



Introduction au développement par composants (à grande échelle) OSGi

Intervenant : Chouki TIBERMACHINE

Bureau : LIRMM (E.311)

Tél. : 04.67.14.97.24

Mél. : Chouki.Tibermachine@lirmm.fr

Web : <http://www.lirmm.fr/~tibermacin/ens/hmin304/>

Plan du cours

- Limites des archives JAVA
- Généralités sur le framework de composants Java OSGi
- Définition de composants OSGi
- Assemblage de composants OSGi

Plan du cours

- Limites des archives JAVA
- Généralités sur le framework de composants Java OSGi
- Définition de composants OSGi
- Assemblage de composants OSGi

Archives Java (fichiers JAR)

- Une archive Java (fichier JAR) représente dans Java l'unité de base du déploiement d'applications
- Est-ce que ces unités de code offrent suffisamment de modularité ?
- Non, pas vraiment. Pourquoi ?

Archives Java en pratique

- Un JAR n'est qu'un regroupement de classes, interfaces et autres ressources au sein d'une même structure réutilisable
- Une fois placé dans le classpath, le contenu d'un JAR se dissout dans l'espace de classes, avec les contenus de tous les autres JARs
- Par conséquent, chaque classe publique dans ce JAR est accessible par n'importe quelle autre classe (dans les autres JARs)
- Les frontières du JAR disparaissent donc complètement à l'exécution:
 - Un JAR n'a pas de sémantique précise à l'exécution

Archives Java en pratique -suite-

- Les archives Java n'offrent aucun moyen de gérer les versions
- Dans de grosses applications où l'on utilise beaucoup de librairies (de JARs) :
 - Si deux composants différents de l'application utilisent la même librairie mais en deux versions différentes :
 - l'une des librairies écrasera l'autre

OSGi est un framework Java qui étend les capacités des JARs offrant ainsi plus de modularité dans la construction d'applications Java de grande taille

Plan du cours

- Limites des archives JAVA
- Généralités sur le framework de composants Java OSGi
- Définition de composants OSGi
- Assemblage de composants OSGi

C'est quoi un composant (= *bundle*) OSGi ?

- Un composant JAVA ayant des dépendances explicites et encapsulant son implémentation
- Concrètement, il s'agit d'un JAR ayant :
 - un Manifest (META-INF/MANIFEST.MF) avec des méta-données spécifiques (permettant entre autres plus de modularité) :
 - Nom du composant
 - Sa version
 - Ses dépendances (interfaces requises), mais aussi ses interfaces fournies
 - Ses paramètres de déploiement
 - ...
 - une éventuelle classe « Activator » exécutée au moment de l'activation ou la désactivation du composant

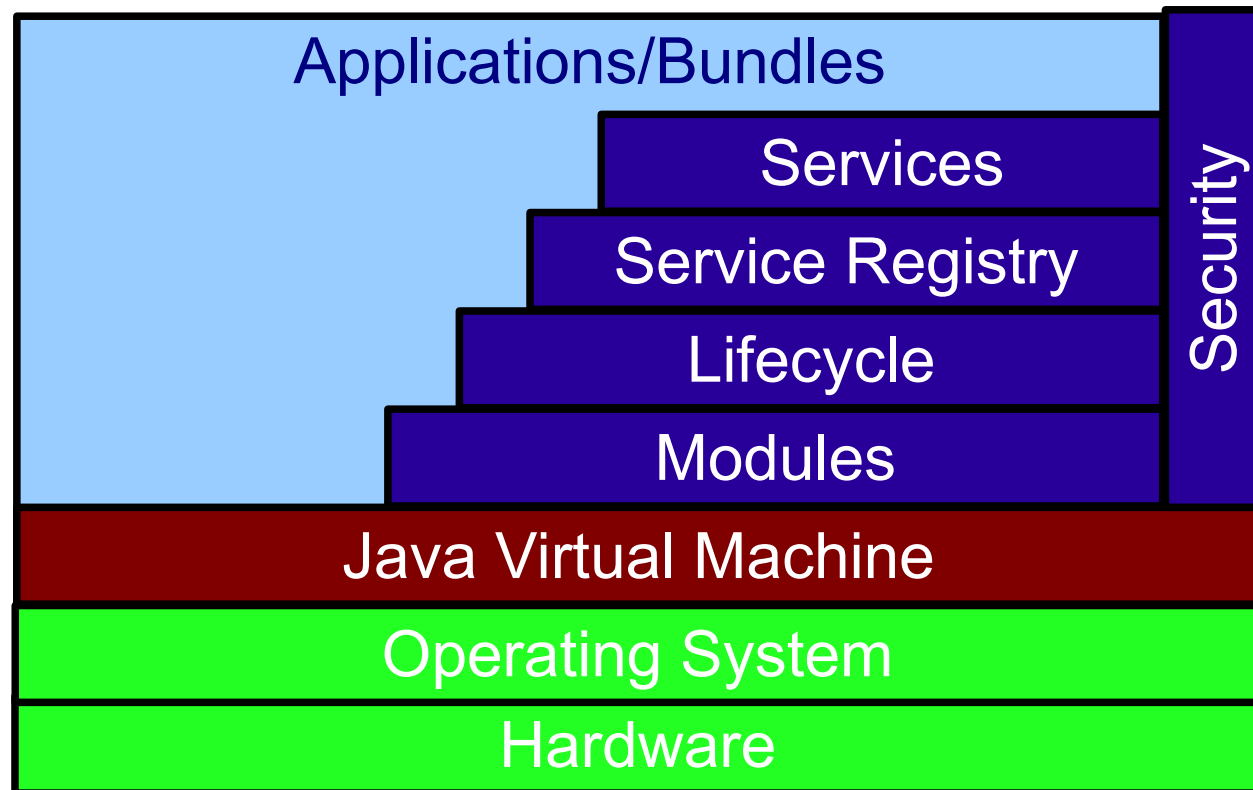
C'est quoi OSGi ?

- Un framework Java pour :
 - La programmation modulaire en JAVA : implémentation d'architectures à base de composants dans une machine virtuelle Java (JVM)
 - La gestion transparente des dépendances entre composants
 - L'implémentation d'architectures orientées services dans une même JVM
 - La gestion de plusieurs autres aspects techniques, comme le service HTTP ou la journalisation, ou non-fonctionnels, comme la sécurité, ...

Histoire du framework OSGi

- OSGi Alliance :
 - Un consortium à but non lucratif, regroupant entre autres IBM, Oracle, SAP, Motorola, Samsung, Nokia, BEA, ...
 - Créé en mars 1999
 - Fournit des spécifications pour proposer des solutions JAVA plus robustes, modulaires, dynamiques, ...
- Distributions du framework :
 - Spec OSGi Release 1: 2000
 - Spec OSGi Release 2-4 : 2001-2005
 - Spec OSGi Release 5,6 : 2012,2014
 - Spec OSGi Release 7 : avril 2018

Architecture du framework OSGi



Quelques implémentations connues de ce framework

- Une multitude d'implémentations : open source, commerciales, pour environnements mobiles, domotiques, ...
- Les plus connues (open source) :
 - Eclipse Equinox : l'implémentation de référence
 - Développée par IBM et fournie sous la licence EPL (*Eclipse Public License*)
 - Eclipse est construit au dessus d'Equinox
 - Un plugin Eclipse = un bundle++
 - Apache Felix (anciennement, Oscar d'ObjectWeb) :
 - Développée par Apache et fournie sous la licence *Apache License 2*
 - Netbeans et Glassfish sont construits dessus
- Liste complète (impl. certifiées) : <http://www.osgi.org/Markets/>



Comment OSGi apporte de la modularité aux apps Java ?

■ **Encapsulation :**

- Chaque composant OSGi est chargé à l'exécution dans son propre espace de classes (indépendant des autres)
- Tout le contenu d'un composant (ses classes) est donc privé, sauf certaines parties qui sont explicitement exportées dans le manifest (ses interfaces fournies et requises)
- Chaque implémentation interne d'un composant peut évoluer indépendamment des autres composants

■ **Service Registry :** Implémentation du patron « Service Locator »

- Un composant peut publier ses services dans un annuaire
- Un composant peut consommer des services publiés par d'autres composants
- Les services sont connus par leurs interfaces publiés (et non leur implémentation) : couplage faible

Comment OSGi apporte de la modularité aux apps Java ?

- **Versions parallèles d'un composant :**

- Chaque composant est publié dans un espace de classes propre
- Plusieurs versions d'un même composant peuvent co-exister dans une même application

- **Reconfiguration dynamique :**

- Il est possible de charger dynamiquement des modules (pendant l'exécution de l'application)
- Un composant peut être remplacé dynamiquement par une nouvelle version de celui-ci

- **Nommage fort :**

- A la différence des JARs traditionnels, les composants OSGi ont un nom et un numéro de version : identification unique
- *Bundle Symbolic Name*

Plan du cours

- Limites des archives JAVA
- Généralités sur le framework de composants Java OSGi
- Définition de composants OSGi
- Assemblage de composants OSGi

Mettre en place un composant OSGi

1. Définir le contenu du bundle :

- Écrire et compiler les interfaces et classes composant le bundle
- Les organiser dans une certaine structure en packages (voir prochains transparents)

2. Écrire le Manifest du bundle :

- Spécifier les méta-données nécessaires :
 - à son identification : son nom, son numéro de version, ...
 - et à son exécution : spécifier entre autres quels sont les packages à exporter (les interfaces fournies), les packages à importer (les interfaces requises)

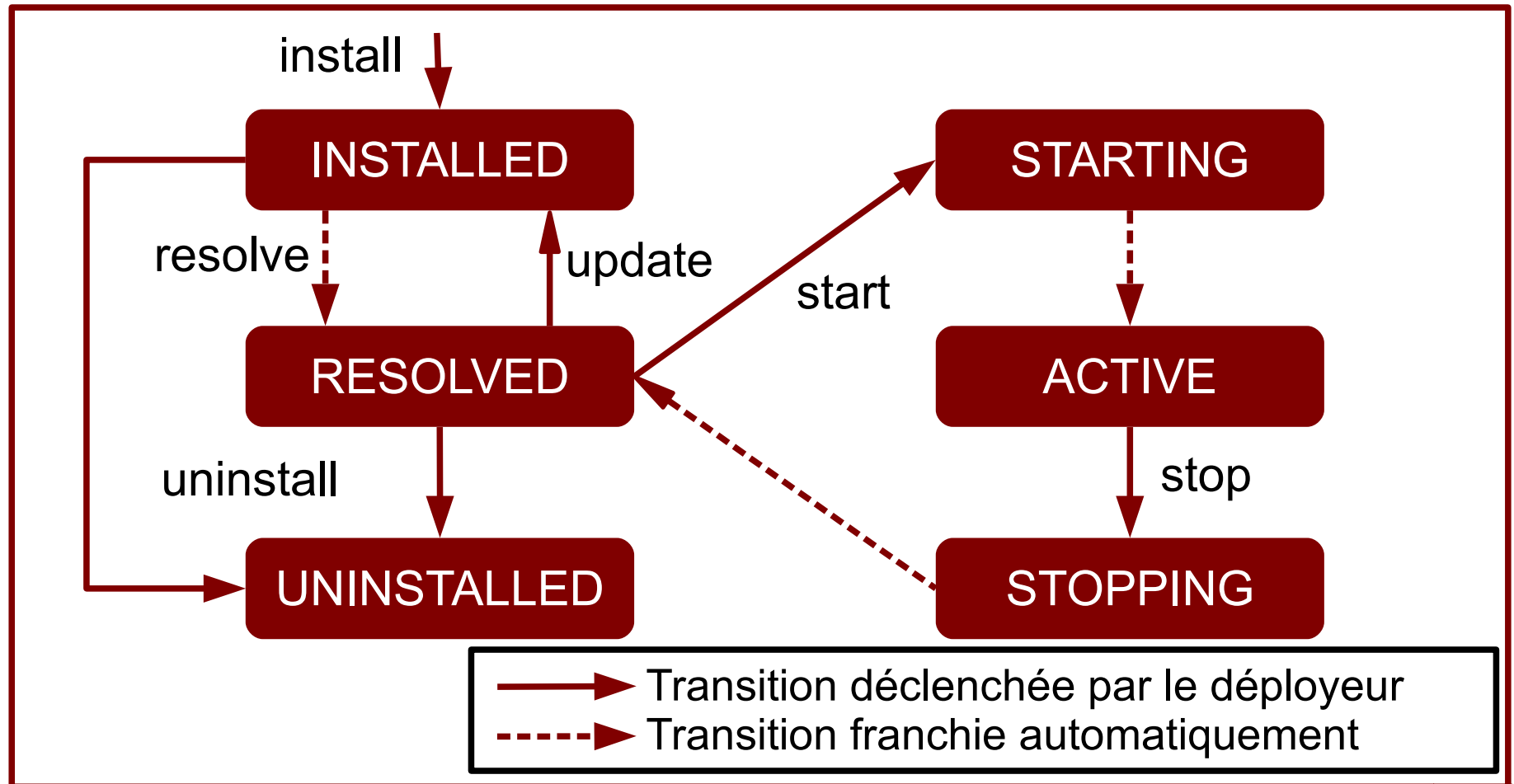
3. Former le bundle : créer le JAR avec les éléments précédents

4. Le déployer dans un **container OSGi** : l'installer puis l'activer

C'est quoi un container OSGi ?

- Une instance de l'implémentation du framework en cours d'exécution
- Un développeur peut déployer des bundles dans un container
- Il peut les activer, les désactiver, ... (voir cycle de vie d'un bundle)
- Le container gère certains aspects techniques de façon automatique et transparente pour le développeur de l'application

Cycle de vie d'un composant OSGi



Organiser son code dans un bundle

- Exporter les packages contenant uniquement des interfaces
- Les packages avec les implémentations des interfaces (les classes) doivent être cachés aux autres composants
- Les autres composants ne vont dépendre que des interfaces de notre bundle
- De cette façon, les implémentations dans notre composant peuvent évoluer sans impact sur les autres composants

Méta-données (Headers) dans le Manifest

- **Bundle-ManifestVersion**: spécification OSGi (utiliser la valeur 2 pour OSGi release 4+, par défaut : 1, recommandé)
- **Bundle-SymbolicName**: Le seul obligatoire pour qu'un JAR soit considéré comme un bundle OSGi
 - Il s'agit du nom unique du bundle (bonne pratique : nom qualifié complet de l'interface fournie par le composant)
- **Bundle-Name**: Un nom lisible (sans espaces)
- **Bundle-Version**
- **Bundle-Activator**: la classe « Activator »
- **Import-Package**: Liste des packages requis par le bundle et qui seront fournis par d'autres bundles
- **Export-Package**: Liste des packages à exporter

La classe « *Activator* »

- Elle contient des méthodes liées au cycle de vie d'un bundle
- Elle n'est pas obligatoire dans un bundle
- Elle doit implémenter l'interface :
`org.osgi.framework.BundleActivator`
- Elle doit donc rendre concrètes les deux méthodes callbacks :
`public void start(BundleContext ctx)`
`public void stop(BundleContext ctx)`
- Ces méthodes sont invoquées automatiquement par le framework au moment de l'activation (start) et la désactivation (stop) du bundle

Exemple de classe « *Activator* »

■ Un composant HelloWorld :

```
package hmin304.cours.hello;

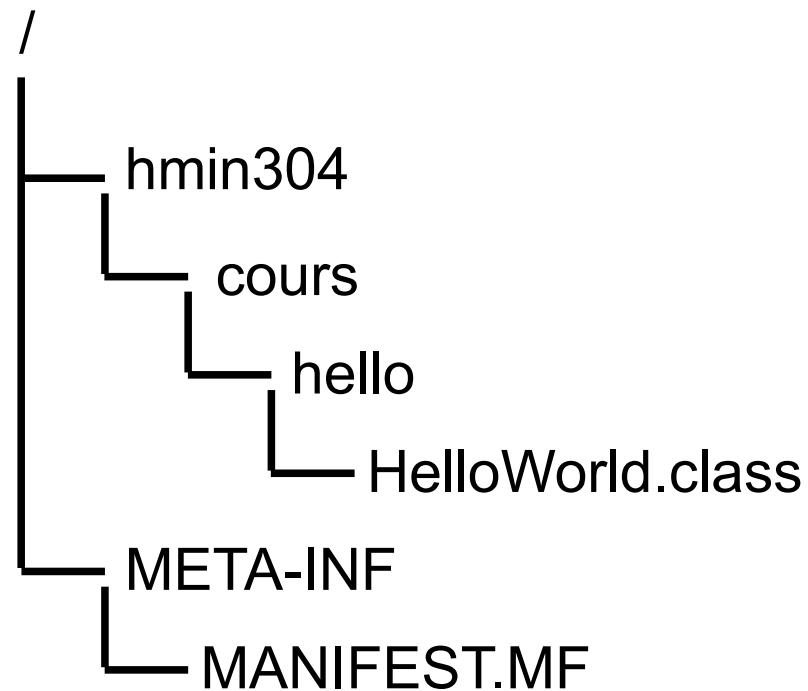
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class HelloWorld implements BundleActivator {

    public void start(BundleContext ctx) throws Exception {
        System.out.println("Hello World !");
    }

    public void stop(BundleContext ctx) throws Exception {
        System.out.println("Goodbye World !");
    }
}
```

Structure du composant OSGi



Contenu du Manifest de notre exemple

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: hmin304.cours.hello.HelloWorld
Bundle-Name: HelloWorld
Bundle-Version: 1.0.0
Bundle-Activator: hmin304.cours.hello.HelloWorld
Import-Package: org.osgi.framework
```


Différence entre Import-Package et Require-Bundle ?

- Dans un Manifest, on peut utiliser pour déclarer les requis d'un bundle les entêtes `Import-Package` et/ou `Require-Bundle`
- **Import-Package** : le bundle requiert un package qui peut être fourni par n'importe quel autre bundle (la dépendance entre les bundles sera dynamiquement résolue par le framework)
- **Require-Bundle** : le bundle requiert un autre bundle (en précisant son `Bundle-SymbolicName`)
Le premier bundle a accès à tous les `Export-Package` du second
- Il faut privilégier `Import-Package` à `Require-Bundle` :
Couplage minimal entre composants (le package requis peut être fourni par n'importe quel bundle)

Et quand vos bundles dépendent de JARs « non OSGi » ?

- **Solution 1** : Intégrer le JAR dans le bundle qui en dépend
 - Le mettre dans un répertoire lib
 - Ajouter le chemin vers ce JAR au Manifest :
`Bundle-ClassPath: lib/<le-jar>`
 - Exemple :
`Bundle-ClassPath: lib/jdom.jar,mdt_ocl.jar`
 - Inconvénient : lorsque le jar est utilisé par plusieurs bundles
- **Solution 2** : Envelopper le JAR dans un nouveau bundle
 - Modifier son Manifest :
 - Ajouter un `Bundle-SymbolicName` (devenir OSGi-Ready)
 - Exporter les packages requis par les autres bundles :
 - Ajouter un `Export-Package`

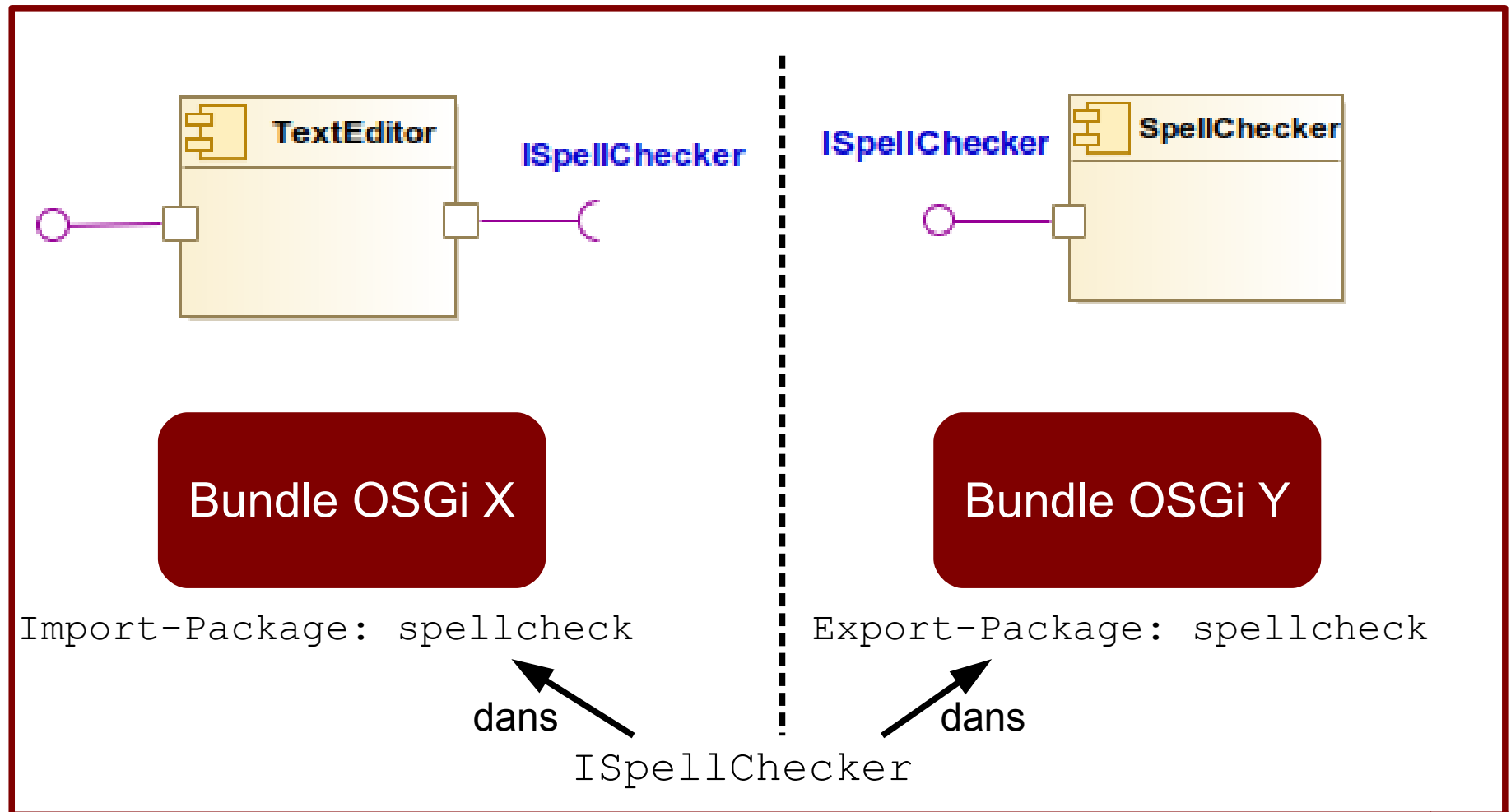
Directives pour raffiner les dépendances dans Equinox

- Il est possible d'ajouter à l'entête Export-Package des directives permettant de raffiner les dépendances entre bundles :
 - x-friends : permet d'indiquer les bundles pour lesquels les packages sont exportés
Export-Package: mon.package.a.exporter.aux.amis; x-friends:="mon.ami1, mon.ami2"
Le framework va juste décourager les bundles qui ne sont pas listés à utiliser les packages de ce bundles (non bloquant)
 - x-internal : permet d'indiquer (par true ou false) si un package exporté est interne (ne fait partie pas de l'API, il est provisoirement exporté)
Export-Package: mon.package.a.exporter.mais.provisoirement; x-internal:=true

Plan du cours

- Limites des archives JAVA
- Généralités sur le framework de composants Java OSGi
- Définition de composants OSGi
- Assemblage de composants OSGi

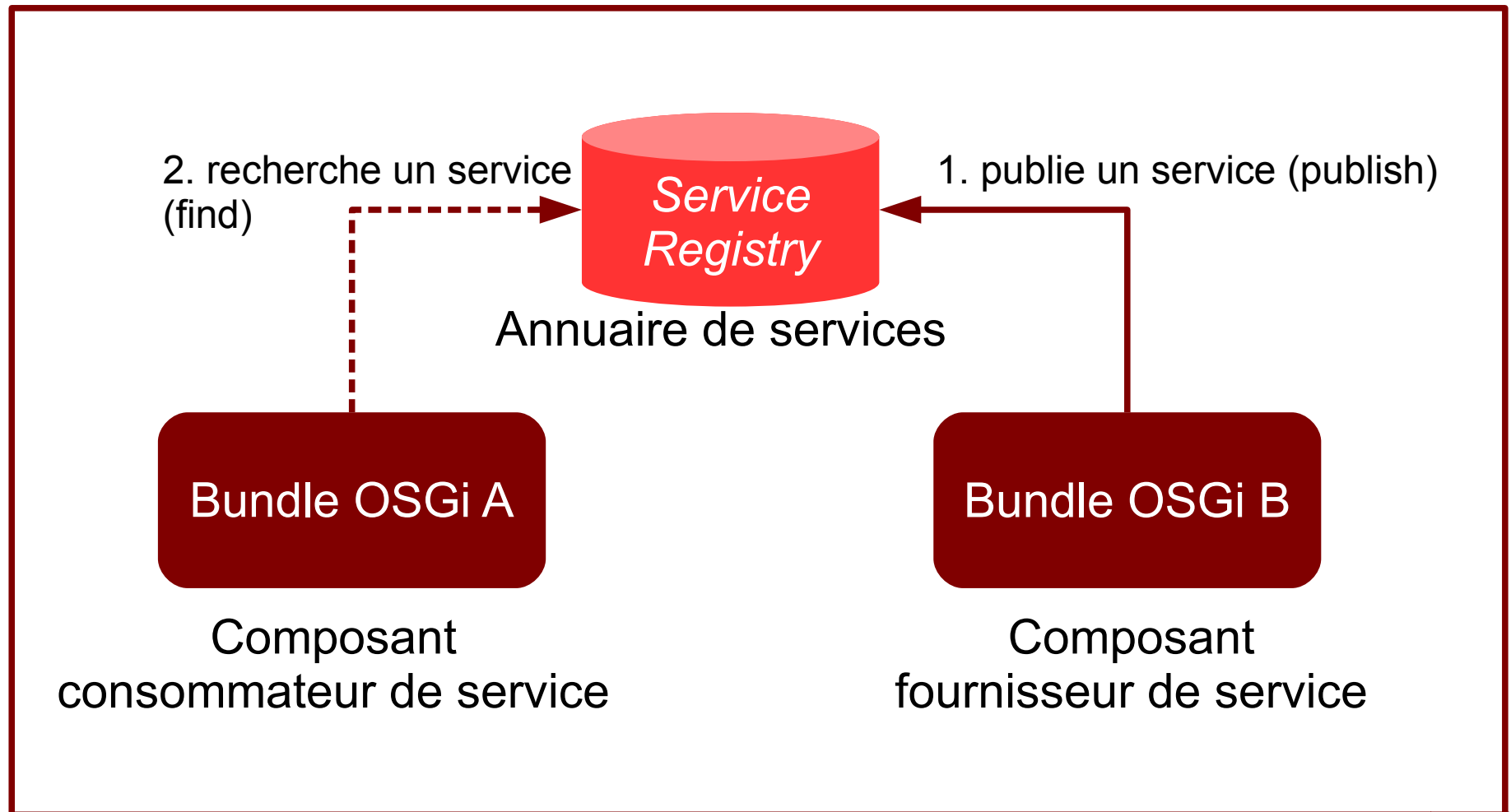
Une architecture à base de composants



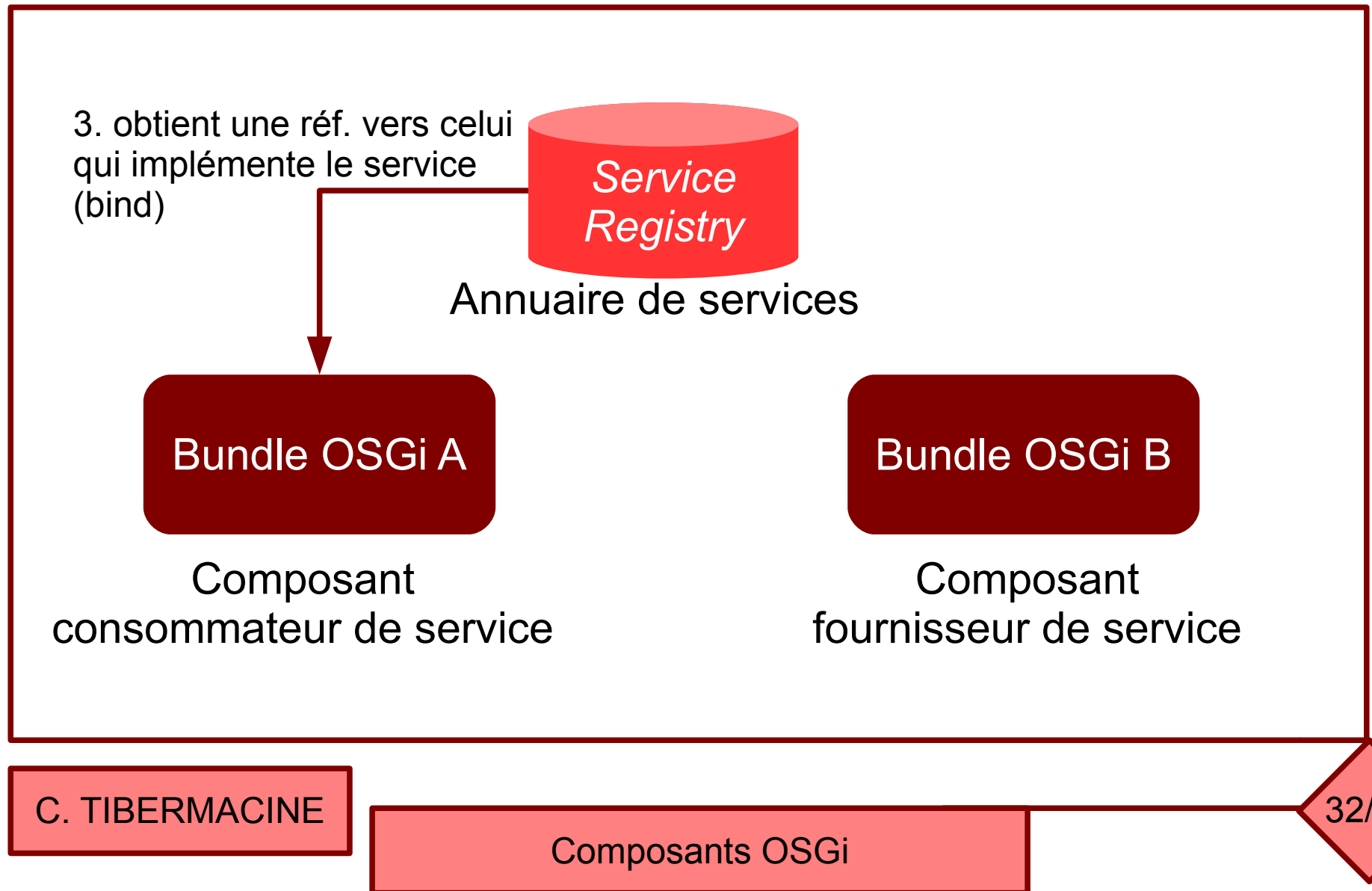
Connexion entre composants en utilisant une Factory

- Le composant déclarant l'interface fournie exporte également une classe Factory (d'objets qui implémentent `ISpellChecker`)
- Cette classe Factory définit une ou plusieurs méthodes statiques qui ont comme type de retour l'interface fournie (`ISpellChecker`)
- Cette classe Factory est définie dans le package de l'interface fournie, qui est exporté par le bundle (`spellcheck`)
- Le composant déclarant l'interface requise importe ce package
- Il invoque les méthodes de la classe Factory pour obtenir les instances dont il a besoin

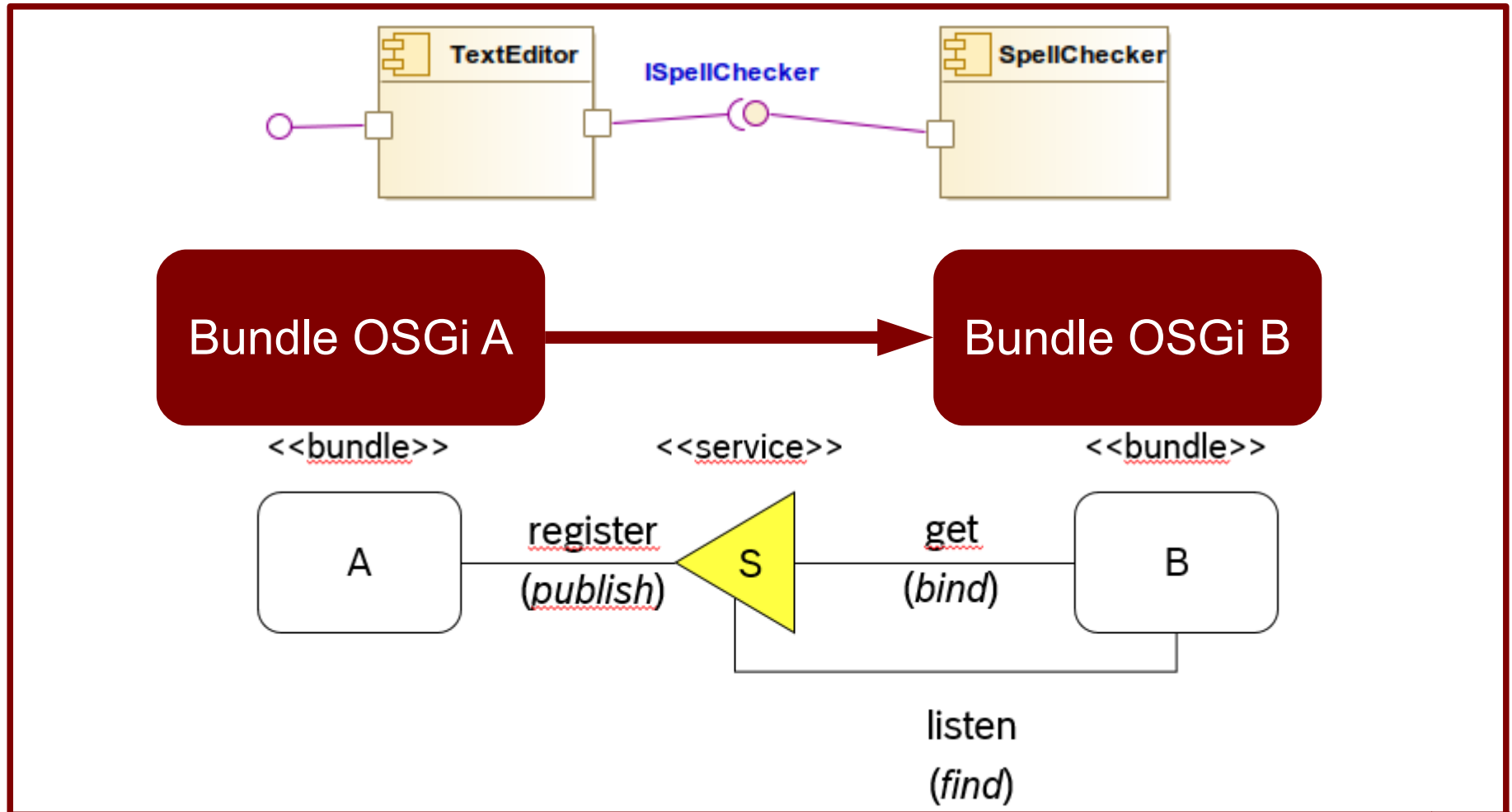
Une architecture orientée-services (SOA) *publish-find-bind*



Une architecture orientée-services *publish-find-bind*



Une architecture orientée-services -suite-



Composant fournisseur de services : « *Service Bundle* »

- Un composant OSGi peut publier certaines de ses méthodes publiques fournies par un objet sous la forme de services
- Ces services sont d'abord enregistrés dans le « *Service Registry* » (l'annuaire de services géré par le container OSGi)
- Enregistrement d'un service auprès d'un annuaire :
 - Inscrire l'interface et une instance implémentant cette interface
- Ensuite, d'autres bundles déployés **dans le même container** peuvent rechercher ces services dans l'annuaire (**SOA in JVM**)
- Ces bundles reçoivent de l'annuaire la référence d'une instance de l'implémentation du service

Structure type d'un « *Service Bundle* »

- La structure type d'un « *Service Bundle* » est la suivante :
 - Une interface Java qui exporte les méthodes publiques (qui sont publiées comme opérations du service)
 - Une classe qui implémente l'interface (**une classe POJO** : rien dans la classe n'indique qu'il s'agit de l'implémentation d'un service)
 - Une classe « *Activator* » qui publie le service (l'ensemble des méthodes) dans l'annuaire grâce à la méthode *start* et qui le dés-enregistre dans la méthode *stop*
- Le bundle est un JAR dans lequel on exporte uniquement l'interface du service publié

Exemple de « Service Bundle »

- L'interface (une simple interface Java) :

```
package hmin304.cours.hello.service;  
public interface HelloService {  
    String getHelloMessage();  
    String getGoodbyeMessage();  
}
```

- La classe (une simple classe JAVA) :

```
package hmin304.cours.hello.service.impl;  
import hmin304.cours.hello.service>HelloService;  
public class HelloImpl implements HelloService {  
    public String getHelloMessage() {  
        return "Hello World !";  
    }  
    // ...  
}
```

Exemple de « Service Bundle »

■ Le « ServicePublisher » (la classe « *Activator* ») :

```
package hmin304.cours.hello.service.impl;
import org.osgi.framework.*;
import hmin304.cours.hello.service.HelloService;
public class HelloPublisher implements BundleActivator {
    private ServiceRegistration registration;

    public void start(BundleContext ctx) throws Exception {
        registration = ctx.registerService(
            HelloService.class.getName(),
            new HelloImpl(), null);
    }

    public void stop(BundleContext ctx) throws Exception {
        registration.unregister(); }
}
```

Exemple de « Service Bundle »

- Le « ServicePublisher » (la classe « *Activator* ») :

```
package hmin304.cours.hello.service.impl;
import org.osgi.framework.*;
import hmin304.cours.hello.service.HelloService;
public class HelloPublisher implements BundleActivator {
    private ServiceRegistration registration;

    public void start(BundleContext ctx) throws Exception {
        registration = ctx.registerService(
            HelloService.class.getName()
            new HelloImpl(), null);
    }

    // ...
}
```

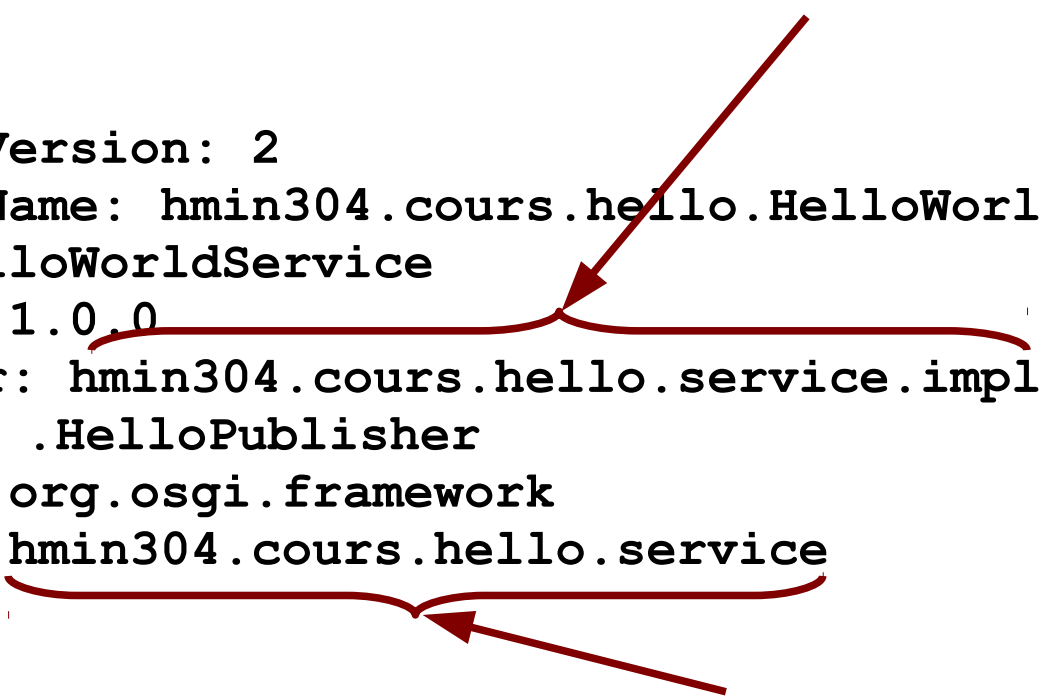
Clé :
Nom qualifié
complet de l'interface

Contenu du Manifest de notre exemple

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: hmin304.cours.hello.HelloWorldService
Bundle-Name: HelloWorldService
Bundle-Version: 1.0.0
Bundle-Activator: hmin304.cours.hello.service.impl
                  .HelloPublisher
Import-Package: org.osgi.framework
Export-Package: hmin304.cours.hello.service
```

Contenu du Manifest de notre exemple

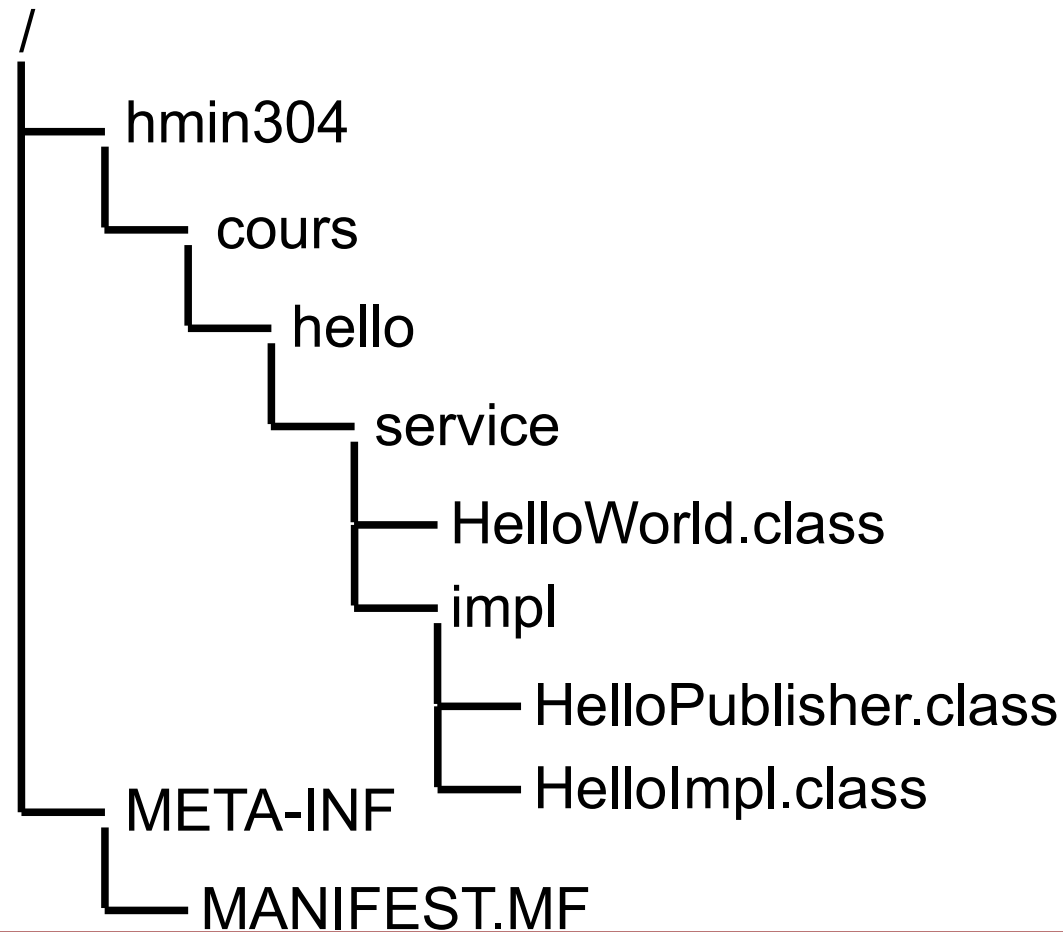
```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: hmin304.cours.hello.HelloWorldService
Bundle-Name: HelloWorldService
Bundle-Version: 1.0.0
Bundle-Activator: hmin304.cours.hello.service.impl
                  .HelloPublisher
Import-Package: org.osgi.framework
Export-Package: hmin304.cours.hello.service
```



Classe qui publie le service
dans l'annuaire

Package qui contient uniquement
l'interface du service

Structure de notre composant



Composant consommateur de service

```
package hmin304.cours.hello;
import org.osgi.framework.*;
import hmin304.cours.hello.service.HelloService;
public class HelloWorld implements BundleActivator {
    public void start(BundleContext ctx) throws Exception {
        HelloService helloService = getHelloService(ctx);
        System.out.println(helloService.getHelloMessage());
    }
    private HelloService getHelloService(BundleContext ctx) {
        ServiceReference ref = ctx.getServiceReference(
                                HelloService.class.getName());
        HelloService helloService = (HelloService)ctx.getService(
                                ref); //Tester ref!=null
        return helloService; }
    //... Même chose pour stop : getGoodbyeMessage()
}
```

Contenu du Manifest du consommateur du Service

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: hmin304.cours.hello.HelloWorld
Bundle-Name: HelloWorld
Bundle-Version: 1.0.1
Bundle-Activator: hmin304.cours.hello.HelloWorld
Import-Package: org.osgi.framework,
               hmin304.cours.hello.service
```



Le bundle dépend d'une interface

Problème de disponibilité de services

- Les services dans OSGi sont de nature volatile
- Leur publication est dynamique, ils sont donc parfois indisponibles au niveau de l'annuaire pour un consommateur de services :
par exemple, le cas où le bundle qui enregistre un service est activé après le bundle qui le consomme
- Trois solutions possibles :
 - Programmer l'attente : Itération ou Timers
 - Avantage : solution simple à mettre en place
 - Inconv. : solution consommatrice en ressources et verbeuse
 - Utiliser un « *Service Tracker* » : évolution des « *listeners* » de services (« *Service Listener* »)
 - Exploiter le mécanisme de « *Declarative Services* »

Programmer l'attente de la disponibilité du service

■ Méthode *start* de l'activateur :

```
public void start(BundleContext ctx) throws Exception {
    ServiceReference ref;
    do {
        ref = ctx.getServiceReference(ISpellChecker.class
                                   .getName());
    }
    while(ref == null);
    ISpellChecker scService=(ISpellChecker) ctx
                               .getService(ref);
    registration = ctx.registerService(ITextEditor.class
                                       .getName(),
    new TextEditor(scService), null);
}
```

Le bundle est bloqué dans l'état **Starting** jusqu'à disponibilité du service

Utiliser un « *Service Tracker* »

- Mécanisme sophistiqué pour repérer les ajouts, mises à jour et suppressions de services du *Service Registry*

- Classe de l'activateur :

```
ServiceTracker tracker;//package org.osgi.util.tracker
public void start(BundleContext ctx) throws Exception {
    tracker = new ServiceTracker(ctx,ISpellChecker.class
.getName(),null);
tracker.open();
    ISpellChecker scService=(ISpellChecker) tracker.getService();
}
public void stop(BundleContext ctx) throws Exception {
    tracker.close();
}
```

Ne pas oublier de mettre à jour l'entête Import-Package du Manifest :
ajouter le package `org.osgi.util.tracker`

Utiliser les « Declarative Services » (DS) pour déclarer un fournisseur de services

- Pas besoin d'une classe « Activator »

- Déclarer dans un fichier XML le service publié :

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="hmin304.cours.spellchecker">
  <implementation class="...impl.FrSpellChecker"/>
  <service>
    <provide interface="....ISpellChecker"/>
  </service>
</scr:component>
```

scr veut dire Service Component Runtime

- Ici, nous avons déclaré un composant qui publie un service ISpellChecker

Utiliser DS pour déclarer un consommateur de services

- Là aussi, pas besoin d'une classe « Activator »
- Déclarer dans un fichier XML un consommateur de services :

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component
  xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="hmin304.cours.texteditor">
  <implementation class="...impl.TextEditor"/>
  <reference bind="setSpellChecker" cardinality="1..1"
    interface="...ISpellChecker" policy="static"/>
</scr:component>
```

- Injection automatique des dépendances via l'appel au setter : dès que le service ...ISpellChecker est disponible
- Le composant texteditor est activé quand toutes ses références sont satisfaites (tous les services à consommer sont disponibles) : cardinality = 1..1 et policy = static

Configurer DS

- Spécifier dans les Manifests des bundles le ou les chemin(s) vers le(s) fichier(s) XML grâce au header Service-Component
- Exemples :
Service-Component: OSGI-INF/component.xml
ou
Service-Component: OSGI-INF/emailer.xml, OSGI-INF/...xml
- Il est possible de déclarer des composants qui ne publient pas et ne consomment pas de services (de simples composants implémentés par des POJOs qui sont démarrés automatiquement)
- Définir des méthodes de cycle de vie (activate et deactivate) pour les composants : ces méthodes sont invoquées automatiquement

Declarative Services sans XML

- Il est possible d'utiliser des annotations prédéfinies dans OSGi :

- Déclarer un composant-service :

```
import org.osgi.service.component.annotations.Component;  
@Component(service=hmin304.IHelloer.class)  
public class Helloer implements IHelloer { ... }
```

- Pour référencer un service :

```
import org.osgi.service.component.annotations.Reference;  
...  
@Reference  
void bindHelloService(Helloer helloer) {  
    this.helloer = helloer;  
}
```

Le fichier XML est généré automatiquement

Quelques références

- **OSGi and Equinox: Creating Highly Modular Java Systems.** Jeff McAffer, Paul VanderLei et Simon Archer. The Eclipse Series. Dans les éditions de Jeff McAffer, Erich Gamma et John Weigand. Addison Wesley, 2010.
- **Modular Java: Creating Flexible Applications with OSGi and Spring.** Craig Walls. The Pragmatic Programmers, 2009.
- **Eclipse Plug-ins, 3rd edition.** Eric Clayberg et Dan Rubel. The Eclipse Series. Dans les éditions de Erich Gamma, Lee Nackman et John Wiegand. Addison Wesley, 2009.
4ème édition à paraître

Questions

