecture orientée services de SmartTools au-dessus du frame-work OSGi.
- Chapitre 6: C'est une petite conclusion.
- Chapitre 7: Les références
- Chapitre 8: Les détails techniques pour la démarche de transformation

2. SmartTools

SmartTools – un atelier logiciel construit par l'équipe **SmartTools** de l'INRIA - est un générateur d'outils pour les langages de programmation ou métiers qui s'appuie fortement sur les technologies objets et XML[1]. L'architecture de SmartTools est fondée sur l'approche à composants orientés services (SOA). SmartTools possède son propre modèle de composants qui est facilement transposable vers d'autres technologies de composants. Cet atelier logiciel se compose de plusieurs composants, chaque composant fournit un ensemble de services et requit aussi des services d'autres composants. L'interaction entre les composants est réalisée en se basant sur une architecture orientée services (SOA) qui rend SmartTools plus dynamique et flexible. Dans les deux parties qui suivent, nous allons présenter en détails le modèle de composants et l'architecture orientée services de SmartTools.

2.1. Modèle de composants de SmartTools

Dans SmartTools, On utilise un langage spécifique pour décrire des composants. Ce langage est nommé le langage CDML, il est basé sur XML. Chaque composant de SmartTools se modélise par un fichier CDML, à partir du fichier de description, SmartTools utilise un générateur de composant pour générer le code des composants. Au niveau de conception, un composant SmartTools se compose deux parties :

- **Le conteneur**

Le conteneur est chargé d'interagir avec l'environnement d'exécution et de réaliser la communication entre les composants. Tous les échanges avec autre composant réalisés dans cette partie. Le mode de communication entre les composants SmartTools est le mode asynchrone. Les composants communiquent en envoyant des messages.

- **La façade**

C'est la partie métier du composant où tous les services du composant sont implémentés. Les services du composant SmartTools sont décris dans le fichier **cdml**. Dans SmartTools, il y a deux types de services: ***les services d'entrées*** (***Les inputs***: sont des services fournis par le composant) et **le*s service de sorties*** (***Les outputs***: sont des services requis, l'implémentation de ces services est réalisée par les autre composants). Au niveau de la conception d'un composant SmartTools, la façade est un élément qui appartient au conteneur. Une contrainte dans la conception des composants de SmartTools est que la façade soit indépendante de son conteneur, c-à-d Il n'y a aucune référence du container dans la façade. Cette contrainte rend la partie métier (la façade) indépendante de toutes les technologies sur lesquelles on implémente le composant. Donc pour établir la communication entre la façade et son conteneur, dans SmartTools, on utilise le patron de conception "listener". Pour plus de détails sur cette architecture, le lecteur pourra se reporter à [1].

La figure 1 montre un exemple de la description de composant **Graph** (visualisation d'un graphe) et la figure 2 est sa représentation graphique.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="graph" type="graph" extends="abstractContainer">

    <containerclass name="GraphContainer"/>
    <facadeclass name="GraphFacade"/>
    <dependance name="koala-graphics" jar="koala-graphics.jar"/>
    <attribute name="nodeType" javatype="java.lang.String"/>

   <input name="addComponent" method="addNode">
      <parameter name="nodeName" javatype="java.lang.String"/>
      <parameter name="nodeColor" javatype="java.lang.String"/>
   </input>

   <input name="addEdge" method="addEdge">
      <parameter name="srcNodeName" javatype="java.lang.String"/>
      <parameter name="destNodeName" javatype="java.lang.String"/>
   </input>

</component>
```

Figure 1: Description du composant Graph

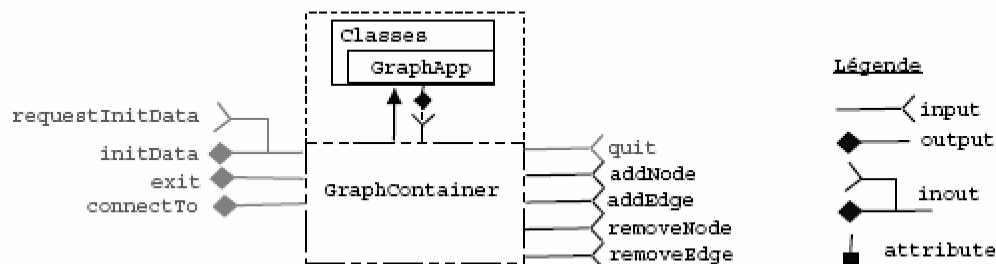


Figure 2: Modèle graphique du composant Graph [1]

2.2. Architecture orientée services de SmartTools

Dans SmartTools, les fonctionnalités de composants sont exposées sous forme des services (les services d’entrées et les services de sorties). Pour gérer des services, SmartTools a introduit un architecture orientée services sur laquelle toutes les échanges entre des composants sont réalisées. L’élément principale de cette architecture est un composant spécial appelé le Gestionnaire de Composants (Component Manager ou CM). CM est chargé de gérer le lancement, la mise à jour et l’arrêt des composants. CM lie les composants entre eux à l’aide du service ***connectTo*** qui permet à un composant de

demander à être connecté avec un autre composant. Après que les connexions sont établies, les composants communiquent directement entre eux en envoyant des messages

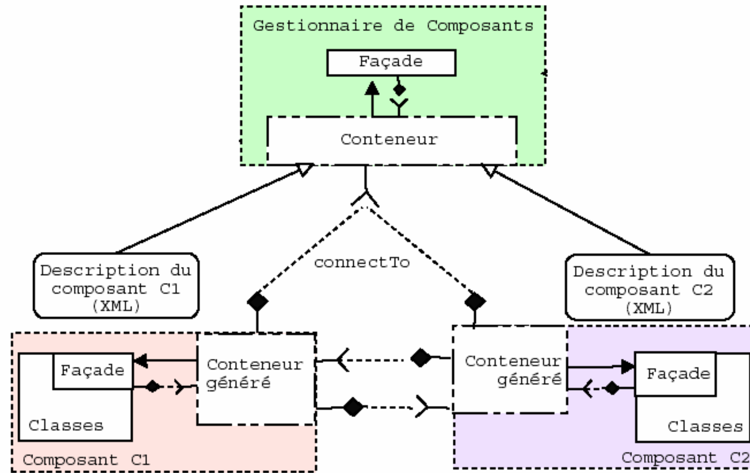


Figure 3: Fonctionnement du Component Manager [1]

3. La plate-forme OSGi

Formé sur 1999, le consortium d'OSGi (Open Services Gateway Initiative) s'est focalisé au commencement sur des solutions pour les systèmes embarqués et les réseaux de dispositifs. Aujourd'hui, la technologie d'OSGi a été acceptée par la communauté Open Source, comme les projets : Apache Felix, Derby, Eclipse Callisto, Equinox, Corona, OSCAR, et Knopflerfish, etc... En conséquence cette technologie d'OSGi est maintenant de plus en plus répandue dans l'entreprise, et on la voit également comme élément clé pour une plate-forme de service de Java de prochaine génération qui permet le déploiement dynamique des services Web 2.0. *[2]*

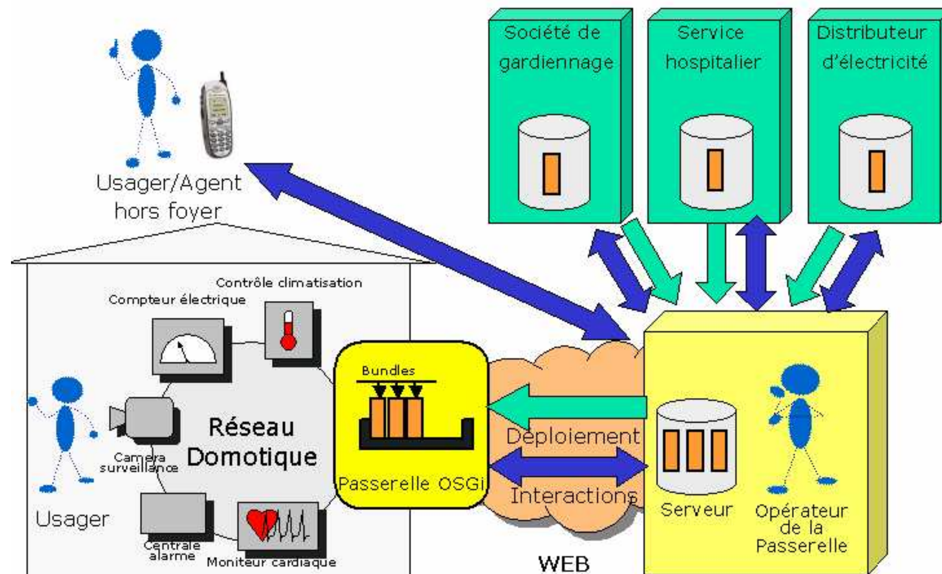


Figure 4: Exemple d'application basé sur OSGi [3]

La plate-forme OSGi est une plate-forme dynamique de service. Elle fournit des moyens permettant de déployer des services à l'intérieur de la plate-forme (appelé le frame-work). Dans OSGi, chaque unité de déploiement est appelée un ***bundle***. Une application OSGi se compose d'un ensemble de ***bundles***. Un bunble OSGi peut être considéré comme la partie du conteneur d'un composant SmartTools. On verra par la suite que cela correspond plus à la notion de type de composant. En effet on ne peut pas instancier plusieurs fois un même bundle.

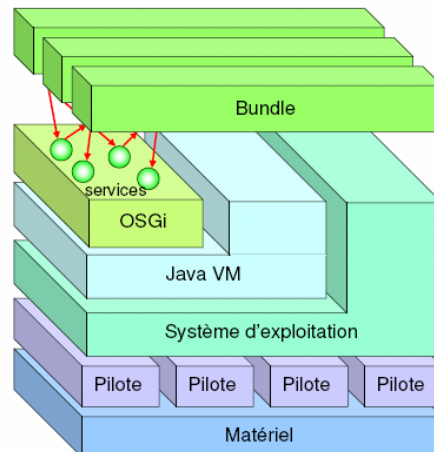


Figure 5: Architecture de la plate-forme OSGi [3]

Le ***frame-work*** OSGi fournit des mécanismes permettant de gérer le cycle de vie des bundles. L'installation, le lancement, la mise à jour, l'arrêt et la retrait de bundles sont effectué dynamiquement sans interruption de l'exécution de l'application.
Lorsque le bundle est installé et lancé, il peut publier ou découvrir des ***services***. Les services sont des objets Java qui implémentent un contrat bien défini. Les services sont publiés et découverts à l'aide un registre de services fourni par la plate-forme. Pour supporter la disponibilité dynamique des services, le frame-work OSGi fournit aussi un

mécanisme de notification d'événements de services permettent de connaître les arrivées et les départs de services dans le frame-work.

Introduction de l'OSCAR

Oscar est une implémentation open-source d'OSGi. Le but est de fournir une implémentation complètement conforme aux spécifications d'OSGi. Oscar est actuellement conforme avec une grande partie de la spécifications OSGi version 3.
L'exécutable d'OSCAR et ses documents se trouvent sur le site web http://oscar.objectweb.org/

4. Objectif du travail

L'objectif du stage est de comprendre comment instancier l'architecture de SmartTools au-dessus du frame-work OSGi, sachant qu'à plus long terme, nous souhaitons intégrer SmartTools dans Eclipse. En effet depuis sa version 3.0, l'Eclipse utilise la plate-forme OSGi comme support d'exécution de ses plug-ins, les plug-ins de l'Eclipse sont désormais des bundles OSGi. Mais dans un premier temps nous nous focalisons à construire un modèle de composant SmartTools au-dessus d'OSGi indépendamment d'Eclipse. Pour simplifier le travail, nous utiliserons OSCAR comme l'implémentation d'OSGi dans toutes nos expérimentations.

En particulièrement, les tâches que nous avons a effectuées sont :

- Proposer un modèle des composants SmartTools au-dessus la plate-forme OSGi.
- Réaliser le mécanisme de communication entre la façade et le conteneur.
- Réaliser le mécanisme de communication asynchrone entre les composants.
- Construire une version de démonstration de SmartTools au-dessus OSGi

5. Demarche

Pour atteindre notre objectif, nous divisions le travail en plusieurs phases, à chaque phase, nous faisons quelques expérimentations pour identifier les points faibles, les point forts et les problèmes restants de la solution proposée.

L'idée des expérimentations est de prendre deux composants cmp1 et cmp2 (de SmartTool) et d'établir la communication (un seul service) dans le contexte OSGi en essayant de respecter toutes les contraintes de conception définies dans SmartTools. Pour toutes ces expérimentations, nous utilisons les 4 composants de base suivants :

- ✓ **St-jar**: le composant qui contient les librairies (fichier jar utile pour SmartTools).
- ✓ **St-core**: le noyau de SmartTools. Ce composant contient toutes les classes et les ressources nécessaires pour construire les autres composants.
- ✓ **Cmp1**: un composant simple. qui envoie un message *updateLabel* vers l'autre composant.
- ✓ **Cmp2**: un composant SmartTools qui reçoit un message *updateLabel* et affichera la valeur (chaîne de caractère) dans une interface graphique.

Eventuellement, pour quelques expérimentions, nous utilisons aussi le Component Manager.

3.1. Phase 1

<u>*Objectif*</u>: Mettre en œuvre un modèle simple des composants SmartTools dans OSGi et comprendre le mécanisme de communication entre les bundles OSGi.

3.1.1. Expérimentation 1

Motivation: Cette expérimentation nous permet de comprendre comment fonctionne un bundle dans OSGi et comment les bundles communiquent.

Démarche:

- Utiliser la partie métier des deux composants ***cmp*1** et ***cmp2*** (les deux classes ***Cmp1App.java*** , ***Cmp2App.java*** *et* ***les façades des composants***) de SmartTools
- Créer le bundle ***cmp2*** (la classe activer) qui fournit le service *UpdateLabel* Service en utilisant ***Cmp2App.java*** comme l'implémentation de ce service.
- Créer le bundle cmp1 avec la partie métier ***Cmp1App.java*** qui utilise le service *UpdateLabel* Service fourni par ***cmp2***
- Lancer les deux composants dans le frame-work d'Oscar, envoyer un message de ***cmp1*** a ***cmp2***

(Les *détails techniques pour créer ces bundles sont présentés dans l'annexe)*

Dans la figure 3, le diagramme représente la communication entre les deux bundles pour cette expérimentation.

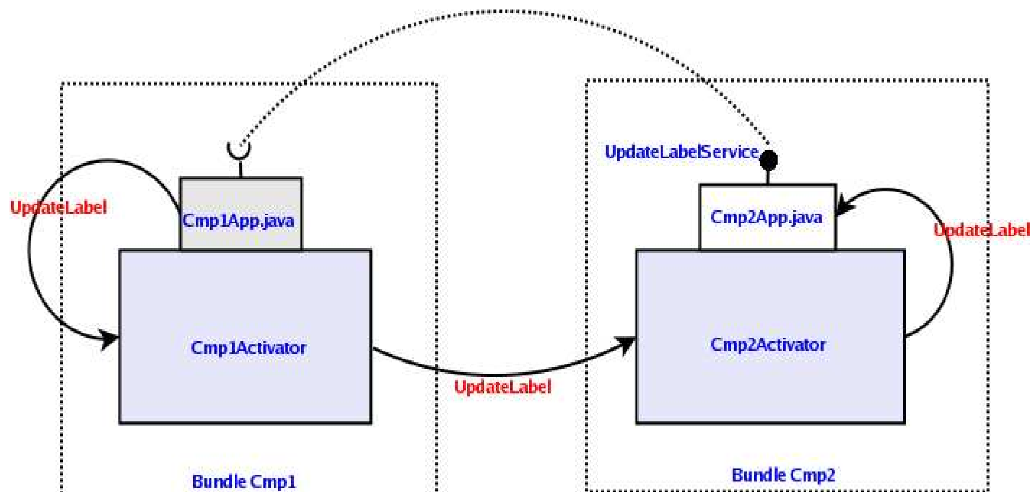


Figure 6: Communication entre deux bundles

Evaluation: Bien que la communication entre les deux bundles fonctionne bien avec ce modèle, cependant pour que la partie métier puisse communiquer avec son conteneur (le conteneur dans le contexte d'un bundle c'est ***l'activator*** du bundle), il faut une référence du conteneur dans l'implémentation de la partie métier (***la façade***). Cela rend la partie métier dépendante du conteneur, cela viole la contrainte de conception d'un composant SmartTools.

3.1.2. Expérimentation 2

Motivation**:* Dans cette expérimentation, nous essayons de réutiliser les codes générés par SmartTools (les conteneurs) des deux composants ***cmp1 et ***cmp2*** pour construire deux bundles ***cmp1*** et ***cmp2***. Pour résoudre le problème de dépendance de la partie métier et son conteneur, nous emploierons le patron de conception "***listener***" pour réaliser les échanges entre la partie métier et son conteneur comme cela est fait dans SmartTools.

***Démarche*:**

- Utiliser les codes générés par SmartTools des deux composants ***cmp*1** et ***cmp2***
- Créer le bundle ***cmp2*** qui fournit le service *cmp2AppFacadeInterface* et puis créer le bundle ***cmp1*** qui utilise ce service.
- Lancer les 4 composants ***st-jar, st-core, cmp2, cmp1*** dans le frame-work d'Oscar, envoyer un message de cmp1 à cmp2.

(*Les détails techniques pour créer ces bundles sont présentés dans l'annexe*)

Evaluation: Dans cette expérimentation, nous résoudrons bien le problème de séparation entre la partie métier et son conteneur pour un composant basé sur l'OSGi. Cependant il existe encore un point faible avec cette solution: En fait tous les services dans OSGI sont déclarés comme des interfaces Java et pour utiliser un service fourni, le composant qui utilise ce service doit connaître l'interface Java correspondant. Le code de ce composant doit spécifier explicitement le nom de l'interface dont il a besoin (il faut faire un import pour référencer à cet interface). Ce point faible diminue la flexibilité du composant.

3.1.3. Expérimentation 3

Service Binder:

Dans OSGi, toutes les activités concernant l'enregistrement de services, l'assemblage d'une application et l'adaptation par rapport aux changements doivent être réalisés à travers une programmation explicite. Le *ServiceBinder* - développé au laboratoire IMAG par Humberto Cervantes - est un service qui peut nous aider à gérer l'enregistrement de services et les liaisons de services en utilisant une descriptive XML. Le Service Binder est aussi un bundle qui est compatible avec toutes les implémentations d'OSGi.

***Motivation**:* Cette expérimentation a pour but d'appliquer le Service Binder aux deux bundles cmp1 et cmp2 et puis identifier les avantages et les inconvénients de cette solution.

***Démarche*:**

- Réutiliser les codes sources de **cmp1 et cmp2** de l'expérimentation 2.
- Modifier le code de la classe d'activation (***Activator***) pour utiliser les fonctionnalités du Service Binder
- Lancer les 5 composants ***st-jar***, ***st-core***, Service Binder, cmp2, cmp1 dans le frame-work d'Oscar, et envoyer un message de ***cmp1 à cmp2***

(*Les détails techniques pour créer ces bundles sont présentés dans l'annexe*)

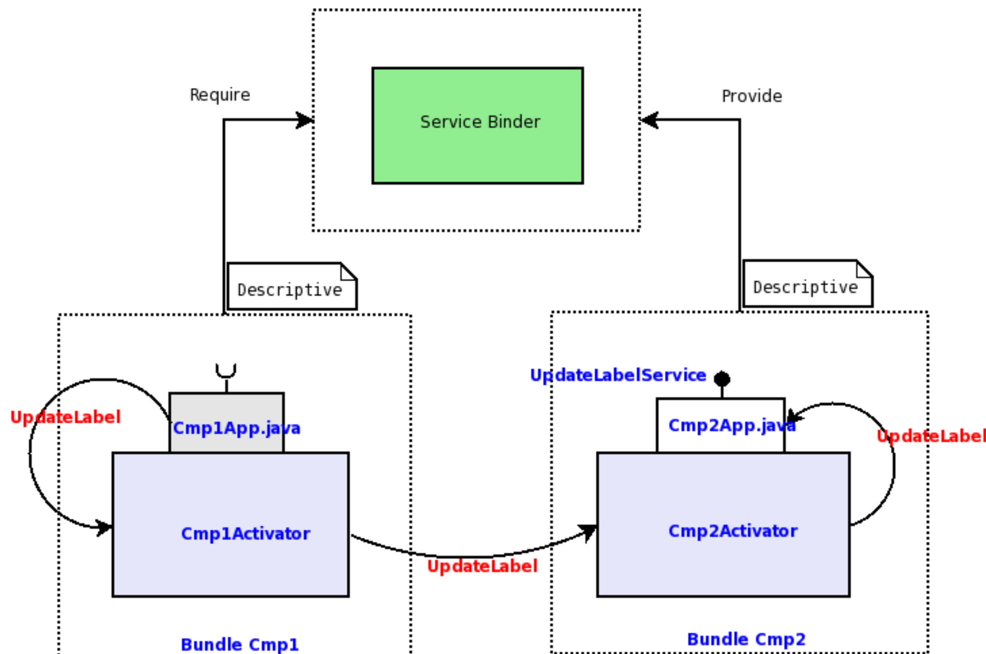


Figure 7: Communication entre deux bundles avec Service Binder

Evaluation: Le Service Binder gère bien l'enregistrement, les liaisons, la disponibilité dynamique de services. Cependant quand nous utilisons le Service Binder, il nous faut laisser le Service Binder créer pour nous la partie métier (la façade). Donc il est impossible d'appliquer le patron de conception "listener" pour lier la partie métier et son conteneur (nous ne savons pas quand la partie métier sera créée pour ajouter les listeners)

3.1.4. Expérimentation 4

Motivation: Comme le Service Binder ne peut pas être appliqué pour résoudre le problème de communication entre les bundles. Nous pensons à l'idée d'utiliser le mécanisme de notification des événements du frame-work pour réaliser la communication. Comme dans le Service Binder, nous utilisons aussi un fichier de description pour déclarer les services fournis et les services requis d'un bundle. Pour chaque service requis, nous utilisons un observateur pour écouter les événements du frame-work pour connaître l'arrivée et le départ du service. Grâce à ces observateurs, le conteneur est capable de gérer la disponibilité dynamique de services comme on le fait avec le Service Binder.

Dans cette expérimentation, nous réalisons une application avec 3 bundles: ***cmp1***, ***cmp2*** comme dans le deux expérimentation précédentes et un autre bundle ***cmptest*** qui fournit un service et requérir l'autre service.

Au dessous, le diagramme représente la communication entre 3 bundles dans cette expérimentation.

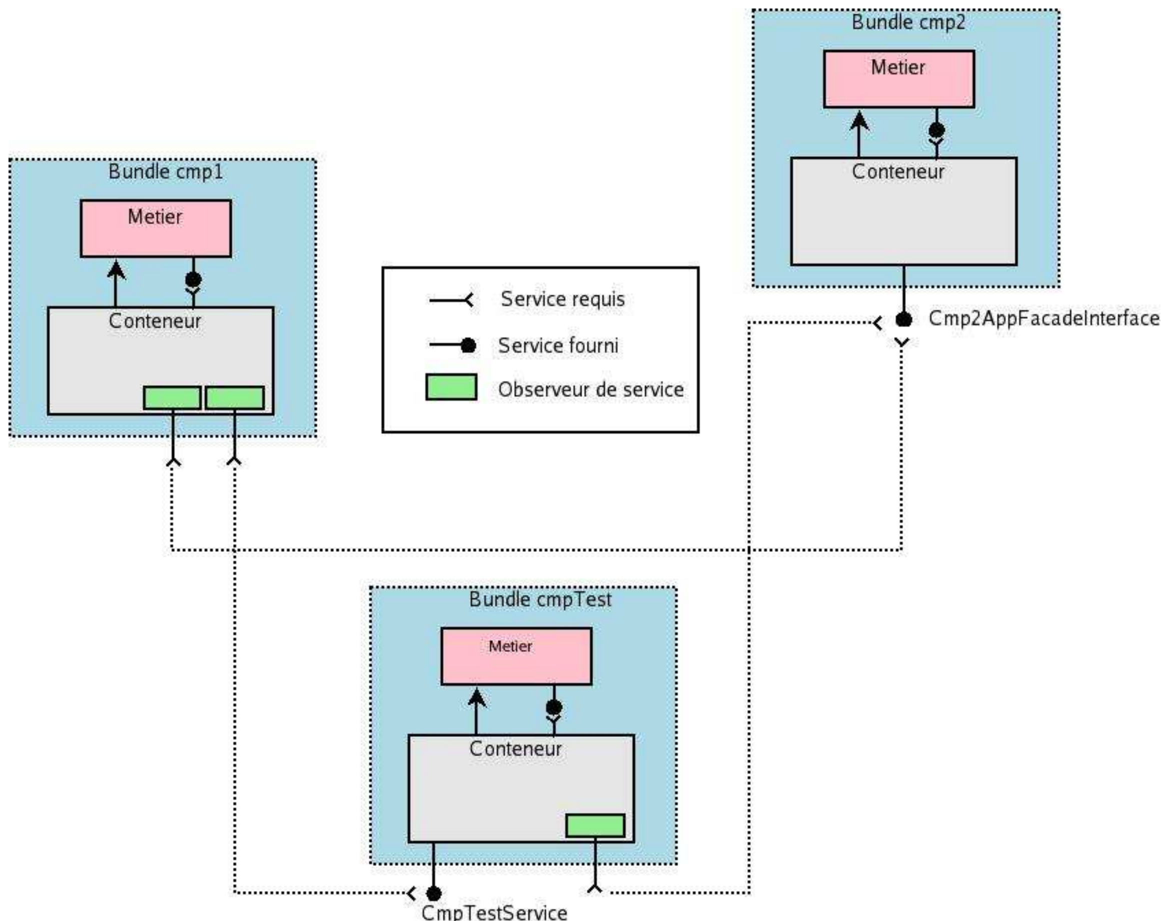


Figure 8: Communication entre 3 bundles avec ServiceListener et des observateurs de service

***Démarche*:**

- Réutiliser les codes sources de ***cmp1*** et ***cmp2*** de l'expérimentation 2.
- Ajouter une classes qui joue le rôle d'un observateur de service (en implantant l'interface ***ServiceListener***)
- Modifier le code de la classe d'activation (Activator)
  - ✓ Lire le fichier de description.
  - ✓ Créer et enregistrer les services
  - ✓ Récupérer les références de services requis
  - ✓ Traiter les événements de services (d'arrivée et de départ d'un service requis)
- Créer le bundle ***cmptest*** qui fournit un service et requérir le service fourni par cmp2.
- Lancer les 5 composants ***st-jar, st-core, cmp2, cmp1, cmptest*** dans le frame-work d'Oscar, envoyer un message de ***cmp1*** a ***cmp2***

(*Les détails techniques pour créer ces bundles sont présentés dans l'annexe*)

Evaluation: L'avantage de ce modèle est qu'il est capable de résoudre le problème de communication entre plusieurs bundles. Il respecte bien la contrainte de séparation entre

la partie métier et son conteneur. Cependant ce modèle ne répond pas la question de la communication asynchrone entre des composants SmartTools.
En fait dans SmartTools si un composant veut appeler un méthode (un service) d'un autre composant, Il lui suffit d'envoyer un message qui contient tous les informations nécessaires (le nom de la méthode et les paramètres) pour invoquer cette méthode au composant destinataire. Le composant destinataire stocke ce message dans une file d'attente et un processus spécifique de ce composant (dans le conteneur) traitera tous les messages stockés dans la file d'attente. Ce mécanisme rend la communication entre les composants SmartTools plus flexible et il permet d'éviter des blocages au cas où le temps d'exécution d'un service est assez grand.
Par contre, dans le modèle de cette expérimentation, tous les services sont déclarés comme des interfaces Java. Pour invoquer une méthode d'un service fournis par autre composant, il nous faut la référence d'objet qui implémente le service et puis nous faisons un appel explicitement en utilisant le nom de la méthode et attendons dès que ce méthode est terminé. C-à-d la communication entre des composants dans ce modèle est synchrone.

3.1.5. Expérimentation 5

Motivation: En fait, pour résoudre le problème d'appels asynchrones, la spécification OSGi R4 a introduit un service supplémentaire nommé ***EventAdmin*** Service qui permet aux bundles de communiquer en envoyant des événements. Malheureusement, jusqu'à présent il n'y a pas d'implémentation complète de ce service, ou elles sont en cours. De plus nous ne somme pas convaincu que cela va pouvoir résoudre notre problème.
Cette expérimentation a pour but de réaliser des appels asynchrones en combinaison avec le mécanisme de communication par services d'OSGi et la technique d'envoie des événements de SmartTools. La figure ci-dessous décrit le modèle de communication asynchrone entre les bundles.

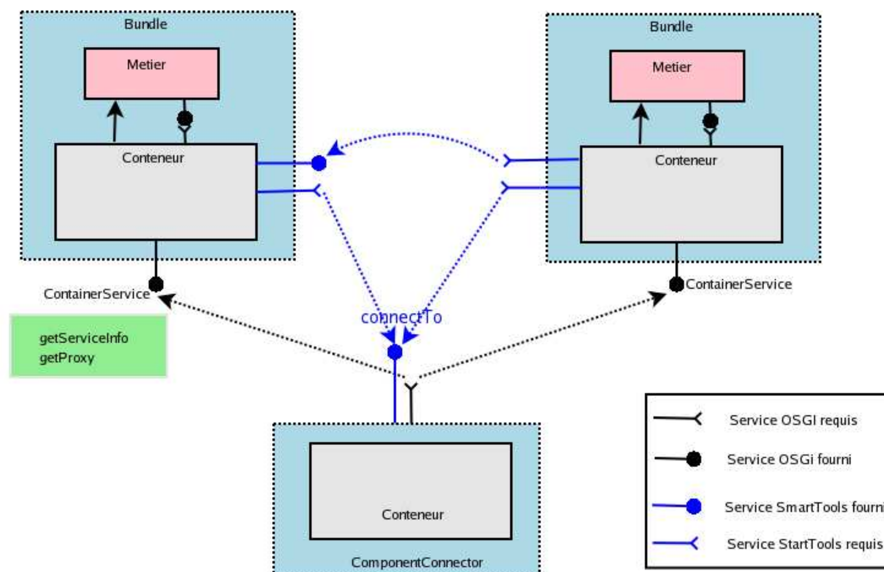


Figure 9: Modèle de communication asynchrone entre des bundles

Dans ce modèle :

- ✓ **Un bundle**: c'est un bundle OSGi qui se compose des classes, des ressources, d'un *activator* et d'un fichier manifest. *L'activator* est chargé de gérer le cycle de vie du bundle. Il contient le conteneur du composant SmartTools. Ce container sera lancé au démarrage du bundle. Chaque bundle proposera deux types de service : les services SmartTools et les services OSGi.
  - Un service OSGi : C'est un service déclaré comme une interface Java, et il est enregistré auprès du frame-work OSGi. Les services OSGi n'appartient pas à la partie métier du système. Dans le modèle au-dessus, chaque bundle propose un service OSGi ***ContainerService***, ce service sera utilisé par le *Component Connector* (voir ci-dessous, pour son rôle) pour établir des connexions entre les bundles.
  - Un service SmartTools:c'est un service déclaré dans le fichier **cdml** d'un composant SmartTools. SmartTools possède un mécanisme spécifique pour gérer les services SmartTools. Le framework OSGi ne connaît pas les services SmartTools. Les services SmartTools réalisent la partie métier du système.
- ✓ **Component Connector(Cc)**: C'est un bundle particulier qui est chargé de gérer une liste des bundles disponibles du système. Grâce au mécanisme de notification des événements de service du frame-work OSGi, Cc est capable de savoir s'il y a un bundle qui arrive ou il y a un bundle qui part (dans le cas du modèle au-dessus, Component Connector écoutera tous les événements du service *ContainerService* pour connaitre les arrivées et les départs des bundles).

  ComponentConnector joue le rôle de Component Manager de SmartTools (un Component Manager simplifié). Il ne fournit qu'un service SmartTools ***ConnectTo*** permettant aux bundles d'établir les connexions entre eux. Après avoir établi les connexions les bundles communiquent directement à l'aide du mécanisme de communication de SmartTools

Démarche:

1. *Modification de st-core*

Dans OSGi, on ne peut pas utiliser les même méthodes d'accès aux ressources des bundles que pour SmartTools. Pour que le coeur de SmartTools puisse fonctionner sous OSGi, Il nous faut modifier un peu le code d'accès des ressources.

- ✓ Dans la classe **AbstractContainer**, modifier la méthode ***internalInit*** comme suit:

```
public void internalInit() {
        // lecture du fichier de description si necessaire
        if (containerDescription == null && resourceFilename != null) {
          containerDescription = new ComponentDescriptionImpl();

          URL url = getClass().getResource(resourceFilename);

          if (url != null) {
                containerDescription.setResourceFilename(resourceFilename);
                containerDescription.process();
          }
          else{
                System.out.println("URL is null");
          }

        }
   }
```

✓ Dans la classe **ComponentDescriptionImpl**, modifier la méthode **load**

```
protected void load() throws MalformedURLException, IOException,
        SAXException, FactoryConfigurationError,
        ParserConfigurationException {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();

        InputStream docFile = getClass().getResourceAsStream(resourceFilename);

        Document document = builder.parse(docFile);
        //Document document = builder.parse(resourceFilename);
        root = document.getDocumentElement();
        System.out.println(root.getNodeName());

   }
```

✓ Ajouter une nouvelle interface **ContainerService** qui fournit le service OSGi permettant à Component Connector de consulter les informations des services des autres composants.

```
package fr.smarttools.core.component;

public interface ContainerService {
        ComponentDescription getServicesInfo();
        ContainerProxy getProxy();
}
```

- ✓ Pour que les ressources soient visibles pour les autres bundles, il nous faut modifier le fichier MANIFEST.MF en exportant le package *fr.smarttools.core.component.resources*

```
Export-Package: fr.smarttools.core.component.resources,
fr.smarttools.core.document,
....
```

2. *Réalisation du composant cmp1*

Pour réaliser le cmp1, nous réutilisons tous les codes générés par SmartTools. En plus nous ajoutons un ***Activator*** pour gérer le cycle de vie du composant. La figure ci-dessous montre la conception du composant ***cmp1*** :

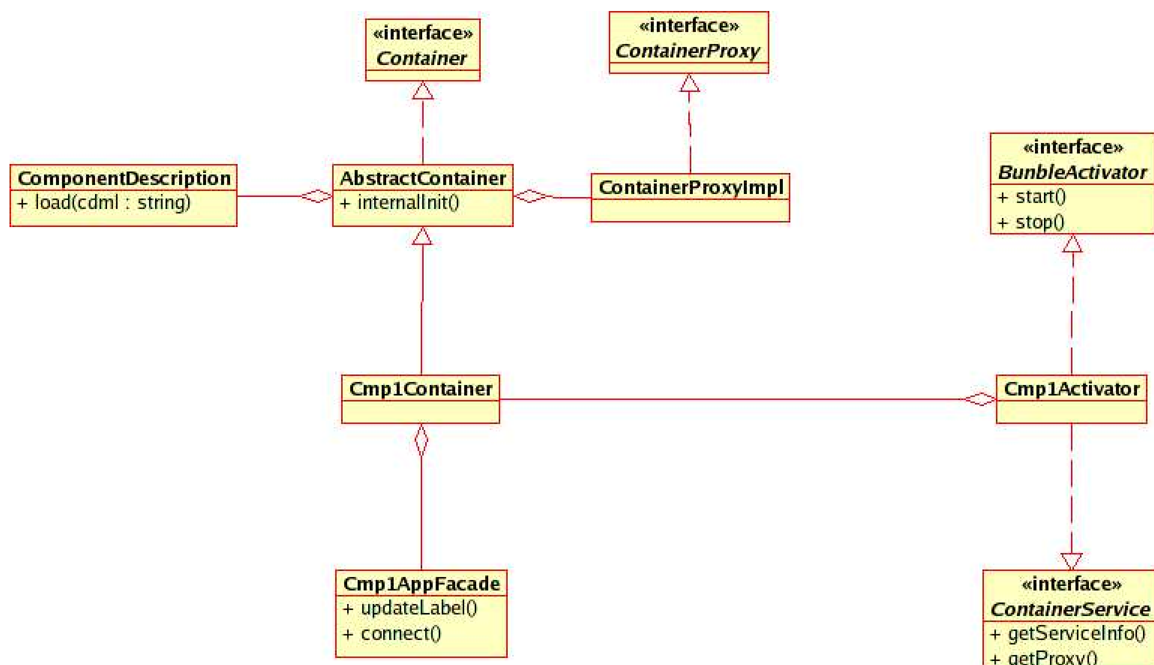


Figure 10: Modèle des classes de cmp1

✓ ***La classe Cmp1Activator:***

Cette classe est l'activator du bundle. Elle implémente le service **ContainerService** et contiens une instance de ***Cmp1Container***:

```
private BundleContext   context = null;
private Cmp1Container  container = null;
private Thread         componentThread = null;
```

Dans la méthode **start**, ajouter le code pour initialiser le composant:

```
public void start(BundleContext context) throws Exception {
		// TODO Auto-generated method stub
		System.out.println("Cmp1 started...");
		//===SmartTools initialization===
		container = new Cmp1Container();
		container.internalInit();
		container.setState(Container.ON);
		componentThread = new Thread(container.getProxy(), "cmp61");
		componentThread.start();
		//===OSGi initialization=========
		this.context = context;
		registerService();
	}
```

Ajouter la méthode pour enregistrer le service **ContainerService** :

```
void registerService(){
		context.registerService(
				ContainerService.class.getName(), this, null);

	}
```

Ajouter le code pour implémenter l'interface **ContainerService** :

```
//=============Container service implementation============
	public ContainerProxy getProxy() {
		// TODO Auto-generated method stub
		return container.getProxy();
	}

	public ComponentDescription getServicesInfo() {
		// TODO Auto-generated method stub
		return container.getContainerDescription();
	}
```

- ✓ **La classe Cmp1App**

Dans cette classe, nous ajoutons une méthode ***connect*** pour demander de ce connecter au composant ***cmp2.*** Cette méthode a pour but de faire une démonstration de la fonctionnalité ***connectTo*** du composant Component Connector.

- ✓ ***La classe Cmp1AppFacade***

Nous surchargeons la méthode ***connect*** comme suit :

```
public  void connect(){
            for(int i = 0 ; i < updateLabelListeners.size() ; i++)
            (( UpdateLabelListener ) updateLabelListeners.elementAt(i)).connect();
          }
```

✓ ***La classe UpdateLabelListener***

Nous ajoutons la méthode ***connect*** dans cette classe :

```
public interface UpdateLabelListener {
  public  void updateLabel(UpdateLabelEvent ev);
  public  void connect();
}
```

✓ ***La classe Cmp1Container***

Nous réalisons l'implémentation de la méthode ***connect*** dans cette classe (Cmp1Container)i, Dans cette méthode nous envoyons un message vers le ***ComposantConnector*** pour demander de lier ce composant avec le composant cmp2.

```
  public void connect(){
          HashMap args = new HashMap();
          args.put("ref_src","cmp1");
          args.put("type_dest","cmp");
          args.put("id_dest","cmp2");
          args.put("actions",null);
          Message m = new MessageImpl("connectTo", args , null);
          send(m);
  }
```

Attention: La fonctionnalité de la méthode ***connect*** a pour but de faire une démonstration seulement. Dans le cas général, cette méthode sera réalisée autrement.

✓ *Le fichier MANIFEST.MF*

Pour que les ressources soient visibles, Il nous faut modifier le fichier MANIFEST.MF en ajoutant le package *cmp1.resources* comme un package importé.

```
Export-Package: cmp1, cmp1.resources
```

3. *Réalisation du composant cmp2*

La réalisation de cmp2 est presque identique à celle de cmp1.
Le diagramme des classes de cmp2 est présenté dans la figure 6

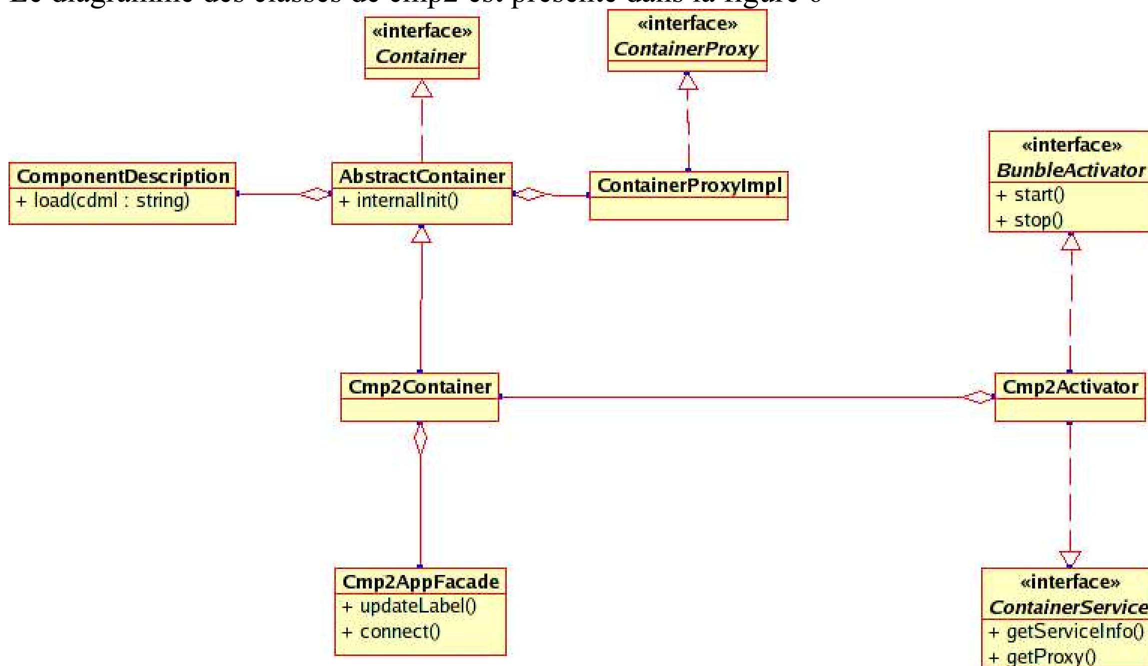


Figure 11: Modèle des classes de cmp2

4. *Réalisation de ComposantConnector*

Le diagramme des classes de ComponentConnector est présenté dans la figure 7 :

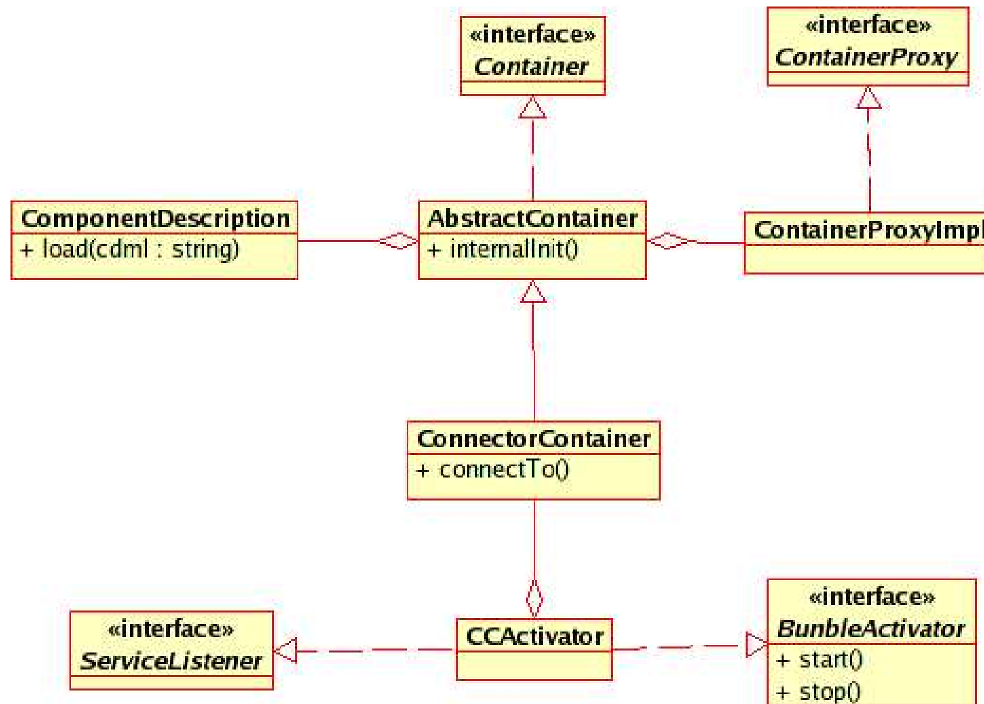


Figure 12: Modèle des classes de cmp2

✓ ***Le fichier cmdl***

ComponentConnector ne fournit qu'un service SmartTools ***ConnectTo*** permettant aux bundles d'établir des connexions entre eux. Ce service est déclaré dans un fichier **cdml** de ce composant, le tableau ci-dessous montre le contenu de ce fichier

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--                          -->
<!--  Connector Component Declaration  -->
<!--                          -->
<component name="connector"
        type="connector"
        extends="abstractContainer"
        ns="connector">
         <containerclass name="ConnectorContainer"/>
         <!-- INPUT -->

         <input name="connectTo"  doc=""  method="connectTo">
           <arg name="ref_src" doc=""     javatype="java.lang.String"/>
           <arg name="type_dest" doc=""  javatype="java.lang.String"/>
           <arg name="id_dest" doc=""     javatype="java.lang.String"/>
           <arg name="actions" doc=""     javatype="java.util.HashMap"/>
         </input>
</component>
```

La classe **ConnectorContainer**

C'est le conteneur SmartTools du bundle, dans cette classe, on garde une liste de tous les composants disponibles du système, cette liste est passée par ***l'activator*** du composant.

Nous réalisons aussi le service **connectTo** ici:

```
//calls
  {
    calls.put("connectTo", new MethodCall() {
      public Object call(ContainerProxy expeditor, HashMap parameters) {
                              connectTo((String) parameters.get("ref_src"),
                                        (String) parameters.get("type_dest"),
                                        (String) parameters.get("id_dest"),
                                        (HashMap) parameters.get("actions"));
                              return null;
        }});
  }

  //connect two components
  void connectTo(String ref_src, String type_dest, String id_dest, HashMap actions){

          Object src = otherComponents.get(ref_src);
          if(src == null){
                  System.out.println("Component " + ref_src + " does not exist");
                  return;
          }

          Object dest = otherComponents.get(id_dest);
          if(dest == null){
                  System.out.println("Component " + id_dest + " does not exist");
                  return;
          }

          ContainerProxy srcComponentProxy = ((ContainerService)src).getProxy();
          ContainerProxy destComponentProxy = ((ContainerService)dest).getProxy();
          srcComponentProxy.connect(destComponentProxy);

  }
```

La classe CCActivator

Cette classe est l'activator du bundle. Elle implémente l'interface **ServiceListener** pour écouter les événements du framework, cela permet d'établir une liste des bundles disponibles dans le système.

```
private BundleContext       context         = null;
private ConnectorContainer  container       = null;
private Thread              componentThread = null;
private HashMap             otherComponents = new HashMap();
```

Dans la méthode **start**, nous réalisons l'initialisation du composant :

```
public void start(BundleContext context) throws Exception {
            // TODO Auto-generated method stub
            System.out.println("Conector started...");

            //===SmartTools initialization===
            container = new ConnectorContainer();
            container.internalInit();
            container.setOtherComponents(otherComponents);
            container.setState(Container.ON);
            componentThread = new Thread(container.getProxy(), "connector");
            componentThread.start();


            //===OSGi initialization=========
            this.context = context;
            context.addServiceListener(this,
    "(objectClass=fr.smarttools.core.component.ContainerService)");

}
```

Ajouter le code pour implémenter l'interface **ServiceListener** :

```
public void serviceChanged(ServiceEvent event) {
        // TODO Auto-generated method stub
        Object service= context.getService(event.getServiceReference());
        Object thisService= (Object)this;

        if (event.getType() == ServiceEvent.REGISTERED){
          connect(service);
        }
        else{
          disconnect(service);
        }
}
```

Après d’avoir exporté st-core, componentConnector, cmp1, cmp2 en fichiers jar Nous lançons ces 4 composants st-jar, st-core, cc, cmp2, cmp1 dans le framework d'Oscar.

Evaluation: Ce modèle a bien résolu tous les problèmes de communication des composants SmartTools dans OSGi. Cependant avec ce modèle, on considère un bundle comme un composant SmartTools donc il est impossible de créer plusieurs instances de composant de même type, car un bundle n’est instancié qu’une fois.

3.1.6. Expérimentation 6

Avec le modèle de l'expérimentation 5, chaque bundle est considéré comme un composant. Comme on ne peut pas instancier un bundle plusieurs fois, donc il est impossible de créer plusieurs instances d'un composant de même type.
Pour résoudre ce problème, nous proposons un autre modèle dans lequel un bundle est considéré comme un type de composant (au sens de SmartTools). Lors du lancement, le bundle fournira un service OSGi qui permet de créer des instances de ce type de composant et de récupérer la description du type de composant (description **CDML**). Ce modèle introduit aussi un **ComponentManager** qui a le même rôle que le **ComponentManager** de SmartTools. **ComponentManager**(CM) est chargé de gérer la liste des types de composants, de créer des instances des composants et de lier ces instances de composant entre eux (avec le service ***connectTo***).

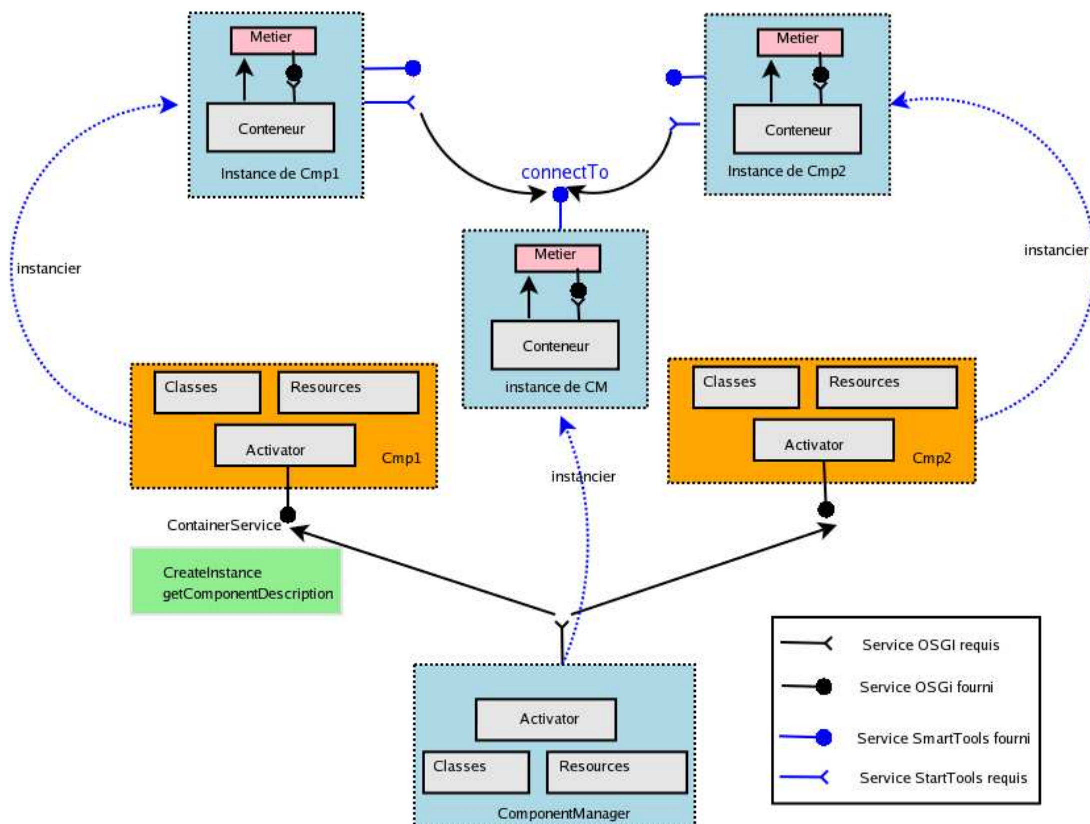


Figure 13: Modèle de communication des composants SmartTools dans OSGi

Dans ce modèle :

✓ **Un bundle**: Un bundle dans ce modèle représente un type de composant. Il contient toutes les facilités (des classes, des ressources, la description du composant) nécessaires pour créer des instances du composant. Lorsque le bundle est lancé, il proposera un service nommé **ContainerService.** Ce service permet au gestionnaire de

composants (Component Manager qui sera présenté après) de consulter la description du composant ou de créer les instances ce composant.
Un bundle se compose des éléments suivants :

- Les classes, les ressources d'un composant SmartTools
- Un ***activator*** qui est utilisé par le frame-work OSGi pour gérer le cycle de vie du bundle
- Un fichier ***manifest*** qui présent des informations concernant le bundle (les packages exportés, les packages importés, le nom du bundle, etc..)

✓ **ComponentManager :** C'est un bundle particulier qui gère tous les bundles (les types de composant) dans le système. Lors de son lancement, une instance de ***ComponentManager***(CM) de SmartTools sera créé. Cette instance lira un fichier de lancement (fichier ***world***). C'est le fichier de lancement des applications SmartTools basé sur XML. Dans le fichier ***world***, on précise tous les composants à charger, tous les instances de composants à instancier et les liaisons entre les instances de composant au démarrage de l'application. Voici un exemple d'un fichier ***world*** :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<world repository="file:/user/hvdang/home/deploy/"
       debug="ON">
   <!--                        -->
   <!-- Components declaration -->
   <!--                        -->
   <load_component jar="graph.jar"
                url="file:/user/hvdang/home/deploy/graph.jar"
                name="graph"/>


   <!--            -->
   <!-- connect -->
   <!--            -->
   <connectTo id_src="ComponentsManager"
      type_dest="worldStateGraph" id_dest="GraphContainer"/>
</world>
```

Figure 14: Un exemple d'un fichier world

Le CM traite ce fichier pour charger les bundles nécessaires, créer les instances de composant et faire les connexions entre les instances de composant.

***Démarche*:**

1. *Modification de st-core*

Les changements des **st-core** sont faits comme dans l'expérimentation 5 sauf la classe ContainerService. **ContainerService** fournit le service OSGi permettant à ***ComponentManager*** de créer une instance d'un composant et de consulter la description des composants.

```
package fr.smarttools.core.component;

public interface ContainerService {
	ComponentDescription getComponentDescription();
	ContainerProxy createInstance(String componentID);
}
```

2. *Réalisation de ComponentManager*

Pour réaliser le ***ComponentManager***, nous réutilisons le code du CM de SmartTools et puis nous modifions un peu dans cette classe pour l'adapter au contexte d'OSGi. En plus nous ajoutons un Activator pour gérer le cycle de vie du bundle.

- ✓ ***La classe Activator***

Cette classe est l'activator du bundle. Elle implémente l'interface **ServiceListener** pour écouter les événements du framework, cela permet d'établir une liste des bundles disponibles dans le système.

```
	public static BundleContext	context	= null;
	private ComponentsManager	cm	= null;
	private Thread	componentThread = null;
```

Dans la méthode **start**, nous réalisons l'initialisation du CM et laissons CM lire le fichier ***world.***

```
public void start(BundleContext context) throws Exception {
		System.out.println("starting component manager....");
		Activator.context = context;

		//===SmartTools initialization===
		cm = new ComponentsManager();
		loadWorld("file:/user/hvdang/home/deploy/st.xml");

		componentThread = new Thread(cm.getProxy(), "Component manager");
		componentThread.start();
		System.out.println("Load world");


		//=====================OSGi initialization=================

		context.addServiceListener(this,
		  "(objectClass=fr.smarttools.core.component.ContainerService)");

}
```

Ajouter le méthode loadWorld pour lire le fichier world

```
void loadWorld(String worldFile) throws Exception{
		World w = new World();
		w.setWorldFile(worldFile);
		w.loadWorld();
		cm.setWorld(w);
}
```

Ajouter le code pour implémenter l'interface **ServiceListener** pour écouter les événements du framework OSGi.

```
public void serviceChanged(ServiceEvent event) {
	// TODO Auto-generated method stub
	System.out.println("CM Service changed");
	Object service= context.getService(event.getServiceReference());
	System.out.println(event.getType()+ service.getClass().getName());

	if (event.getType() == ServiceEvent.REGISTERED){
		System.out.println("new component added");

		cm.componentsLoaded.put(service);
	}
	else{
		System.out.println("unregistre");
		disconnect(service);
	}
}
```

La classe ComponentManager

Il y a quelques modifications dans cette classe par rapport à la classe origine de SmartTools.

Modifier la méthode ***run***, on ajoute le code pour faire les initialisations : lancer les bundles, créer les instances de composants et lier les instances.

```
	this.setIdName("ComponentsManager");			;
	this.getProxy().setContainerDescription(this.getContainerDescription());
	componentsStarted.put(getProxy());
	world.play(this);
```

Pour charger des components en mémoire, SmartTools utilise un class loader particulier. Dans OSGi, nous utilisons la méthode **install** et **start** de la classe **BundleContext** pour charger et démarrer un bundle.

Nous modifions le code pour charger des composants comme suit :

```
public void addComponent(String jarName,
                          String componentName,
                          String uriToJar) {

   try{
        Bundle cmp = Activator.context.installBundle(uriToJar);
        cmp.start();

   }
   catch(Exception e){
        logINFO("Cannot add components "+ componentName);
   }

        if (verbose.booleanValue())
           logINFO("done.\n");
   }
```

Nous modifions aussi la méthode **internalStartComponent** pour créer les instances d'un composant.

```
protected ContainerProxy internalStartComponent(String instance_id, String type) {
   // Look for the requested component <type>
   ContainerService cmp = (ContainerService)componentsLoaded.get(type);
   if (cmp != null) {
            ContainerProxy proxy = cmp.createComponent(instance_id);
           //registerNewComponent(proxy, cmp.getComponentDescription());
           new Thread(proxy, instance_id).start();
           return proxy;
    } else {
             logERROR("Unable to find component " + type +
                     " in component repository\n");
             return null;
        }
   }
```

3. *Réalisation de cmp1 et cmp2*

Pour réaliser le ***cmp1***(***cmp2***) nous réutilisons tous les codes générés par SmartTools. En plus nous ajoutons un **Activator** pour gérer le cycle de vie du composant

✓ ***La classe Activator***

Cette classe est l'activator du bundle. Elle implémente le service **ContainerService .**
Dans la méthode **start**, ajout du code pour initialiser le bundle : lire la description du composant et enregistrer le service ***ContainerService*** auprès du frame-work OSGi.

```
public void start(BundleContext context) throws Exception {
		// TODO Auto-generated method stub

		//===SmartTools initialization===
		System.out.println("Cmp1 started...");
		cmpDescription = new ComponentDescriptionImpl();
		String resourceFilename =  "/cmp1/resources/cmp1.cdml";
		cmpDescription.setResourceFilename(resourceFilename);
		cmpDescription.process();

		//===OSGi initialization=========
		this.context = context;
		registerService();

}
```

Ajouter le code pour implémenter l'interface **ContainerService** qui permet à **ComponentManager** de créer des instances de ce composant.

```
public ContainerProxy createComponent(String componentID) {
		// TODO Auto-generated method stub
		Cmp1Container container = new Cmp1Container();
		container.setIdName(componentID);
		container.setContainerDescription(cmpDescription);
		return container.getProxy();
}

public ComponentDescription getComponentDescription() {
		// TODO Auto-generated method stub
		return cmpDescription;
}
```

4. *Lancement d' application*

Les bundles nécessaires pour lancer l'application sont st-jar, st-core, CM, cmp1, cmp2. Après avoir créer les fichiers jar de ces bundles. Nous ajoutons dans le fichier ***system.properties*** (qui se trouve dans le répertoire **lib** d'Oscar) les 3 bundles st-jar, st-core et CM pour que le framework Oscar puisse lancer notre application automatiquement lors du lancement du framework.

```
oscar.auto.start.1=file:bundle/bundlerepository.jar
file:/user/hvdang/home/deploy/st-jar.jar file:/user/hvdang/home/deploy/st-core.jar \
file:/user/hvdang/home/deploy/cm.jar file:bundle/shell.jar file:bundle/shelltui.jar
```

Et voici le contenu du fichier ***world*** de notre application

```
<load_component* jar="cmp61.jar"
        url="file:/user/hvdang/home/deploy/cmp61.jar"
        name="cmp61"*/>*
<load_component* jar="cmp62.jar"
        url="file:/user/hvdang/home/deploy/cmp62.jar"
        name="cmp62"*/>

<connectTo id_src="ComponentsManager"      type_dest="cmp1" id_dest="cmp1-1"/>
<connectTo id_src="ComponentsManager"     type_dest="cmp2" id_dest="cmp2-1"/>
<connectTo id_src="ComponentsManager"     type_dest="cmp2" id_dest="cmp2-2"/>
<connectTo id_src="cmp1-1"           type_dest="cmp1" id_dest="cmp2-1"*/>
<connectTo id_src="cmp1-1"           type_dest="cmp1" id_dest="cmp2-2"/>
```

Evaluation: Avec ce modèle, chaque bundle n'est plus un composant mais un type de composant, le type de composant fournit les services pour instancier des composants et consulter les informations de ce composant. Le mécanisme de communication des instances de composants reste le même que SmartTools. Dans ce modèle l'OSGi ne joue que le rôle un support d'exécution des composant SmartTools.
En fait dans le version 4 de la spécification d'OSGi, elle a introduit un service appelé EventAdmin, Ce service fournit un mécanisme de communication d'inter-bundles basé sur le modèle de « publish and subscribe » des événements. En employant ce service, on peut réaliser la communication asynchrone entre les composants SmartTools.

Vérification du modèle : Pour vérifier le modèle proposé au-dessus (figure 13), nous avons réalisé une version simple de SmartTools. Cette version se compose des composants suivants :

- Deux composants de base de SmartTools : ***st-core, st-jar***
- Le gestionnaire de composants CM
- Les composants cmp1 et cmp2 ;.
- Un composant document ***lml*** : ***lml*** est un langage basé sur XML décrivant la structure de l'interface d'utilisateur. Le composant document ***lml*** fournit ensemble des outils pour manipuler des documents ***lml***.
- Un composant vue ***glayout*** : C'est le composant qui est chargé de transformer la descriptive d'un document ***lml*** en des objets graphiques en utilisant une feuille de style pour construire l'interface d'utilisateur. Ce composant utilise des services fournis par le composant ***lml*** pour manipuler les documents ***lml***.
- Un composant Graphe qui montre la représentation graphique des composants disponibles dans notre application et des interconnexions entre eux sous forme une graphe

Voici, le contenue du fichier de lancement de notre application (fichier ***world***).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<world repository="file:/user/hvdang/home/deploy/"
       debug="ON">
```

```
   <!--                          -->
   <!-- Components declaration -->
   <!--                          -->
   <load_component jar="cmp1.jar"
                url="file:/user/hvdang/home/deploy/cmp61.jar"
                name="cmp1"/>
   <load_component jar="cmps2.jar"
                url="file:/user/hvdang/home/deploy/cmp62.jar"
                name="cmp2"/>
   <load_component jar="graph.jar"
                url="file:/user/hvdang/home/deploy/graph.jar"
                name="graph"/>
   <load_component jar="glayout.jar"
                url="file:/user/hvdang/home/deploy/glayout.jar"
                name="glayout"/>
   <load_component jar="lml.jar"
                url="file:/user/hvdang/home/deploy/lml.jar"
                name="lml"/>
   <!--         -->
   <!-- connect -->
   <!--         -->
   <connectTo id_src="ComponentsManager"
      type_dest="worldStateGraph" id_dest="GraphContainer"/>

   <connectTo id_src="ComponentsManager"  type_dest="cmp1"
id_dest="cmp1-1"/>
   <connectTo id_src="ComponentsManager"  type_dest="cmp2"
id_dest="cmp2-1"/>
   <connectTo id_src="ComponentsManager"  type_dest="cmp2"
id_dest="cmp2-2"/>
   <connectTo id_src="ComponentsManager"  type_dest="lml"
id_dest="lml1"/>

   <connectTo id_src="cmp1-1"  type_dest="cmp1" id_dest="cmp2-1"/>
   <connectTo id_src="cmp1-1"  type_dest="cmp1" id_dest="cmp2-2"/>


   <connectTo id_src="ComponentsManager"
         type_dest="glayout" id_dest="glayout1">
      <attribute name="docRef"
               value="file:/user/hvdang/home/deploy/empty.lml"/>
      <attribute name="xslTransform"
               value="resources:lml/xsl/lml2bml.xsl"/>
      <attribute name="behaviors"
               value="resources:stcore/behaviors/bootbehav.xml"/>
      <message name="initData">
            <attribute name="inits">
                  <collection>
                        <item name="behavior"
value="resources:cm/cmbehaviors.xml"/>
                  </collection>
            </attribute>
      </message>
   </connectTo>

</world>
```

La représentation graphique de ce fichier est montrée par le composant ***Graph*** lorsque l’application est lancée dans l’OSCAR :

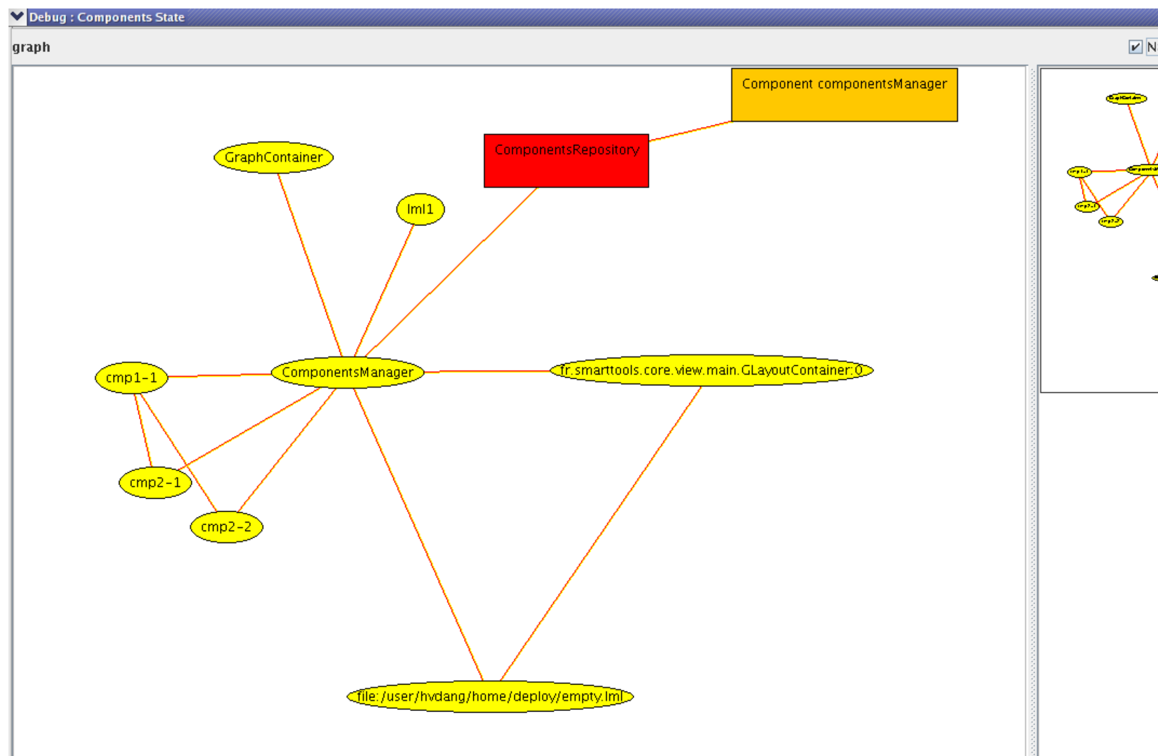


Figure 15 : Une vue graphiques des liaisons entre des instances des composants

L’interface d’utilisateur de notre application est décrit par un document ***lml***, voici c’est le contenu de ce fichier :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:lml.dtd">
<layout>
  <frame title="SmartTools" statusBar="on"
         width="900"
         height="700"
         dynTabSwitch="off">
  </frame>
</layout>
```

Et le résultat lors d’exécution de notre application :

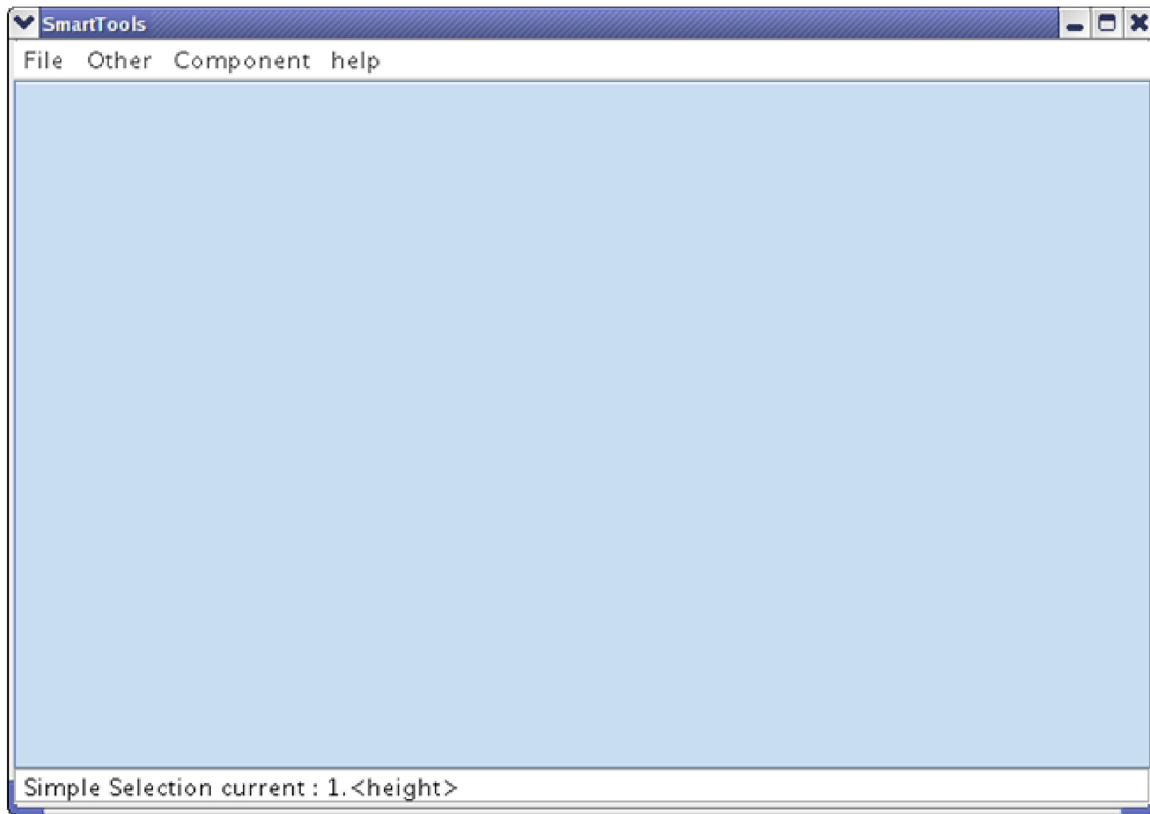


Figure 16: Une version simple de SmartTools au-dessus OSGi

Cette application ne comporte pas tous les composants de SmartTools. Cependant elle est construite avec la plupart des composants de base de SmartTools (Le noyau de SmartTools, le gestionnaire de composants, un composant document et un composant vue). Il est donc possible de construire un version complète de SmartTools en transformant les autres composants de SmartTools en des bundles OSGi (en appliquant la même technique présentée au-dessus). Le résultat montre que le modèle proposé (figure 13) instancie convenablement l'architecture orientées-services de SmartTools au-dessus OSGi.

6. Conclusion et perspectives

Actuellement la plate-forme OSGi devient de plus en plus un standard. Elle est soutenu par des grandes entreprises informatiques, des entreprises automobiles, les fabricants de téléphones mobiles, etc.. Eclipse a adopté OSGi comme la plate-forme de base pour ses plug-ins et Apache utilise maintenant OSGi pour ses serveurs[3]

Comme une plate-forme dynamique de service, OSGi définit un ensemble de mécanismes permettant de déployer, de découvrir et de gérer des services à l'intérieur de la plate-forme (appelé le frame-work). Cependant tous les notions liés aux services d'OSGi sont encore très élémentaires (les services sont définis comme interfaces Java, il manque des mécanismes de communication effectivement entre des services, les liaisons entre des services sont fixées etc..) Cela ne permet pas à la plate-forme OSGi de devenir une vraie plat-forme orientée services.
La transformation de l'architecture à composants orienté services de SmartTools[1] au-dessus la plate-forme OSGi nous permet d'ajouter à OSGi les notions de l'architecture orientée services de SmartTools en profitant de tous les avantages (la fiabilité, la dynamité, etc..) et les standards de OSGi. Une version de SmartTools au-dessus OSGi est une démarche importante pour l'intégration de SmartTools dans Eclipse.

A travers de plusieurs d'expérimentations, nous avons réussi à introduire un modèle pour transformer l'architecture à composants orienté services de SmartTools au-dessus OSGi. Dans ce modèle, OSGi joue le rôle d'un support d'exécution des composants SmartTools. Les bundles sont considérés comme des types de composants, OSGi fournit les mécanismes permettent de gérer des types de composants et de créer des instances de composants. Et pour vérifier le modèle, nous avons réalisé une application simple de SmartTools avec un composant de Graphe, un composant vue et un composant document (Le figure 16 montre le résultat de l'application). Cette application montre que la transformation de SmartTools au-dessus OSGi est réalisable.

A partir du modèle proposé, nous avons réussit à introduire dans OSGi une architecture orienté services définie par SmartTools. Cela permet d'enrichir la spécification OSGi pour définir une vrai plate-forme orientée services. Ce modèle nous permet aussi de construire une version simple de SmartTools dans OSGi.

Cependant le modèle reste encore simple. Il ne profite pas encore de toutes les facilités fournies par la plat-forme OSGi. En fait, la spécification OSGi R4 a introduit un service supplémentaire nommé ***EventAdmin*** qui permet aux bundles de communiquer en envoyant des événements. Ce type de communication nous permettra de réaliser le mécanisme de communication asynchrone de SmartTools dans OSGi. Il nous permettra de profiter le plus possible des capacités de OSGi pour transformer l'architecture de SmartTools. A l'heure actuel, le projet Equinox (Une implémentation OSGi d'Eclipse) a introduit une version stable du service ***EventAdmin***, donc les travaux que nous pouvons faire dans l'avenir sont :

- Réaliser le mécanisme de communication de SmartTools dans OSGi en employant le service EventAdmin.
- Faire des études sur l'architecture des plug-ins d'Eclipse afin de intégrer SmartTools dans Eclipse.

7. Références

[1]
Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. **Un modèle abstrait de composants adaptables**. ***revue TSI, Composants et adaptabilité***, 23(2), 2004.

[2]
Cours d'OSGi de Didier Donsez [http://www-adele.imag.fr/users/Didier.Donsez/cours/osgi.pdf]

[3]
Le siteweb du consortium d'OSGi [www.osgi.org]

[4]
Le siteweb du projet Service Binder [http://gravity.sourceforge.net/servicebinder]

[5]
Thèse de Humberto CERVANTES. **Vers un modèle a composants orienté services pour supporter la disponibilité dynamique**, UNIVERSITÉ JOSEPH FOURIER, 2004

[6] Le siteweb d'OSCAR [http://oscar.objectweb.org/]

8. Annexe

6.1. Guide d'utilisation OSCAR

Installation: Pour installer OSCAR, Il faut télécharger le Jar d'installation d'OSCAR depuis le site http://oscar.objectweb.org puis le lancer avec la commande "java -jar oscar-1.0.5.jar". puis suivre les instructions pour compléter l'installation.

Lancement le framework: Dans le répertoire d'installation d'Oscar, lancez la commande suivante: <répertoire d'installation d'Oscar >/oscar.sh

```
[hvdang@smart2 Oscar]$ ./oscar.sh

Welcome to Oscar.
=================

Enter profile name: abc

-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (1.0.5)
[   1] [Active     ] [    1] Shell Service (1.0.2)
[   2] [Active     ] [    1] Shell TUI (1.0.0)
[   3] [Active     ] [    1] Bundle Repository (1.1.2)
->
```

Manipulation:

- Pour lister les bundles et leurs états: lancez la commande **ps**

- Installer un bundle
  Lancez la commande ***install <chemin vers le fichier jar du bundle>***
  Example: **install** file:/user/hvdang/home/deploy/servicebinder.jar

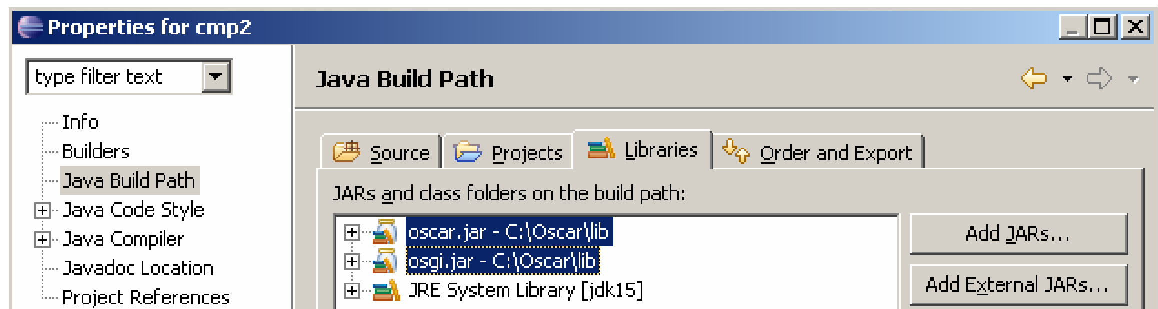
- Activer un bundle déjà installer
  Lancez la commande ***start <bundle ID>.*** L*e bundle ID est obtenu en utilisant la commande ps)*
- Déactiver un bundle: ***stop* <Bundle ID>**
- Déinstaller un bundle: ***uninstall* <Bundle ID>**
- Arrêter le framework: ***shutdown***

6.2. Expérimentation 1 - réalisation le bundle cmp2

Démarche:

- Dans Eclipse, créez un projet vide avec le nom cmp2. Choissisez aussi l'option “Create separate source and output folders”

- Copiez le fichier Cmp2App.java du composants cmp2-com dans le répertoire contrib de SmartTools au répertoire src du projet cmp2 dans le workspace d'Eclipse
- Ouvriez la page de propriété du projet, dans l'onglet "Libraries" ajoutez deux librairies osgi.jar et oscar.jar, ces deux libraries se trouvent dans le répertoire lib du répertoire d'installation d'Oscar.



- Créez l'interface UpdateService comme suivante

```
package cmp2;
public interface UpdateService
{
      public void updateLabel(String message);
}
```

- Créez la classe Cmp2Activator qui implante l'interface BundleActivator d'OSGi, dans la méthode **start**, enregistrez le service **UpdateService**:

```
public void start(BundleContext context)
    {
          context.registerService(
          UpdateService.class.getName(), new Cmp2App(), null);
    }
```

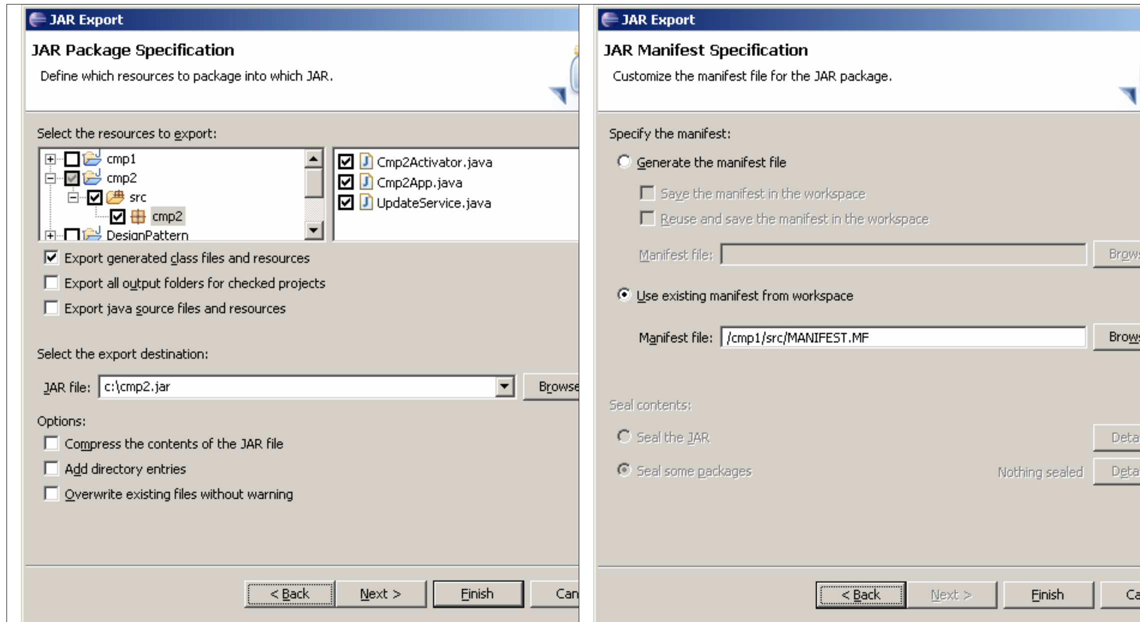
- Modifiez la class Cmp2App

```
public class Cmp2App implements UpdateService{
…
}
```

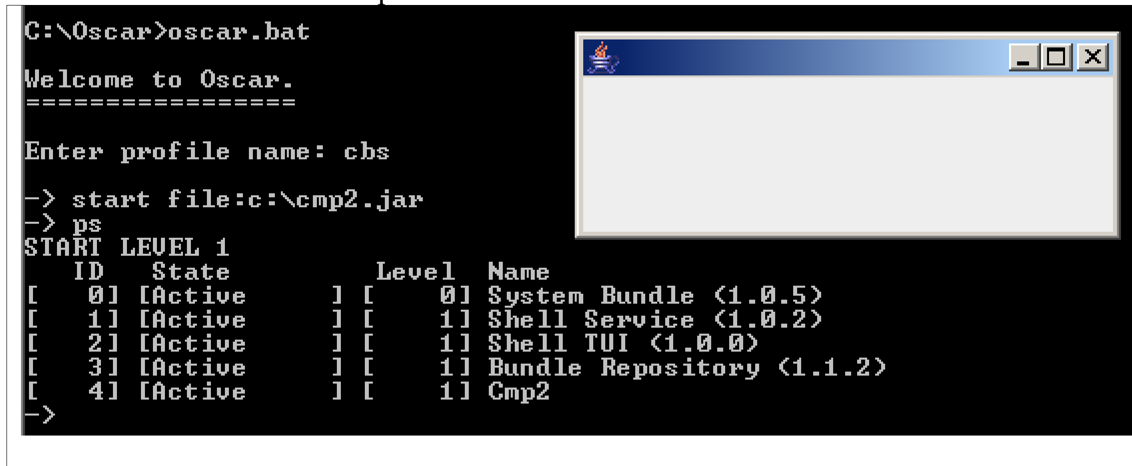
- Ajoutez le fichier MANIFEST.MF avec le contenu comme suivant:

```
Manifest-Version: 1.0
Bundle-Activator: cmp2.Cmp2Activator
Export-Package: cmp2
Bundle-Name: Cmp2
Bundle-Description: A bundle for testing
Bundle-Vendor: VuDang
Bundle-Version: 1.0.0
```

- Créez le fichier de déploiment du bundle en éxpotant le projet en fichier jar, choisiez le fichier MANIFEST.MF comme le fichier manifest du fichier jar exporté



- Lancez le bundle cmp2



6.3. Expérimentation 1 - réalisation le bundle cmp1

Démarche:

- Dans Eclipse, créez un projet vide avec le nom cmp1. Choisissez aussi l'option “Create separate source and output folders”

- Copiez le fichier Cmp1App.java du composant cmp1-com dans le répertoire contrib de SmartTools au répertoire src du projet cmp1 dans le workspace d'Eclipse
- Ouvriez la page de propriété du projet, dans l'onglet "Libraries" ajoutez deux librairies **osgi.jar** et **oscar.jar**, ces deux librairies se trouvent dans le répertoire lib du répertoire d'installation d'Oscar
- Créez la classe Cmp1Activator qui implante l'interface BundleActivator d'OSGi, dans la méthode **start**, ajoutez le code pour chercher le service **UpdateService** fourni par cmp2:

```
public void start(BundleContext context)
    {
        ServiceReference ref = context.getServiceReference(
                            Cmp2AppFacadeInterface.class.getName());

        if(ref != null)
        {
            cmp2Service                                                    =
(Cmp2AppFacadeInterface)context.getService(ref);
        }
        else{
            System.out.println("Service cmp2 not found");
        }
}
```

- Dans la classe Cmp1Activator, ajoutez la méthode **send**(String message) pour servir le requête de la partie métier Cmp1App:

```
public void send(String message)
  {
        cmp2Service.updateLabel(message);
  }
```

- Dans la classe Cmp1App, modifiez le constructeur comme suivant

```
public Cmp1App(Cmp1Activator container) {
        this.container = container;

        JButton button = new JButton("Press Me");
        JLabel label = new JLabel("Text");

}
```

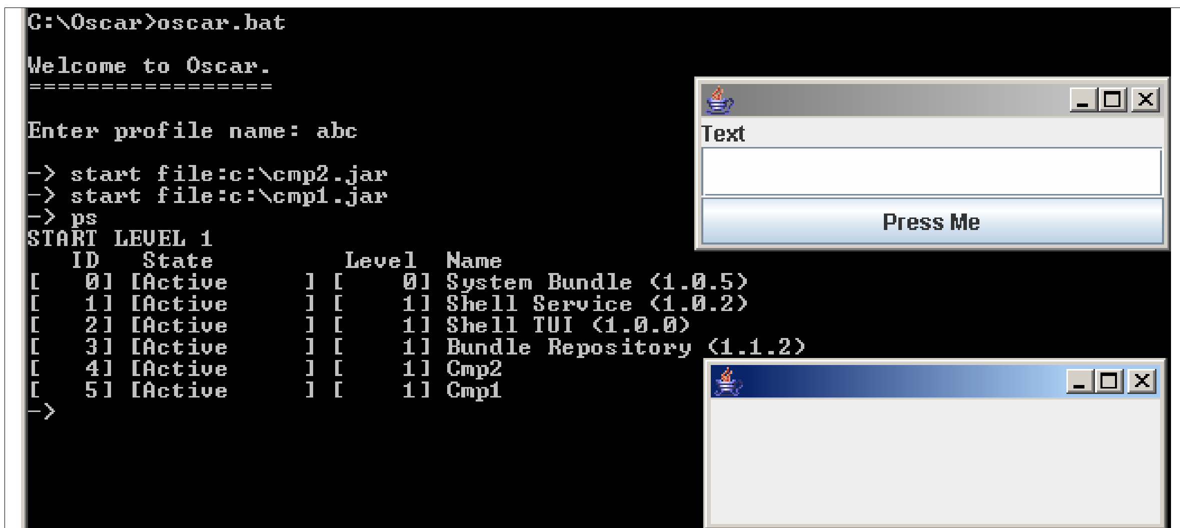
- Modifiez la méthode updateLabel de la classe Cmp1App pour demander le service du bundle cmp2

```
public void updateLabel(String message) {
        container.send(message);
    }
```

- Ajoutez le fichier MANIFEST.MF avec le contenu comme suivant:

```
Manifest-Version: 1.0
Bundle-Activator: cmp1.Cmp1Activator
Export-Package: cmp1
Import-package: cmp2
Bundle-Name: Cmp1
Bundle-Description: A bundle for testing
Bundle-Vendor: VuDang
Bundle-Version: 1.0.0
```

- Créez le fichier de déploiement du bundle en exportant le projet en fichier jar, choissiez le fichier MANIFEST.MF comme le fichier manifest du fichier jar exporté (comme fait dans le projet précédent)
- Lancez le bundle cmp1



6.4. Expérimentation 2 - réalisation le bundle cmp2

Démarche:

- La démarche pour réaliser le composant cmp2 dans cette expérimentation est identique à celle de l'expérimentation 1

6.5. Expérimentation 2 - réalisation le bundle cmp1

Démarche:

- Dans Eclipse, créez un projet vide avec le nom cmp1. Choisissez aussi l'option "Create separate source and output folders"
- Créez un package cmp dans le répertoire src

- Copiez les fichiers suivants du composant cmp1-com dans le répertoire **contrib** de **SmartTools** au répertoire **src/cmp1.**
  - ✓ Cmp1App.java
  - ✓ Cmp1AppFacade.java
  - ✓ Cmp1AppFacadeInterface.java

- Ajoutez l'interface **UpdateLabelListener.java**, c'est l'écouteur (listener) de l'événement ***UpdateLabel***

```
public interface UpdateLabelListener {
   //
   // Methods
   //

   /**
    * updateLabel
    * update Label
    * @param ev a <code>UpdateLabelEvent</code> value : event
    **/
   public  void updateLabel(UpdateLabelEvent ev);
}
```

- Ajoutez la classe **UpdateLabelEvent.java**

```
public class UpdateLabelEvent extends StEventImpl
   protected java.lang.String message;

   /**
    **/
   public void setMessage(java.lang.String v){
      this.message = v;
   }

   public java.lang.String getMessage(){
      return message;
   }
   /**
    * Constructor
    **/
   public   UpdateLabelEvent(java.lang.String message){
      setMessage(message);
   }
}
```

- Créez la classe **Cmp1Container.java** qui implante l'interface **BundleActivator** d'OSGi et l'interface **UpdateLabelListener.** Dans cette classe on déclare un instance de la façade.

```
    /**
     * Facade Object
     **/
    public cmp1.Cmp1AppFacade facade;
```

- Dans le constructeur, on ajoute l'écouteur pour la façade :

```
public   Cmp1Container(){
      try {
         facade = new cmp1.Cmp1AppFacade();
      ((Cmp1AppFacadeInterface)
facade).addUpdateLabelListener(this);
      } catch (Exception e){
         e.printStackTrace();
      }
    }
```

- On réalise l'action lors que l'événement UpdateLabel est envoyé par la façade :

```
public  void updateLabel(UpdateLabelEvent e){
      Message m = new MessageImpl("updateLabel",
e.getAttributes() , null);
      //===call static
      HashMap args = (HashMap)e.getAttributes();
      String msg = (String)args.get("message");
      send(msg);
     }
```

- On définit une méthode **send** pour envoyer le message au composant cmp2

```
private void send(String msg){
//reference to service of cmp2
            cmp2.Cmp2AppFacadeInterface service2 = null;
            //the context is valid???
            if(context==null){
                  System.out.println("Connext is null");
                  return;
            }

            System.out.println("Get service of cmp2");
            try
            {
                  // Query for all service references matching
any language.
                  ServiceReference ref =
context.getServiceReference(

      cmp2.Cmp2AppFacadeInterface.class.getName());

            if (ref == null){
                  System.out.println("Service not found");
            }
            else{
                  System.out.println("Update text:"+ msg);
                  service2=
(cmp2.Cmp2AppFacadeInterface)context.getService(ref);
                        service2.updateLabel(msg);
                  }
            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }

    }
```

- Créez des fichiers jar de ces deux bundles et les déploriez dans l'Oscar comme fait dans l'expérimentation 1

6.6. Expérimentation 4 - réalisation le bundle cmp2

Démarche:

- Créez le projet cmp2 avec tous les configurations et les codes sources du projet cmp2 dans expérimentation 2
- Ajoutez la classe **ServiceObserver.java** qui est chargé d'écouter des événements de service comme arrivé de service ou départ de service.

```
public class ServiceObserver  implements ServiceListener {

      private Cmp1Container container;

      ServiceObserver(Cmp2Container container,String
serviceName){
            this.container = container;
            try{

      container.getContext().addServiceListener(this,
                        "(objectClass=" + serviceName + ")");

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

public void serviceChanged(ServiceEvent event) {
            // TODO Auto-generated method stub
            container.serviceChanged(event);
      }
}
```

- Modifier le code de **Cmp2Container.java**
  - ✓ La classe Cmp2Container implémente l'interface BundleActivator et l'écouter **ServiceListener**

```
public class Cmp2Container implements BundleActivator,
ServiceListener {…}
```

  - ✓ Ajoutez les variables qui stockent les instances des services fournit et services requis du bundle

```
/**
 * Require services and provide service
 */
protected BundleContext context = null;
protected ArrayList requireServices = new ArrayList();
protected ArrayList requireServiceNames = new ArrayList();
protected ArrayList provideServiceNames = new ArrayList();
```

  - ✓ Dans le méthode start, On ajoute le code pour enregistrer les services fournit et récupérer la liste des services requis. Tous ces types de services sont décrits dans le fichier description metadata.xml. Pour chaque service requis, on crée son écouteur correspondant. Cet écouteur notifiera lors que son service est disponible ou lors que son service est terminé.

```
public void start(BundleContext context) throws Exception {
   this.context = context;

   registerService();
   getRequireServices();


   //listenning events of services
  for (int i = 0; i < requireServiceNames.size(); i++) {
     context.addServiceListener(this, "(objectClass=" +
(String)requireServiceNames.get(i) + ")");

   }

}
```

- ✓ Ajoutez le code pour lire le fichier description
- ✓ Ajoutez le code pour traitez les événements de services du frame-work

```
/**
 * Service Listener
 */
public void serviceChanged(ServiceEvent event) {

      Object service=
context.getService(event.getServiceReference());
      if (event.getType() == ServiceEvent.REGISTERED){
              if(!isServiceExist(service))
                    requireServices.add(service);
      }
      else{
              if(isServiceExist(service))
                    requireServices.remove(service);
      }
}
```

- Dans le répertoire **src** du projet, ajoutez le fichier **metadata.xml** qui fournit la description des service requis et des service fournis

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="cmp2.Cmp2AppFacade">
    <provides service="cmp2.Cmp2AppFacadeInterface"/>
    <property name="version" value="1.0.0" type="string"/>
  </component>
</bundle>
```

- Dans le fichier MANIFEST.MF, ajoutez cette ligne à la fin du fichier

```
Manifest-Version: 1.0
Bundle-Activator: cmp2.Cmp2Container
Export-Package: cmp2
DynamicImport-Package: *
Require-Bundle: st_jar,st_core
Bundle-Name: Cmp2
Bundle-Description: A bundle for testing
Bundle-Vendor: VuDang
Bundle-Version: 1.0.0
Metadata-Location: metadata.xml
```

- Recompilez le projet et créez le bundle en utilisant l'outil « **export**« de l'Eclipse

6.7. Expérimentation 4 - réalisation le bundle cmp1

Démarche:

- Créez le projet **cmp1** avec tous les configurations et les codes sources du projet **cmp1** dans **l'expérimentation 2**
- Ajoutez la classe **ServiceObserver.java** qui est chargé d'écouter des événements de service comme arrivé de service ou départ de service.

```
public class ServiceObserver  implements ServiceListener {

private Cmp1Container container;

ServiceObserver(Cmp1Container container,String serviceName){
      this.container = container;
      try{
      container.getContext().addServiceListener(this,
                        "(objectClass=" + serviceName + ")");

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

public void serviceChanged(ServiceEvent event) {
            // TODO Auto-generated method stub
            container.serviceChanged(event);
}
}
```

- Modifier le code de **Cmp1Container.java**

- ✓ La classe Cmp1Container implémente l'interface BundleActivator et l'écouter **ServiceListener**

```
public class Cmp1Container implements BundleActivator,
ServiceListener {…}
```

- ✓ Ajoutez les variables qui stockent les instances des services fournit et services requis du bundle

```
/**
 * Require services and provide service
 */
protected BundleContext context = null;
protected ArrayList requireServices = new ArrayList();
protected ArrayList requireServiceNames = new ArrayList();
protected ArrayList provideServiceNames = new ArrayList();
```

- ✓ Dans le méthode ***start***, On ajoute le code pour enregistrer les services fournit et récupérer la liste des services requis. Tous ces types de services sont décrits dans le fichier description metadata.xml. Pour chaque service requis, on crée son écouteur correspondante. Cet écouteur notifiera lors que son service est disponible ou lors que son service est terminé.

```
public void start(BundleContext context) throws Exception {
   this.context = context;

   registerService();
   getRequireServices();


   //listenning events of services
  for (int i = 0; i < requireServiceNames.size(); i++) {
     context.addServiceListener(this, "(objectClass=" +
(String)requireServiceNames.get(i) + ")");

   }

}
```

- ✓ Ajoutez le code pour lire le fichier description

- ✓ Ajoutez le code pour traitez les événements de services du frame-work

```
/**
 * Service Listener
 */
public void serviceChanged(ServiceEvent event) {

      Object service=
context.getService(event.getServiceReference());
      if (event.getType() == ServiceEvent.REGISTERED){
              if(!isServiceExist(service))
                    requireServices.add(service);
      }
      else{
              if(isServiceExist(service))
                    requireServices.remove(service);
      }
}
```

✓ Ajoutez le code pour envoyer le message UpdateLabel

```
 public  void updateLabel(UpdateLabelEvent e){
         if(requireServices.size() == 0){
               System.out.println("No available service ");
               return;
         }

    Message m = new MessageImpl("updateLabel",
                  e.getAttributes() , null);
    //===call static
    HashMap args = (HashMap)e.getAttributes();
    String msg = (String)args.get("message");


    //get service who has the method updateLabel
    for(int i = 0;i<requireServices.size();i++){
        Class c = requireServices.get(i).getClass();
        try
        {
              Method[] mList = c.getMethods();
              for (int j = 0; j < mList.length; j++) {

if(isThisMethod(mList[j],"updateLabel",void.class,new
Class[]{String.class})){

mList[j].invoke(requireServices.get(i), new Object[] { msg});
                    }
              }
        }
        catch(Exception ex){
             ex.printStackTrace();
        }
      }
}
```

- Dans le répertoire **src** du projet, ajoutez le fichier **metadata.xml** qui fournit la description des services requis et des service fournis.

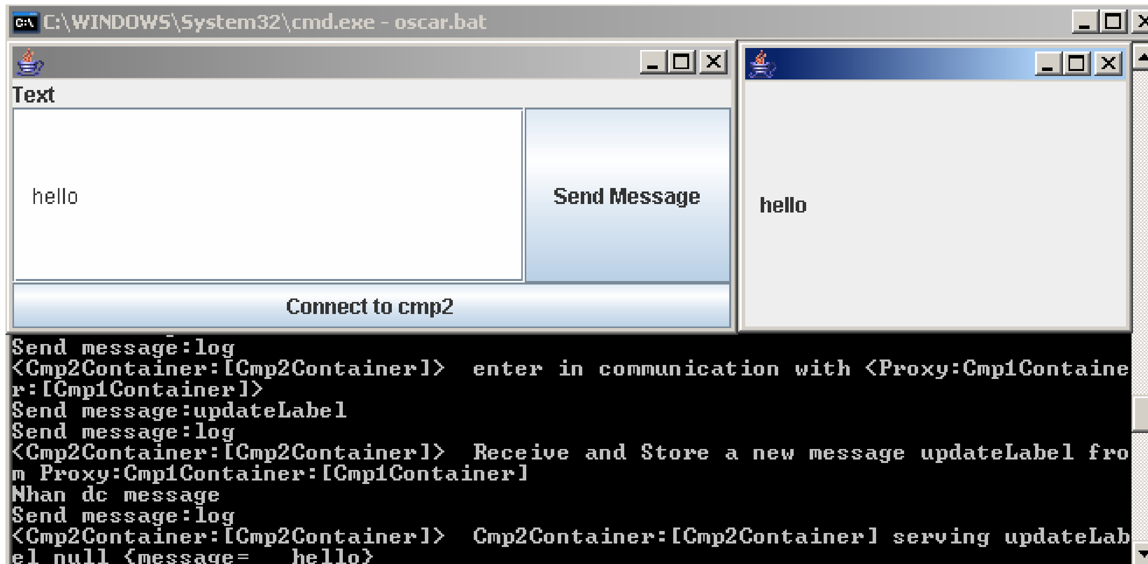
```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="cmp2.Cmp2AppFacade">

        <requires service="cmp2.Cmp2AppFacadeInterface"/>
    <property name="version" value="1.0.0" type="string"/>
  </component>
</bundle>
```

- Modifiez le fichier MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-Activator: cmp1.Cmp1Container
Export-Package: cmp1
Import-package:
fr.smarttools.core.util,fr.smarttools.core.component
Require-Bundle: st_jar,st_core
Bundle-Name: Cmp41
Bundle-Description: A bundle for testing
Bundle-Vendor: VuDang
Bundle-Version: 1.0.0
Metadata-Location: metadata.xml
```

- Recompilez le projet et créez le bundle en utilisant l’outil « **export**« de l’Eclipse

Après d’avoir créé deux bundles cmp1 et cmp2, on installe et démarre 4 bundles suivants dans l’Oscar : st-jar, st-core, cmp1, cmp2. Le résultat est le suivant :



6.8. *La liste des fichier modifiés*

Modification de st-core.

- ✓ Ajouter la classe **STGenericActivator** dans le package **component**
- ✓ Ajouter la classes **ContainerService** dans le package **component**
- ✓ Modifier de façon d'accèss aux résources: Car dans OSGi, la classe URL ne marche pas avec les fichier dans les jars, donc il faut modifier le méthode **getURLForFile** de la classe **StUtilities (fichier StUtilities.java)** Dans la hierachie CVS de SmartTools, on créer un répertoire recources qui contient tous les resources de tous les composants de SmartTools.
- ✓ Modifier le méthode **internalInit** de la classe **AbtractContainer** (fichier AbtractContainer.java)
- ✓ Modifier le méthode **extractData** et la méthode **load** de la classe **ComponentDescriptionImpl** (fichier ComponentDescriptionImpl.java)
- ✓ Modifier le constructor de la classe **Gdocument** (fichier Gdocument.java)
- ✓ Copier deux fichier **abtractView.cdml** (se trouve dans le package **view/resources**) et **logicaldocument.cdml**(se trouve dans le package **document/resources**) au package **component/recources**

Modification de ComponentManager

- ✓ Modifier le méthode **run**, **extractDocumentID**, **foundDocType de la classe ComponentManager** (fichier ComponsantManger.java)
- ✓ Ajouter la classe **Activator** de CM (l'activator du CM est différent de l'activator d'un composant normal).