Université Claude Bernard Lyon 1

Frameworks de composants dynamiques (OSGi)

M2TI – Conception d'Applications Hétérogènes Distribuées

Lionel Médini

Septembre-novembre 2015

Frameworks de composants dynamiques

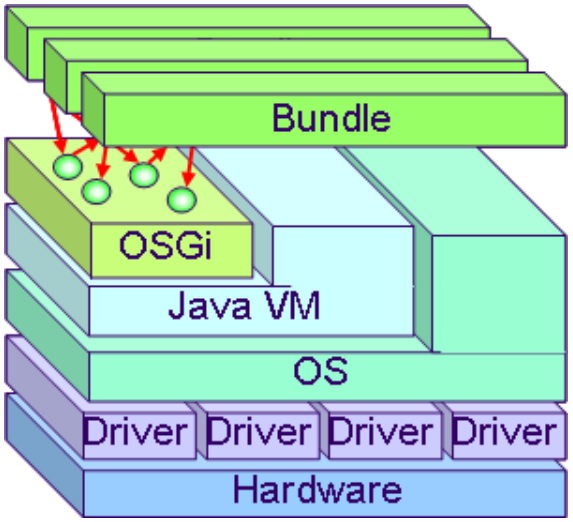
- Définition
  - Framework : IoC, gestion des dépendances, services transversaux
  - Composant : code exécutable packagé
  - Dynamique : déploiement (chargement / déchargement) d'applications « à chaud »
- Exemples
  - OSGi
  - .Net + UPnP…

La spécification OSGi

- Généralités
  - Origine du nom (1998)
    - Open Services Gateway Initiative
  - Spécification open source définie par l'OSGi Alliance
    - Fondation de l'OSGi Alliance : 1999
  - Spécifications en « Releases »
  - Implantée dans de nombreux frameworks applicatifs
  - Langage : Java
  - Nombreuses utilisations dans des applications industrielles

La spécification OSGi

- Principes
  - Architecture orientée composants
    - Composants = services = « bundles »
  - Couplage faible
    - Approche par interface
  - Late binding (chargement à chaud)
    - Reconfiguration dynamique
    - Ex. : plugins, services techniques
  - S'appuie sur un annuaire de composants
    - Capable de localiser les composants
    - Capable d'envoyer du code au client

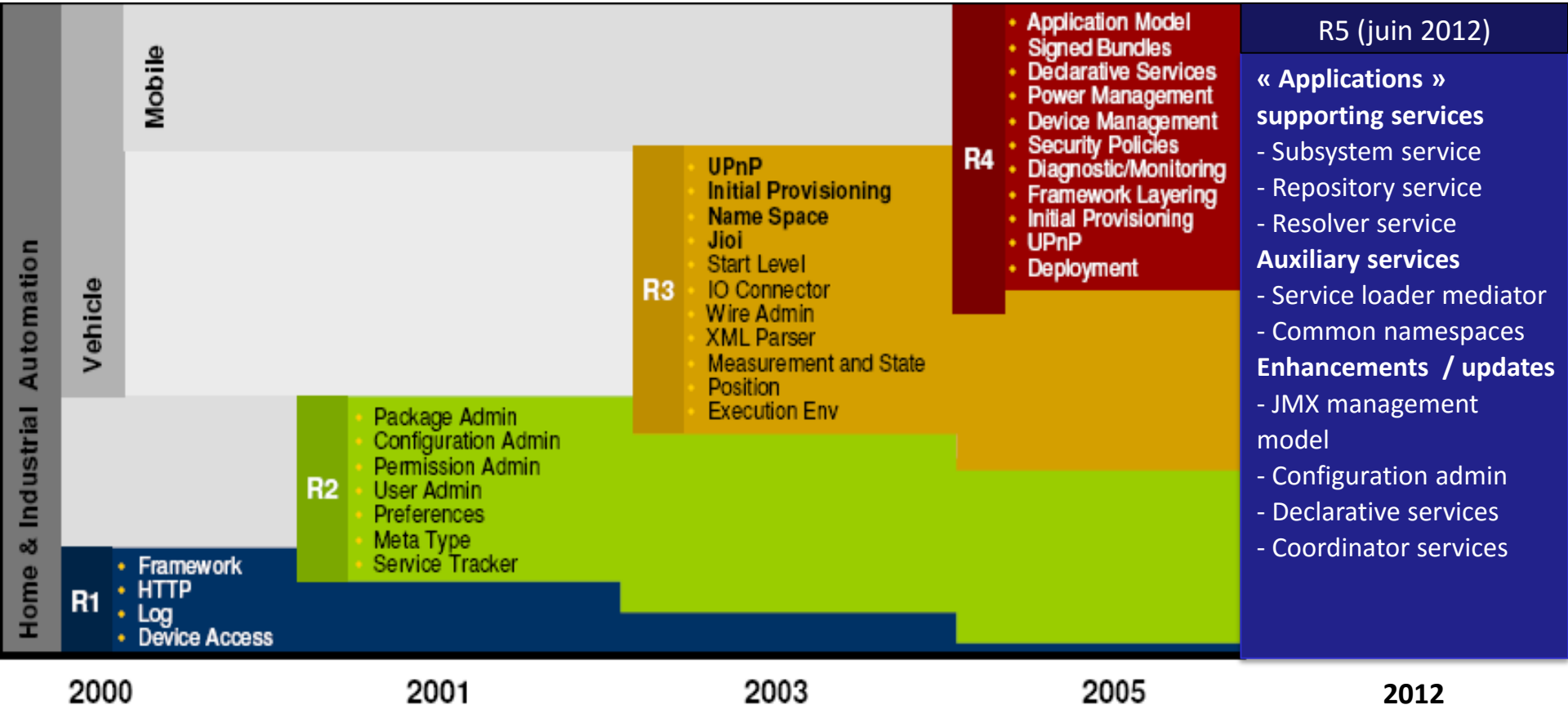


La spécification OSGi

- Avantages / spécificités
  - Modularité des applications
    - Chargement/déchargement de code dynamique
    - Déploiement dynamique d'applications sans interruption de la plate-forme
      - Installation, Lancement, Mise à jour, Arrêt, Retrait
      - « No reboot »
    - Résolution des dépendances versionnées de code
  - Légereté
    - Ne charge que le code nécessaire
    - Vise des systèmes à mémoire restreinte

La spécification OSGi

- Fonctionnalités (©2008 S. Frénot, D. Donsez, N. Le Sommer)



Source R5 : http://www.slideshare.net/bosschaert/the-os-gir5enterpriserelease

Implémentations

- Frameworks OSGi
  - Eclipse Equinox
  - Apache Felix
  - mBedded server…
- Autres frameworks Java
  - Serveurs d'applications Java EE
    - JBoss, GlassFish, Jonas…
  - ESB
    - Fuse, ServiceMix…

Applications ciblées

- À l'origine : mise à jour de modems / passerelles
- Puis : essor dans le domaine des applications
  - Dynamiques : chargement des drivers *via* l'API UPnP
  - Mobiles / ressources limitées : J2ME/CDC
  - Embarquées : mBedded Server
- Depuis :
  - Toute application décomposable : Java Platform 1.5, 6, 7…
  - Tout système multi-applications : Java EE, ESB…

Le framework OSGi

- Objectifs du framework
  - Instancier des objets dynamiquement en dehors de l'initialisation des classes
  - Pouvoir obtenir une nouvelle implémentation
    - À chaque invocation de méthode
    - À chaque mise à jour distante de la classe
  - Instanciation en Java « classique »

```
exemple1.QuelqueChose objet = new exemple2.AutreChose();
```

➔Approche par interface

Le framework OSGi

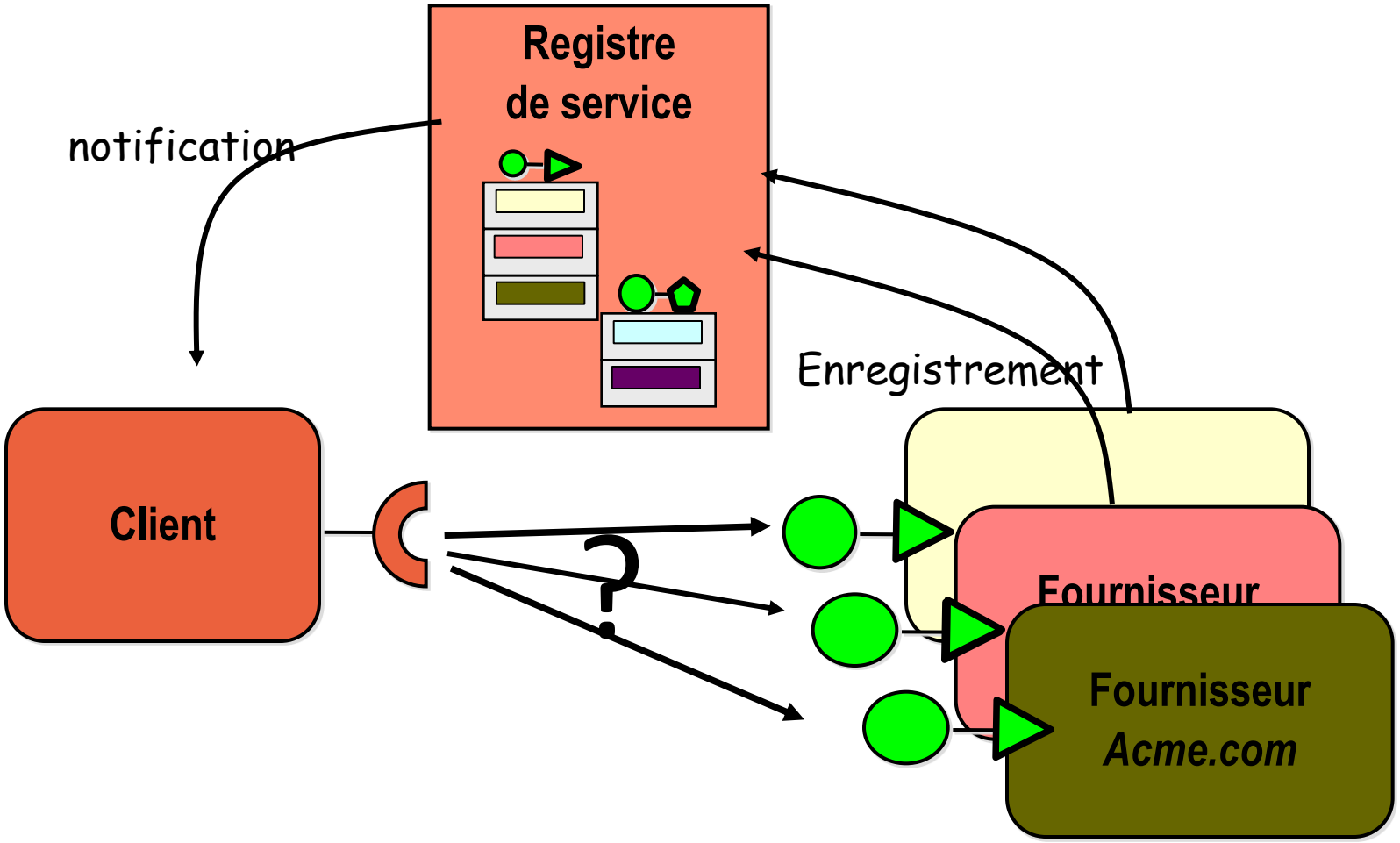
- Fonctionnement du chargement dynamique

```
URLClassLoader myCL = new URLClassLoader("http://www.somewhere.biz");
Class myClass = myCl.load("foo.AServiceImpl");
test.Service  exemple = (test.Service)myClass.newInstance();
```

  - Couplage léger entre demandeur et fournisseur de la classe
    - Seule l'interface du service est couplée au client
    - L'implémentation est chargée à l'exécution
- Remarques
  - Plus de constructeur → injection de dépendances
  - Nécessite de savoir où trouver les implémentations

Le framework OSGi

- Programmation orientée services (notification)



Le framework OSGi

- Caractéristiques d'un composant
  - Nom OSGi : « bundle »
  - Téléchargé à partir d'une URL
  - Packagé dans un jar avec
    - Un point de lancement : interface Activator

```
public void start(BundleContext bc) throws BundleException;

public void stop(BundleContext bc) throws BundleException;
```

    - Des bibliothèques de code
    - Des ressources (code natif, documents…)

Hello World


```
package hello;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class HelloWorld implements BundleActivator {
  public void start(BundleContext bc){
    System.out.println("Bonjour");
  }

  public void stop(BundleContext bc){
    System.out.println("Au revoir");
  }
}
```

+

```
Bundle-Name: Hello World
Bundle-Description: The simple bundle
Bundle-Activator: hello.HelloWorld
```

Le Bundle est “démarré”

Le Bundle est « arrété »

=

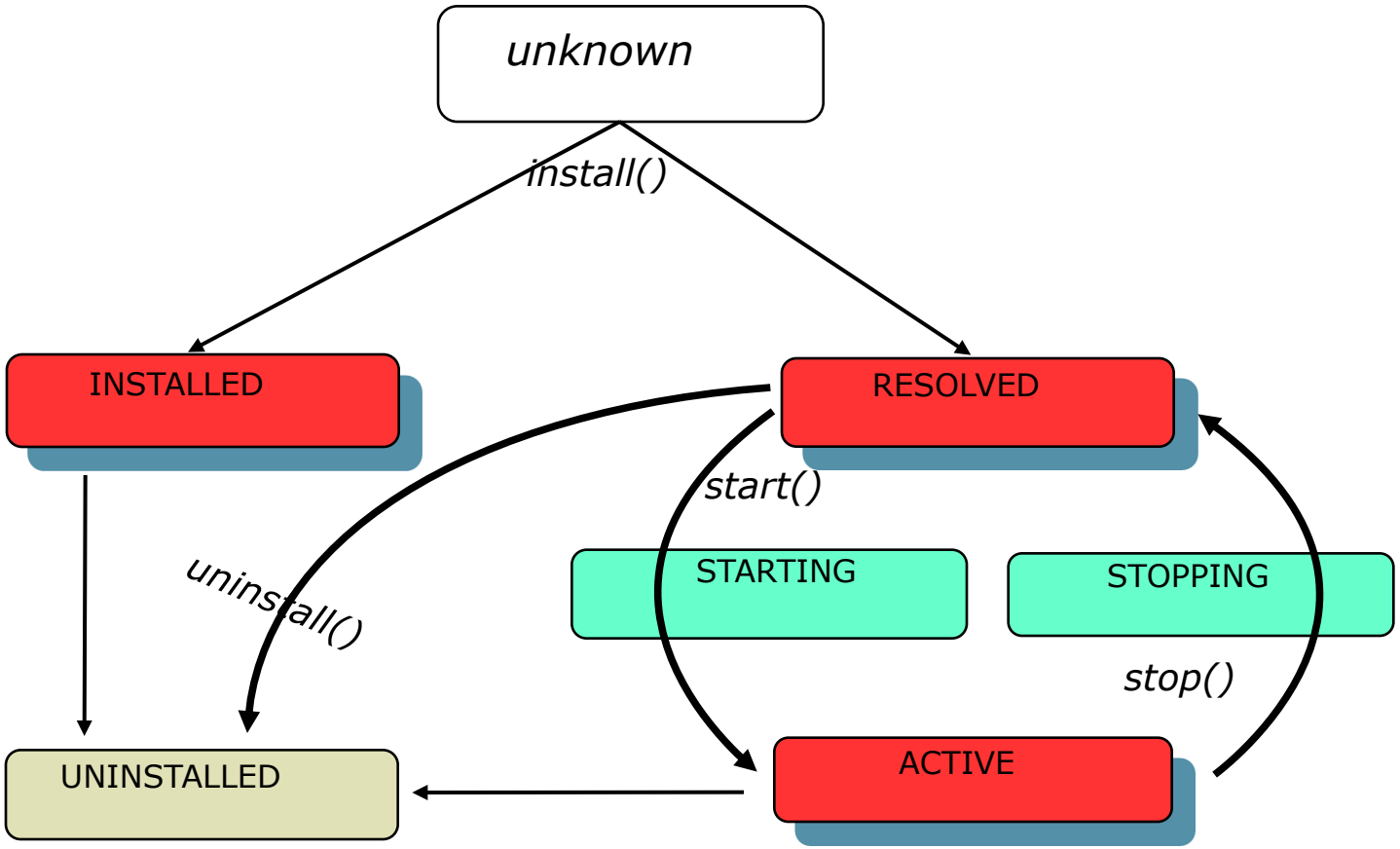
```
200 Tue Jun 15 16:20:00 CEST 2004 META-INF/MANIFEST.MF
723 Tue Jun 15 16:20:00 CEST 2004 hello/HelloWorld.class
```

Démarrage d'un bundle

1) Décompactage de l'archive dans un cache
2) Ouverture du manifest
3) Identification de la clé : Bundle-Activator
4) Création d'un chargeur de classe dédié
5) Instanciation explicite de la classe d'activation
6) Invocation de la méthode start dessus

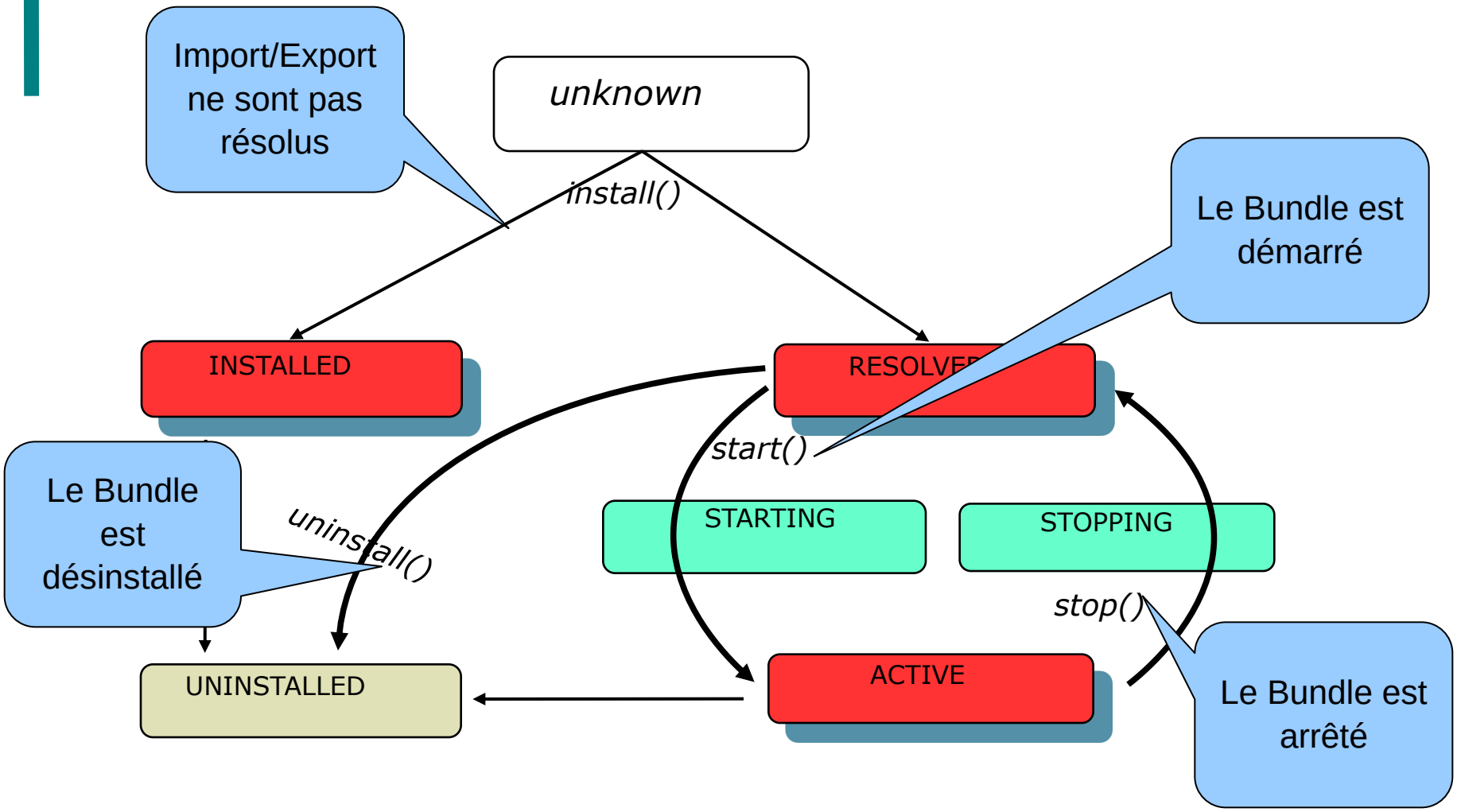
```
BundleClassLoader  myCL = new BundleClassLoader("file:/tmp/hello.jar");
String clazzName=myCL.getManifest().getBundleActivatorName();
Class myClass = myCl.load(clazzName");
BundleActivator  act = (BundleActivator)myClass.newInstance();
act.start(myCl.getBundleContext());
...
```

Cycle de vie : États d'un bundle



Cycle de vie :

Transitions



Exemple : service de log

- Une Interface de service

```
package logsystem;
public interface Log{
  public void log(String level, String message);
}
```

- Plusieurs implantations possibles...

```
package frenot;
public LeLog implements logsystem.Log{
  public void log(String level, String message){
    System.out.println(message);
 }
}
```

Exemple : un bundle de service de log


```
package logsystem;
public interface Log{
  public void log(String level,
     String message);
}
```

+

```
Bundle-Name: logger
Bundle-Description: un logger
Bundle-Activator: frenot.LeLog
Import-Package: org.osgi.framework
Export-Package: logsystem
```

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import logsystem.Log;

package frenot;
public LeLog implements Log, BundleActivator{
  public void log(String level, String message){
    System.out.println(message);
  }
  public void start(BundleContext bc){
    bc.registerService("logsystem.Log", this, null);
  }
  public void stop(BundleContext bc){}
}
```

=

```
200 Tue Jun 15 16:20:00 CEST 2099 META-INF/MANIFEST.MF
723 Tue Jun 15 16:20:00 CEST 2099 logsystem/Log.class
825 Tue Jun 15 16:20:00 CEST 2099 frenot/LeLog.class
```

Exemple : un autre bundle de service de log


```
Bundle-Name: superlogger
Bundle-Description: un autre logger
Bundle-Activator: superlog.SuperLog
Import-Package: logsystem,
org.osgi.framework
```

+

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import logsystem.Log;

package superlog;
public class SuperLog implements Log,
BundleActivator{
  public void log(String level, String message){
    System.out.println(level+ " : "+message");
  }
  public void start(BundleContext bc){
    bc.registerService(logsystem.Log.class.getName(),
this, props);
  }
  public void stop(BundleContext bc){
  }
}
```

Le package du log est fourni par un autre bundle

Des propriétés distinguent les deux implantations

Une implantation radicalement différente

```
200 Tue Jun 15 16:20:00 CEST 2099 META-INF/MANIFEST.MF
825 Tue Jun 15 16:20:00 CEST 2099 superlog/SuperLog.class
```

Lien entre service et bundles

- Un service peut être fourni par
  - 1 bundle
  - 2 bundles
    - Séparation de l'interface et de l'implantation
  - N bundles
    - 1 pour chaque implantation

➔ Il y a indépendance complète entre interface de service, implantations de services et bundles

Un service est fourni par

- Appel direct :

```
ServiceReference sr=
   bc.getServiceReference("logsystem.Log");
Log lelog=(Log)bc.getService(sr);
lelog.log("DEBUG", "it works !");
```

- Programmation réactive

```
bc.addServiceListener(this,
   "(ObjectClass=javax.log.LogInterface"));
public void serviceChanged(ServiceEvent serviceevent) {
   ServiceReference servicereference=
   serviceevent.getServiceReference();
   switch (serviceevent.getType()) {
     case ServiceEvent.REGISTERED :
        this.log=(Log) serviceEvent.getService();
        break;
     case ServiceEvent.UNREGISTERING :...
}
```

Packaging

- Deux types d'APIs
  - Services OSGi
    - Services du framework (administration, events...)
    - Services applicatifs
  - Services Java EE
    - Class A : composants dans un conteneur Java EE (EJB...)
    - Class B : servies annexes (JNDI, JMS...)
    - Class C : utilitaires (JAXB, JPA...)

Packaging

- Plusieurs types d'applications
  - Applications OSGi
    - Services OSGi
  - Applications hybrides
    - Services OSGi et Java EE
    - Packaging « classique »
    - Ajout de métadonnées OSGi

    ➔ Esprit Platform As A Service
  - Web Application Bundles (WAB)
    - Enterprise OSGi Service Platform Enterprise Specification, Version 4.2 [3].

Références

- OSGi
  - Cours de Stéphane Frénot (CITI, INSA Lyon), donné à l'école Intergiciel et Construction d'Applications Réparties
  - Projet Apache Felix : http://felix.apache.org/
  - Site officiel OSGi : http://www.osgi.org/
- Repository
  - Référence sur les bundles disponibles : http://www.osgi.org/Repository/HomePage

Conclusion

- Ne pas hésiter à faire appel à
  - Différents paradigmes de conception
    - Objet : bonnes pratiques
    - Aspects : complémentaire
    - Distribué...
  - Des outils facilitant la programmation
    - Frameworks
    - Bibliothèques de composants
    - Outils de déploiement...

Conclusion

- Penser composants dès les spécifications
  - Précision de la phase de conception et d'analyse (cahier des charges)
  - Modularité, design par interfaces, séparation des préoccupations…
- Rechercher l'existant avant de développer (bibliothèques disponibles)
  - Si l'interface d'une bibliothèque ne correspond pas à vos besoins :
    - Pouvez-vous / devez-vous modifier vos specs ?
    - Éventuellement, utiliser un pattern adapter
    - Sinon, le produit est-il fait pour vous ?

Conclusion

- Réutilisabilité : utiliser des solutions standard
  - Surtout si
    - vos applications s'insèrent dans un SI existant
    - d'autres peuvent devoir s'interfacer avec
  - Prévoir la possibilité de changer radicalement d'interface
    - RIA / RDA
    - Adaptation aux navigateurs / terminaux mobiles
    - Services Web

Conclusion

- Règles d’or de conception de SI
  - Modularité
  - Interopérabilité
  - Évolutivité
- Nécessite un plan, schéma, référentiel
- …Et une méthode de travail

Conclusion

- Grands principes de conception des SI
  - Focaliser sur les objectifs métier
  - Aborder le SI de manière globale
  - Prendre en compte l'évolutivité
    - Des infrastructures techniques
    - Des objectifs métier

➔ À la frontière entre conception et gouvernance