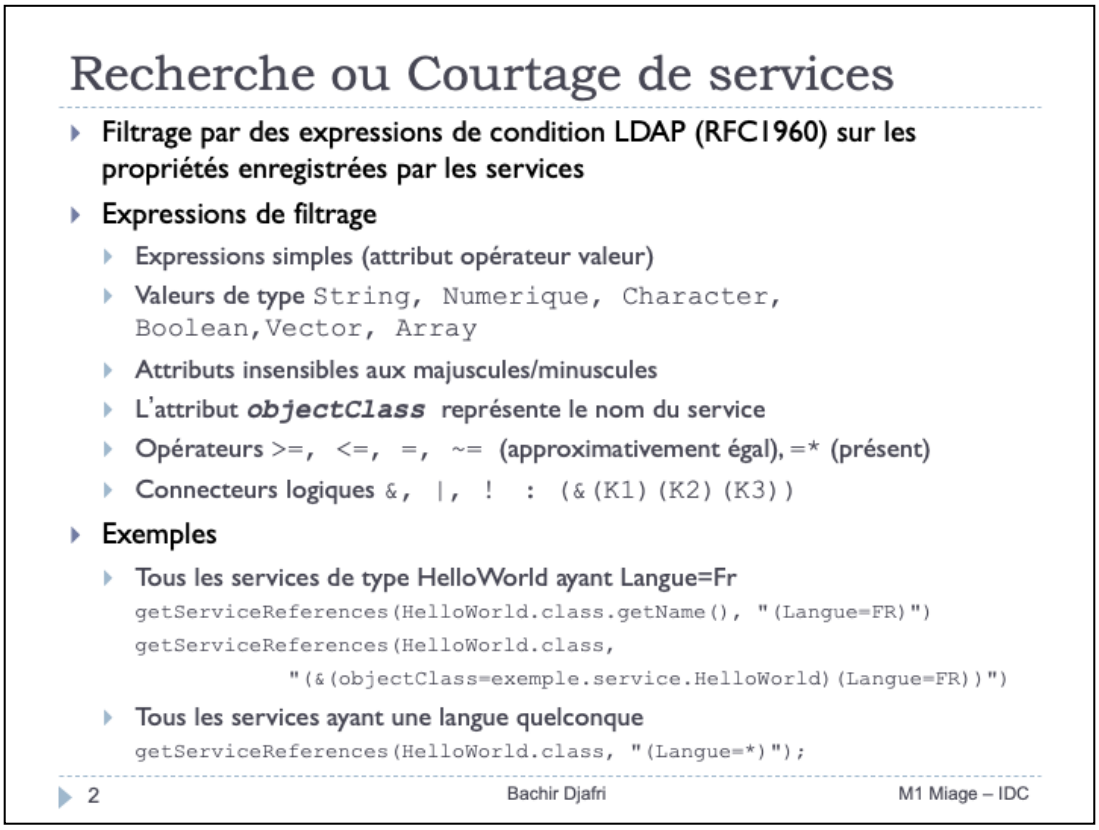
Cours 2.1

Premiers pas avec OSGi (suite – Les écouteurs)

La plate-forme dynamique orientée composants

Bachir Djafri



Basic LDAP Filter Syntax and Operators

LDAP filters consist of one or more criteria. If one than more criterion exist in one filter definition, they can be concatenated by logical **AND** or **OR** operators.

The logical operators are always placed in front of the operands (i.e. the criteria). This is the so-called 'Polish Notation'.

The search criteria have to be put in parentheses and then the whole term has to be bracketed one more time.

AND Operation:

(& (...K1...) (...K2...)) or with more than two criteria: (& (...K1...) (...K2...) (...K3...) (...K4...))

OR Operation:

(| (...K1...) (...K2...)) or with more than two criteria: (| (...K1...) (...K2...) (...K3...) (...K4...))

Nested Operation:

Every AND/OR operation can also be understood as a single criterion:

(|(& (...K1...) (...K2...))(& (...K3...) (...K4...))) means: (K1 AND K2) OR (K3 AND K4)

The search criteria consist of a requirement for an LDAP attribute, e.g. (givenName=Sandra). Following rules should be considered:

Equality: (*attribute*=abc) , e.g. *(&(objectclass=user)(displayName=Foeckeler)*

Negation: (!(*attribute*=abc)) , e.g. *(!objectClass=group)*

Presence: (*attribute*=*) , e.g. *(mailNickName=*)*

Absence: (!(*attribute*=*)) , e.g. *(!proxyAddresses=*)*

Greater than: (*attribute*>=abc) , e.g. *(mdbStorageQuota>=100000)*

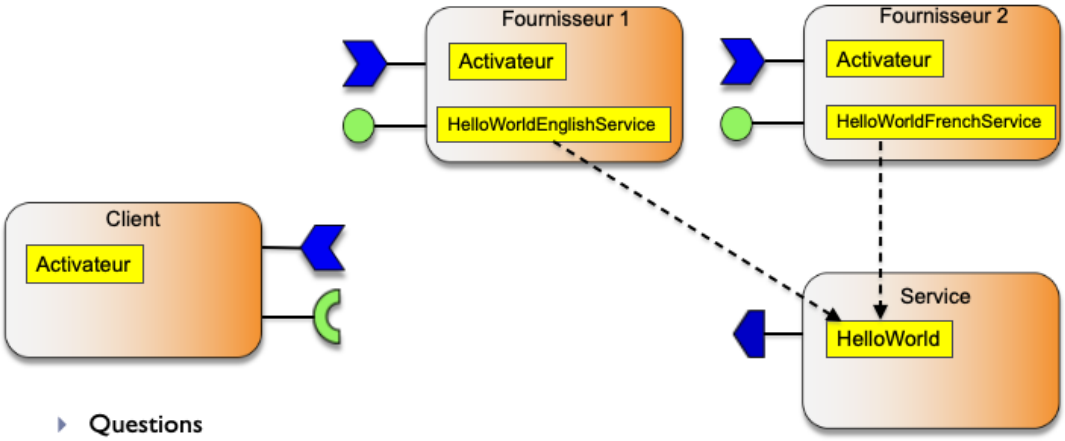
Less than: (*attribute*<=abc) , e.g. *(mdbStorageQuota<=100000)*

Proximity: (*attribute*~=abc) , e.g. *(displayName~=Foeckeler)* **Caution: ~= is treated as = in ADS environments !!**

Wildcards: e.g. (sn=F*) or (mail=*@cerrotorre.de) or (givenName=*Paul*)

Exemple 3 : extension de l'exemple 2

- Un 2nd fournisseur fournit un service satisfaisant le contrat « HelloWorld »



- Questions
  - Quel est le comportement dynamique du client quand les 2 fournisseurs sont actifs avant lui ?
  - Si le client utilise le 1er fournisseur et que celui-ci est désactivé, que se passe t-il ?
  - Peut-il utiliser le second fournisseur ? Si oui, comment ?

Exemple 3

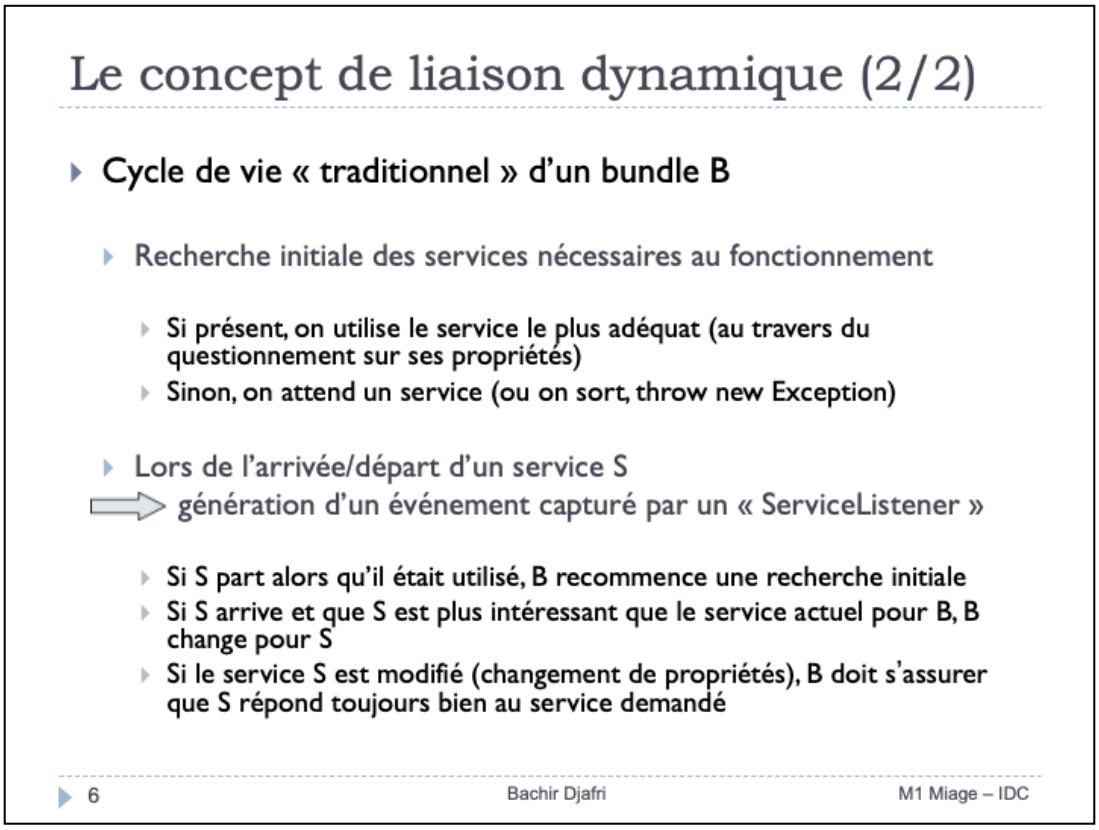
- Pour avoir un bundle dynamique (réactif) aux modifications du contexte, il faut que le bundle soit à l'écoute de ces modifications du contexte
- La première solution consiste à utiliser des Ecouteurs
  - Ecouteurs de services;
  - Ecouteurs de composants (bundles);
  - Ecouteurs d'environnement (Framework);
  - Autres écouteurs.
- Application du patron Observateur/Observable
  - Le composant client = observateur;
  - L'environnement (context) = observable / observé;
  - Le client reçoit les événements envoyés par l'environnement selon le type de l'écouteur enregistré auprès de l'environnement.

Le concept de liaison dynamique (1/2)

- **OSGi permet à ses composants (bundles) de sélectionner à l'exécution le composant qui fournira le service requis (s'il y a plusieurs fournisseurs)**

- **Ceci induit un comportement dynamique fort**
  - Un bundle attend que les services dont il a besoin soient actifs pour être lui-même actif
  - Il peut dynamiquement changer de service s'il trouve un service plus « performant »
  - La disparition d'un bundle est aussi gérée

- **Ce dynamisme est géré au travers d'évènements qui sont**
  - Le départ ou l'arrivée d'un bundle — interface BundleListener
  - Le départ ou l'arrivée d'un service — interface ServiceListener
  - Relatif à la plateforme (environnement) — interface FrameworkListener
  - Autres — interfaces LogListener, …

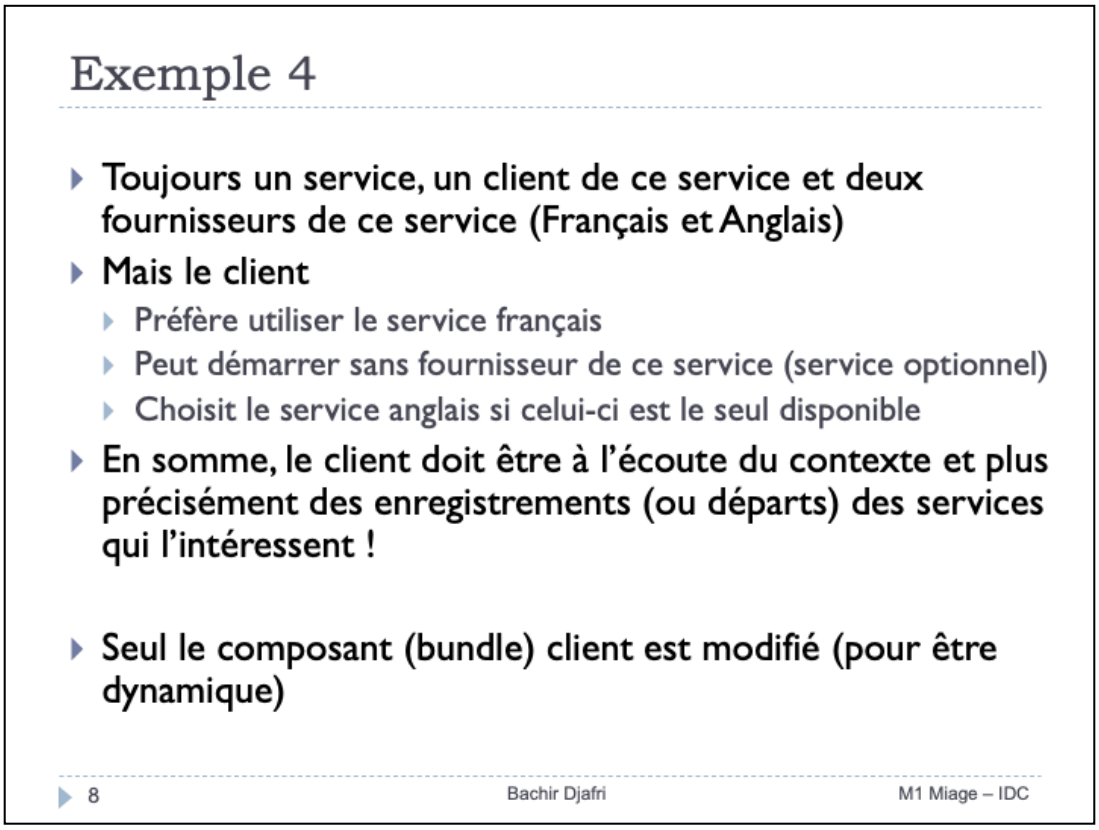


Dans le cas d'une interface d'un service requis obligatoire (nécessaire), si ce dernier est absent, il faut quitter la méthode start() avec une exception.

Une boucle d'attente risque de bloquer touts les autres bundles (composants).

Les trois catégories d'évènements

- FrameworkEvent
  - Notifie le démarrage et les erreurs du Framework (Felix, Equinox, etc.)
  - Interface `FrameworkListener`, Méthode `frameworkEvent`
    - Traitement séquentiel et asynchrone des listeners (par event dispatcher)
- BundleEvent
  - Notifie les changements dans le cycle de vie des bundles
  - Interface `BundleListener`, méthode `bundleChanged`
    - Traitement séquentiel et asynchrone des listeners (par event dispatcher)
  - Interface `SynchronousBundleListener`, méthode `bundleChanged`
    - Traitement séquentiel et synchrone des listeners (avant le traitement du R2 changement d'état)
- ServiceEvent (traité dans ce cours)
  - Notifie l'enregistrement ou le retrait de services
  - Interface `ServiceListener`, méthode `serviceChanged`
    - Traitement séquentiel et synchrone des listeners



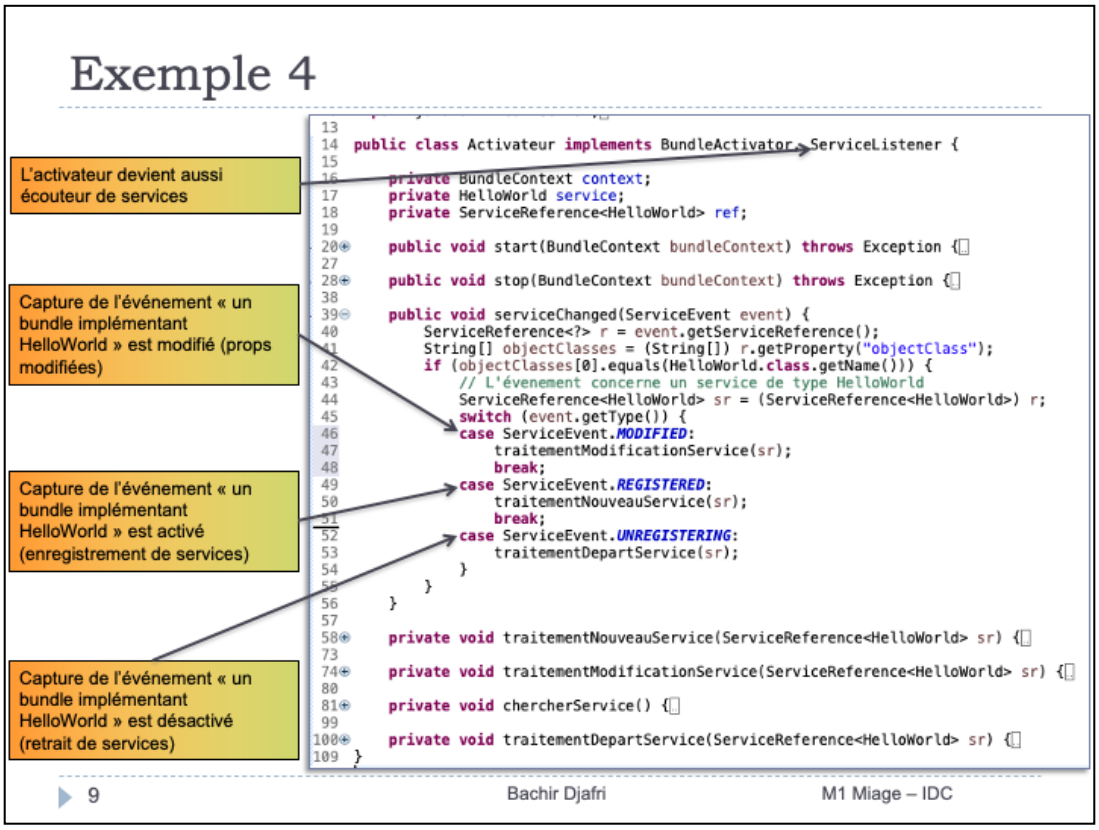
Dans le cas d'une interface de service optionnel (interface optionnelle), le composant peut être activé même si aucun service requis n'est trouvé à l'activation (méthode start()).

Dans le cas contraire, si le service est obligatoire (interface obligatoire, non optionnelle) le composant ne peut pas être activé si un ou plusieurs services requis obligatoires sont absents. Dans ce cas, la méthode start ne doit pas terminer normalement (passage dans l'état Actif).

Il faut lever une exception et indiquer qu'il manque un service obligatoire. La méthode start ne termine pas normalement et est « abortée ».

Lorsque tous les services requis sont optionnels, le composant peut être activé, mais ne va pas vérifier, pendant qu'il est dans l'état Actif, si de nouveaux services dont il a besoin sont enregistrés (arrivés) ou non. Ce type de composant est dit STATIQUE.

Si on veut qu'il soit dynamique (vérifie l'arrivée ou le départ de services dont il a besoin), il doit utiliser un écouteur (ou un tracker, à voir plus tard).



Un écouteur est un objet d'une classe qui implémente l'interface ServiceListener (interface des écouteurs de services).

On peut utiliser la classe Activateur pour implémenter cette interface, comme on peut aussi créer une nouvelle classe pour implémenter cette interface.

Ensuite, il faut implémenter la seule méthode de l'interface : serviceChanged qui prend en paramètre un événement de type ServiceEvent.

Ce qu'il faut retenir ici :

A partir du ServiceEvent, on peut obtenir l'objet ServiceReference qui a déclenché l'événement (event.getServiceReference()).

A partir cet objet ServiceRTeference on peut obtenir toutes les propriétés du service ou une propriété particulière en indiquant sa clé.

Celle qui nous intéresse est celle du type du service. La clé de cette propriété est « objectClass ». La valeur associée à cette clé est un tableau d'object (Object[]) qui correspond aux types (classes/interfaces) enregistrés lors de l'enregistrement du service (context.registerService(TypeService.**class**, **s**, props);).

On récupère à partir de ce tableau transformé en tableau de chaines de caractères, les noms des types (classes/interfaces) implémentés par l'objet de service.

On teste alors si le service qui a déclenché l'événement est de type HelloWorld. Si oui, je commence les traitements. Sinon, je ne fais rien. Le service ne me concerne pas. C'est un service d'un autre type que je n'utilise pas.

Deuxième point important à retenir ici est que l'événement m'indique s'il s'agit d'un nouveau service qui a déclenché l'événement ou bien une modification dans un service déjà enregistré ou bien le dé-enregistrement d'un service donné.

Ces trois cas sont renseignés par une valeur constante de la classe ServiceEvent : REGISTRED, MODIFIED, UNREGISTERING.

A partir de là, on réalise le traitement qui correspond à chaque situation selon notre spécification du composant. On peut faire ce qu'on veut et appliquer la politique qu'on veut.

Pour le reste, c'est de l'implémentation selon le cahier des cherches du composant.

Exemple 4

utilisation de l'écouteur d'évènement

On informe le contexte de l'arrivée de cet écouteur

On recherche les service disponibles

On informe le contexte du départ d'un écouteur

```
14 public class Activateur implements BundleActivator, ServiceListener {
15
16     private BundleContext context;
17     private HelloWorld service;
18     private ServiceReference<HelloWorld> ref;
19
20     public void start(BundleContext bundleContext) throws Exception {
21         context = bundleContext;
22         ref=null;
23         service=null;
24         context.addServiceListener(this);
25         chercherService();
26     }
27
28     public void stop(BundleContext bundleContext) throws Exception {
29         if (service != null) {
30             service.goodbye();
31             service = null;
32             context.ungetService(ref);
33             ref=null;
34             context.removeServiceListener(this);
35         }
36         context = null;
37     }
38
39     public void serviceChanged(ServiceEvent event) {
57
58     private void traitementNouveauService(ServiceReference<HelloWorld> sr) {
73
74     private void traitementModificationService(ServiceReference<HelloWorld> sr) {
80
81     private void chercherService() {
99
100    private void traitementDepartService(ServiceReference<HelloWorld> sr) {
109 }
110
```

Exemple 4

Récupère les services de type « HelloWorld» disponibles

Connexion à un service

Libération du service

```
58    private void traitementNouveauService(ServiceReference<HelloWorld> sr) {
59        System.out.println("Nouveau service : (Langue = " + sr.getProperty("Langue") + ")");
60        if (this.ref == null) { // pas encore de service
61            ref = sr;
62            service = context.getService(ref);
63            service.hello(); // utilisation du service.
64        } else if (!ref.getProperty("Langue").equals("Fr") && sr.getProperty("Langue").equals("Fr")) {
65            // meilleur service
66            service.goodbye();
67            context.ungetService(ref);
68            ref = sr;
69            service = context.getService(ref);
70            service.hello(); // utilisation du service.
71        } // sinon, ne rien faire.
72    }
73
74    private void traitementModificationService(ServiceReference<HelloWorld> sr) {
75        if (this.ref.equals(sr) && !sr.getProperty("Langue").equals("Fr")) {
76            // chercher un meilleur service
77            chercherService();
78        }
79    }
80
81    private void chercherService() {
82        try {
83            Collection<ServiceReference<HelloWorld>> refs = context.getServiceReferences(HelloWorld.class, "(Langue=*)");
84            if (!refs.isEmpty()) { // il existe refs.size() service.s disponible.s
85                Iterator<ServiceReference<HelloWorld>> it = refs.iterator();
86                ref = it.next(); // on prend le premier service et on va chercher un en français.
87                while (!ref.getProperty("Langue").equals("Fr") && it.hasNext()) {
88                    ServiceReference<HelloWorld> r = it.next();
89                    if (r.getProperty("Langue").equals("Fr")) {
90                        ref = r;
91                        break;
92                    }
93                }
94                service = context.getService(ref); // on prend le meilleur service trouvé.
95                service.hello();
96            }
97        } catch (InvalidSyntaxException e) { e.printStackTrace(); }
98    }
99
100   private void traitementDepartService(ServiceReference<HelloWorld> sr) {
101       if (ref != null && ref.equals(sr)) {
102           service.goodbye();
103           service = null;
104           context.ungetService(ref);
105           ref = null;
106           chercherService();
107       }
108   }
```

Exemple 4

- Une classe Ecouteur de services

```
 9 public class Ecouteur implements ServiceListener {
10     private ServiceReference<HelloWorld> ref;
11
12     public void serviceChanged(ServiceEvent event) {
13         ServiceReference<?> r = event.getServiceReference();
14         String[] objectClasses = (String[]) r.getProperty("objectClass");
15         if (objectClasses[0].equals(HelloWorld.class.getName())) {
16             // L'évenement concerne un service de type HelloWorld
17             ref = (ServiceReference<HelloWorld>) r;
18             switch(event.getType()) {
19                 case ServiceEvent.REGISTERED : traitementNouveauService(ref); break;
20                 case ServiceEvent.MODIFIED : traitementModificationService(ref); break;
21                 case ServiceEvent.UNREGISTERING : traitementDepartService(ref);
22             }
23         }
24     }
25
26     private void traitementNouveauService(ServiceReference<HelloWorld> ref) {
29
30     private void traitementModificationService(ServiceReference<HelloWorld> ref) {
33
34     private void traitementDepartService(ServiceReference<HelloWorld> ref) {
37 }
38
```

Première conclusion sur le modèle OSGi

- **OSGi est une modèle de composants dynamiques orienté services**
  - Basé sur Java (implémentation avec Java)
  - Non distribué, mais gestion distante possible (installation, désinstallation, etc.)
- **Les points positifs**
  - C'est une technologie en pleine expansion soutenue par une communauté active
  - Ce modèle fournit un comportement dynamique simple
  - Il fournit une solution aux problèmes de classpath et à la gestion de versions des composants
  - De nombreux bundles fournissent des fonctionnalités avancées
- **Les points négatifs (à ce stade)**
  - Comme pour toutes ces technologies nouvelles : le manque d'outils d'ingénierie logicielle
  - Le déchargement de classes pas toujours facile
  - L'écoute des évènements non plus n'est pas facile (*si l'on s'en tient à ce qu'on a vu jusque là !)*

… et à compléter

- Notion de ServiceListener
  - *introduite avec la release 1 (une) d'OSGi*
  - Enregistrement de services et d'écouteurs d'événements auprès du contexte
  - Le développeur doit, dans la classe d'activation (méthode start) :
    - Créer et enregistrer les services qu'il fournit
    - Créer et enregistrer les écouteurs d'événements des services qu'il requiert pour fonctionner (services requis via interfaces requises)
    - Implémenter au sein des écouteurs d'événements la manière dont le composant doit réagir aux événements perçus (reçus)
- Deux autres notions introduites depuis
  - Release 2 – ServiceTracker
  - Release 4 – Declarative Services (Cours OSGI avancé)