Is there really a way to modularize my application ?

OSGi

Modularity is our Business

What is OSGi ?

OSGi is a modularity framework :
an execution environment where modules are run.

In OSGi, modules are called *bundles*.

Yes, mom, he's telling us about modules and bundles... Did you ever hear of those things?

A bundle is a physical modularity unit.

```
// Bottles of Beer program Java version

class bottles
{
  public static void main(String args[])
    {
    String s = "s";
    for (int beers=99; beers>-1;)
      {
      System.out.print(beers + " bottle" + s + " of beer on the wall, ");
      System.out.println(beers + " bottle" + s + " of beer, ");
      if (beers==0)
        {
        System.exit(0);
        }
      else
        System.out.print("Take one down, pass it around, ");
      s = (--beers == 1)?"":"s";
      System.out.println(beers + " bottle" + s + " of beer on the wall.\n");
      }
    }
}
```

Code

Resources

Metadata

A module is a logical modularity unit.

Encapsulated implementation classes

Public API
Based on a subset of interfaces

A set of dependency
on external code

Onions have layers...

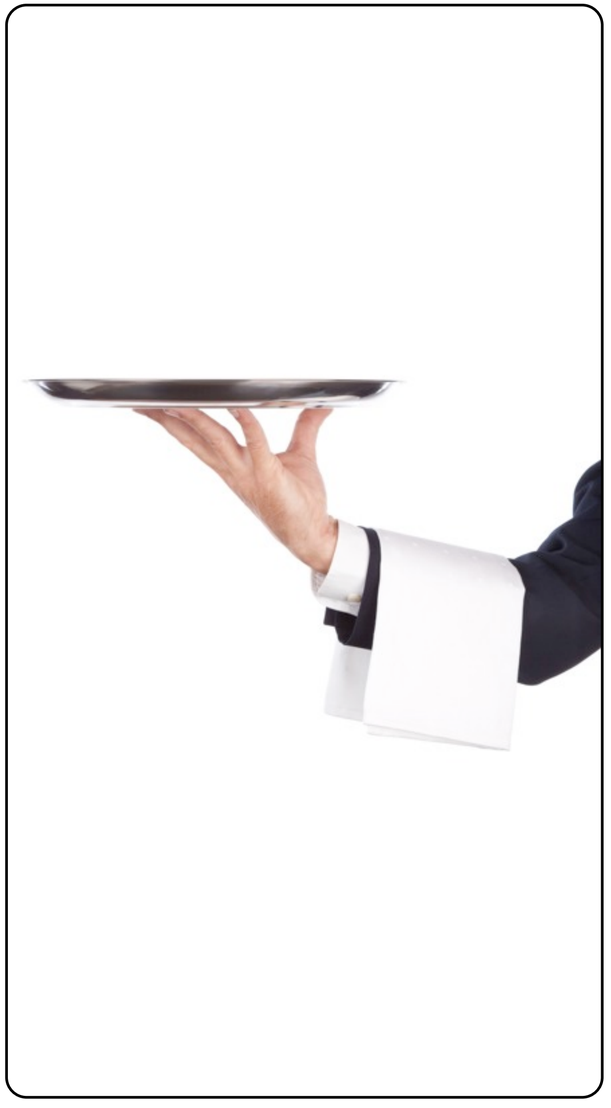
OSGi has layers !

Modules

manages packaging and code export/import

Lifecycle

manages bundles lifecycle
provides bundles an execution context



Services

allows interaction and communication between bundles

Modularity

The bundle modularity is defined in a manifest.

Human readable informations

Bundle Identity

Code visibility

Human Readable informations

(ignored by OSGi)

Bundle-Name

Bundle-Description

Bundle-DocURL

Bundle-Category

Bundle-Vendor

Bundle-ContactAdress

Bundle-Copyright

The Manifest

```
Property-Name: propValue; attr1=foo; dir1:=bar
```

Bundle identity

Bundle-SymbolicName

mandatory since OSGi R4

Bundle-Version

major.minor.micro.qualifier

Bundle-ManifestVersion

2 means OSGi R4+, defaults to 1

The Manifest

```
Property-Name: propValue; attr1=foo; dir1:=bar
```

Code visibility

Export-Package

comma separated list of packages

allows attribute discrimination and versionning

Import-Package

same as the export part, but version declaration allows ranges

Bundle-ClassPath

defaults to root (.)

can include pathes and embedded jars

The Manifest

```
Property-Name: propValue; attr1=foo; dir1:=bar
```

DEMO

basic exporting
basic importing

Bundle versions ordering

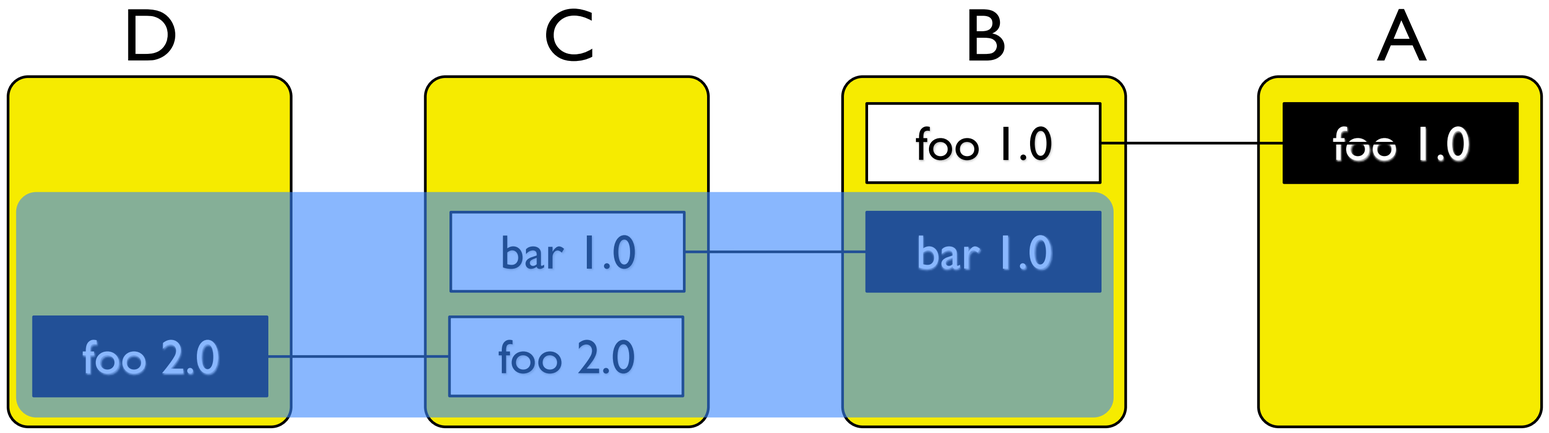
1.0.0 1.0.0.a 1.0.0.b 1.0.1 1.1.0 1.2.0

Bundle versions ranges

`"[min,max]"`	`min <= x <= max`
`"[min,max)"`	`min <= x < max`
`"(min,max]"`	`min < x <= max`
`"(min,max)"`	`min < x < max`
`"min"`	`min <= x`

Sorry, sir, you cannot get this package.
Your class space would become inconsistent !

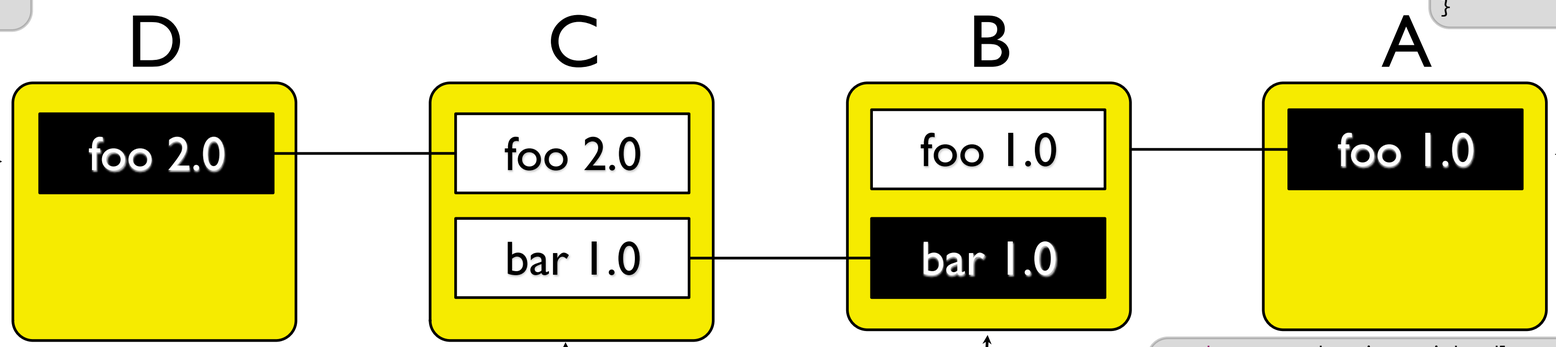
The "Uses" Constraint



POP QUIZZ

```
public class Foo {
        public void printIt() {
                System.out.println("Value is :
D");
        }
}
```

```
public class Foo {
        public void printIt() {
                System.out.println("Value is :
A");
        }
}
```




```
package org.phoenix.osgi.bundles.uses.activator;

import org.phoenix.osgi.bundles.uses.foo.Foo;
import org.phoenix.osgi.bundles.uses.bar.Bar;

public class Activator implements BundleActivator {
        public void start(BundleContext context) throws
Exception {
                Bar bar = new Bar();
        Foo foo = bar.getFoo();
        foo.printIt();
        }
}
```

```
package org.phoenix.osgi.bundles.uses.bar;

import org.phoenix.osgi.bundles.uses.foo.Foo;

public class Bar {
        public Bar() {
                super();
                System.out.println("Constructing
B");
        }

        public Foo getFoo() {
                return new Foo();
        }
}
```

Bar in B exposes Foo from A. C uses the Foo from Bar imported from D.
What is going to happen ?

Answer A :

Bundle C will use the Foo class provided by A.
Result will be : “Value is A”

Answer B :

Bundle C will use the Foo class provided by D.
Result will be : “Value is D”

POP QUIZZ

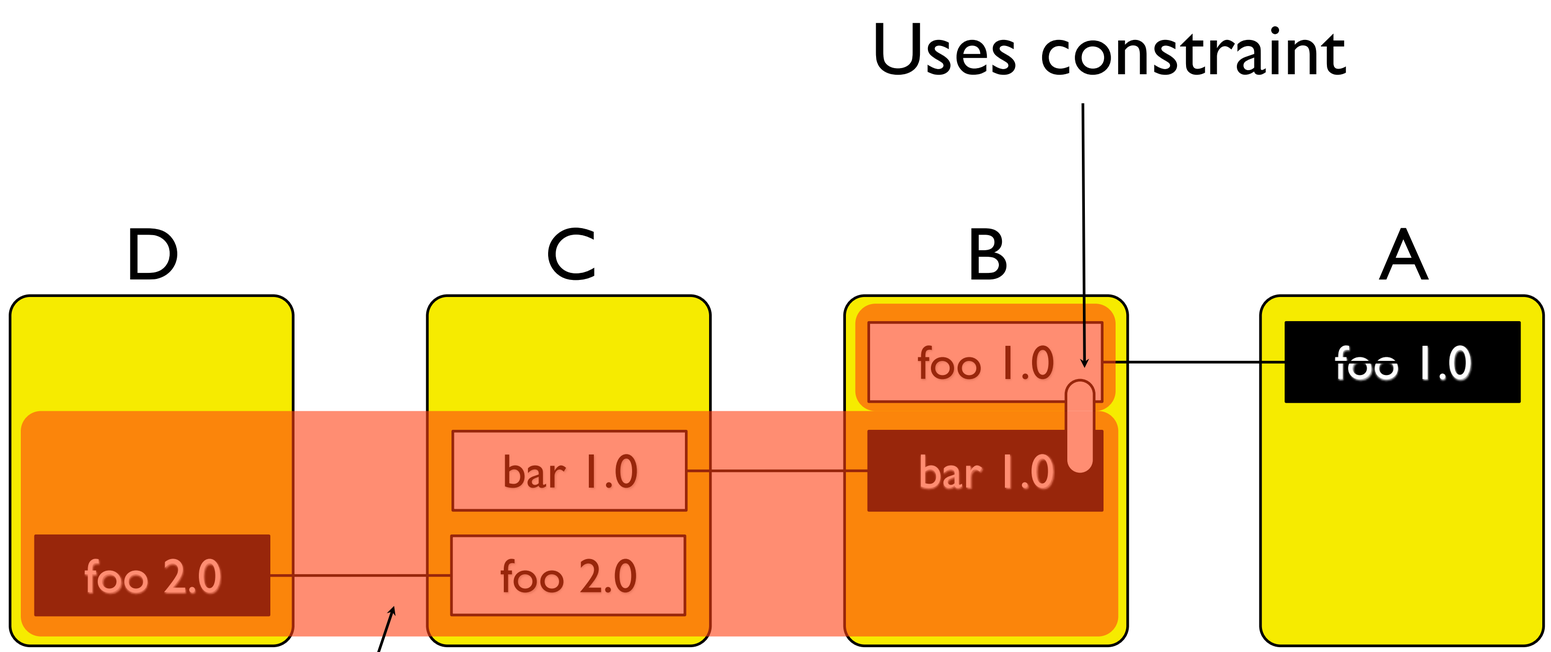
Secret answer C :

The Foo classes from A and D are not compatible.
A class identity is its fully qualified name plus
the classloader id.

The result is an error :

```
java.lang.LinkageError: loader constraint violation: loader (instance of org/eclipse/osgi/internal/baseadaptor/
DefaultClassLoader) previously initiated loading for a different type with name "org/phoenix/osgi/bundles/uses/foo/
Foo"   at org.phoenix.osgi.bundles.uses.activator.Activator.start(Activator.java:14)   at
org.eclipse.osgi.framework.internal.core.BundleContextImpl$1.run(BundleContextImpl.java:783) at
java.security.AccessController.doPrivileged(Native Method)  at
org.eclipse.osgi.framework.internal.core.BundleContextImpl.startActivator(BundleContextImpl.java:774)     at
org.eclipse.osgi.framework.internal.core.BundleContextImpl.start(BundleContextImpl.java:755) at
org.eclipse.osgi.framework.internal.core.BundleHost.startWorker(BundleHost.java:370)   at
org.eclipse.osgi.framework.internal.core.AbstractBundle.resume(AbstractBundle.java:374)      at
org.eclipse.osgi.framework.internal.core.Framework.resumeBundle(Framework.java:1067)   at
org.eclipse.osgi.framework.internal.core.PackageAdminImpl.resumeBundles(PackageAdminImpl.java:302)  at
org.eclipse.osgi.framework.internal.core.PackageAdminImpl.processDelta(PackageAdminImpl.java:546)   at
org.eclipse.osgi.framework.internal.core.PackageAdminImpl.doResolveBundles(PackageAdminImpl.java:239)     at
org.eclipse.osgi.framework.internal.core.PackageAdminImpl$1.run(PackageAdminImpl.java:174)   at java.lang.Thread.run
(Thread.java:680)
```

The "Uses" Constraint



Dependency not resolvable

DEMO

uses-demo

A word about fragments

Fragments are special bundles. They depend on a host bundle.
The host is declared in their manifest : Fragment-Host

The host can be run without any fragment, but a fragment needs it host.

Fragments are merged into their host.
- merged content
- merged metadata
- merged classloader (one)
- merged bundle-classpath

Use case : localization. Add as many fragments containing i18n files as needed : add any language support @runtime

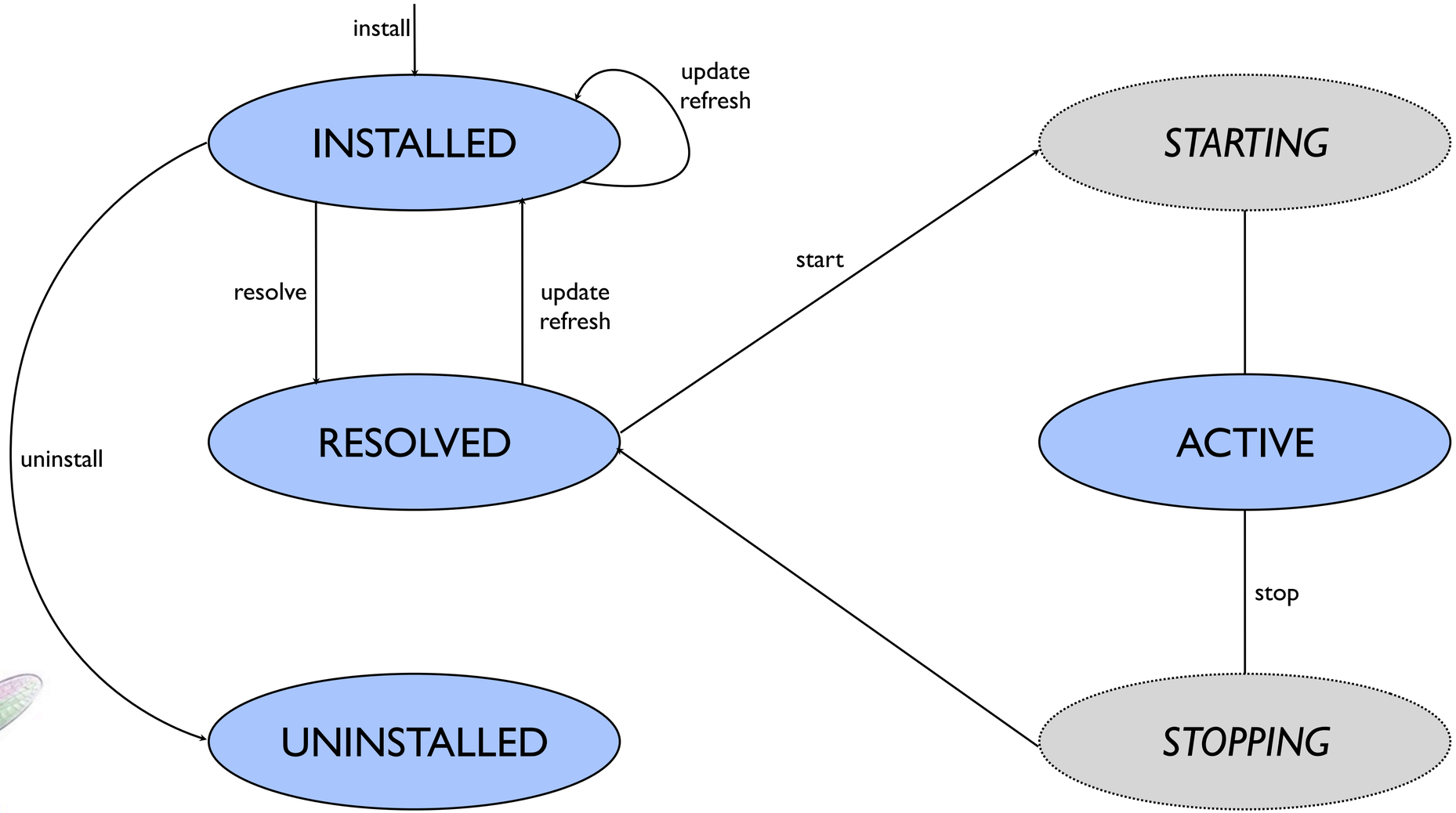
But this is just an example, you can services, packages, resources...

DEMO

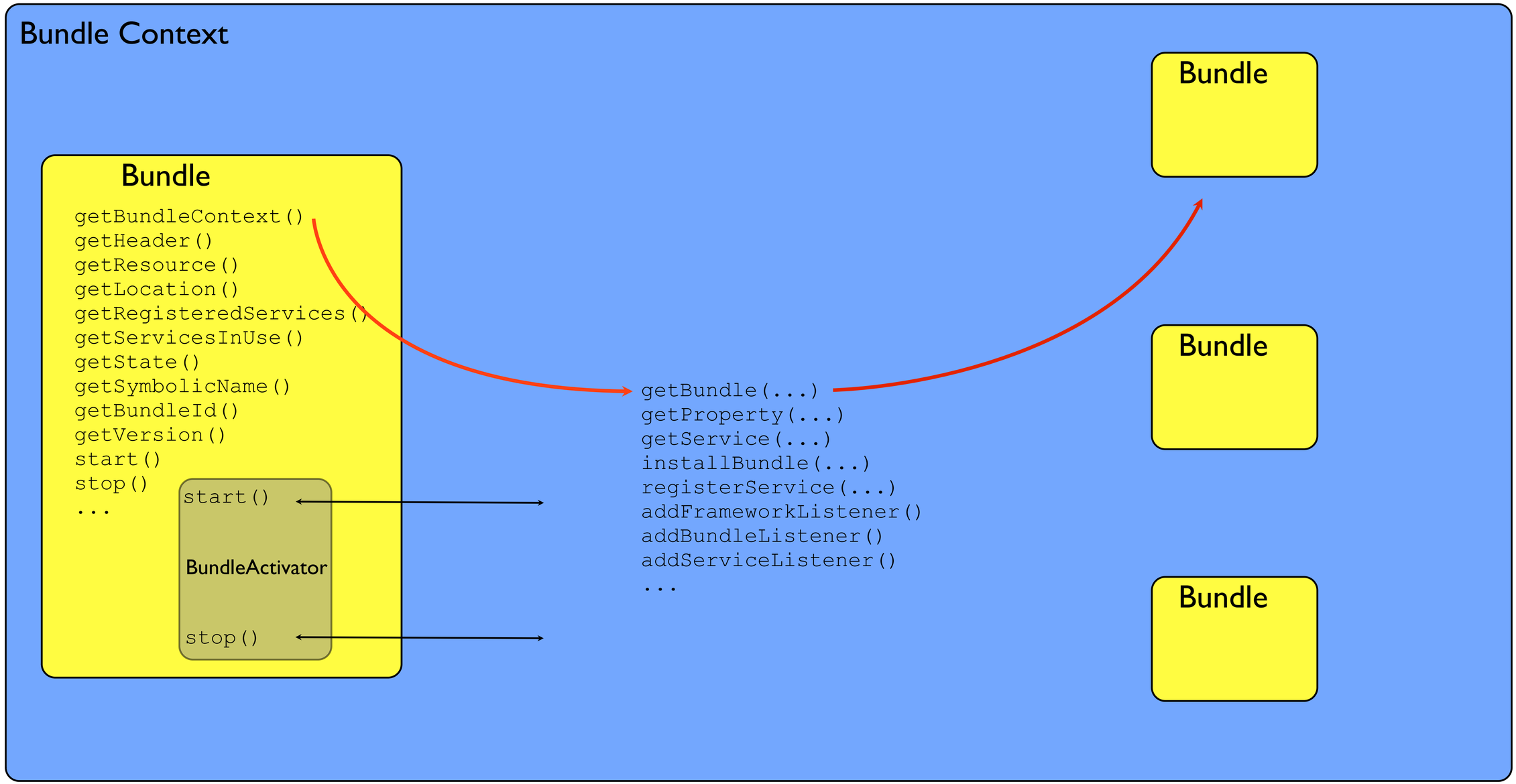
fragment-demo
logger-using-fragment

Lifecycle

A bundle's life



Lifecycle layer allows a bundle to interact with its context.



Oh my god !
A bundle juste went away !

OSGi events

OSGi events

OSGi notifies listeners for different events :

FrameworkListener

frameworkEvent(FrameworkEvent fe)
STARTED, INFO, WARNING, ERROR, PACKAGES_REFRESHED

BundleListener

BundleListener : bundleChanged(BundleEvent be)
INSTALLED, RESOLVED, STARTED, STOPPED, UNINSTALLED, UNRESOLVED

Events notifications are asynchronous, except for :

SynchronousBundleListener

bundleChanged(BundleEvent be)
STOPPING, STARTING
Notified synchronously and before BundleListener.

The Extender pattern

Here, this is my contribution to your application !

The Extender pattern

OSGi pattern for extending an application when a bundle is started.
Uses a special bundle called the *Extender*.

Listens to synchronous events (STARTING/STOPPING),
Reads bundle manifest headers or content,
Checks it the bundle is an extension,
Performs extension operations.

```
Extension-Name : MyExtension
Extension-Class : org.phoenix.osgi.extension.MyExt
```

The extender is called XXXTracker

Extends BundleTracker (standardized in OSGi R4.2)
addingBundle(...), removingBundle(...), modifiedBundle(...)

Can define a BundleTrackerCustomizer to customize, select tracked bundles.

Services

Service

A work done for another, a contract between the caller and the callee.
Implementation and provider are not important, as long as the contract is respected.

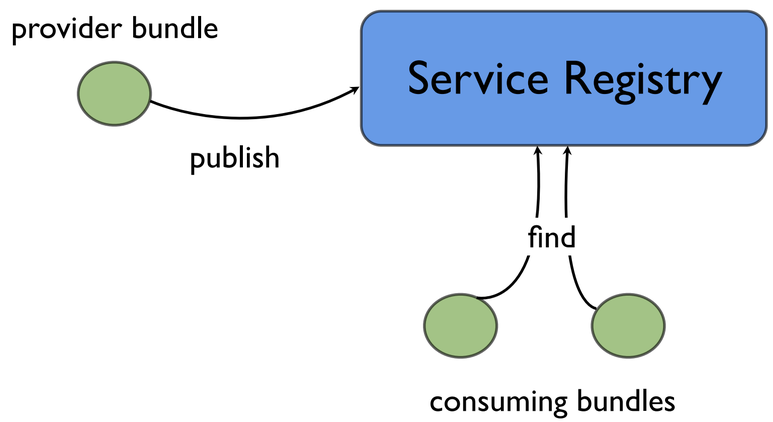
The interface defines the contract. The choice is made based on interface and metadata.
Spring allows choice only based on the interface.

It supports multiple implementations.

OSGi services are dynamic : the framework provides techniques and utility classes to handle that.

OSGi service registry :
publish-find-bind model

Providing bundles publishes POJOs as services.
Consuming bundles find and bind them.
The registry is accessible from the bundle context.



Publishing a service

Publishing a service is done by providing the service registry with an interface (or an array of interfaces), a service implementation, and optionally a dictionary of metadata.

```
String interfaceName = MyServiceInterface.getClass().getName();
Dictionary metadata = new Properties();
metadata.setProperty("country", "FR");
ServiceRegistration registration = bundleContext.registerService(
interfaceName,
myService,
metadata);
```

Finding a service

```
ServiceReference reference = bundleContext.getServiceReference(MyServiceInterface.getClass().getName());
```

Returns a reference to the service, allowing the framework to separate the use of a service from its implementation.

A bundle holding a reference to a service may use different implementations from time to time without even noticing !

Filtering can be achieved using LDAP filter strings :

```
attribute matching:    (name=foo), (country=fr)
wildcard:              (country=f*)
attr existence check:  (name=*)
match all:             (&(name=foo)(country=fr))
match at least one:    (|(name=foo)(country=fr))
negation:              (!(country=fr))

ServiceReference reference = bundleContext.getServiceReference(MyServiceInterface.getClass().getName()
, "(&(country=fr)(dws=yes))");
```

Binding a service

Binding a service is getting the implementation behind the reference.

```
MyServiceInterface myService = bundleContext.getService(myServiceReference);
```

WARNING

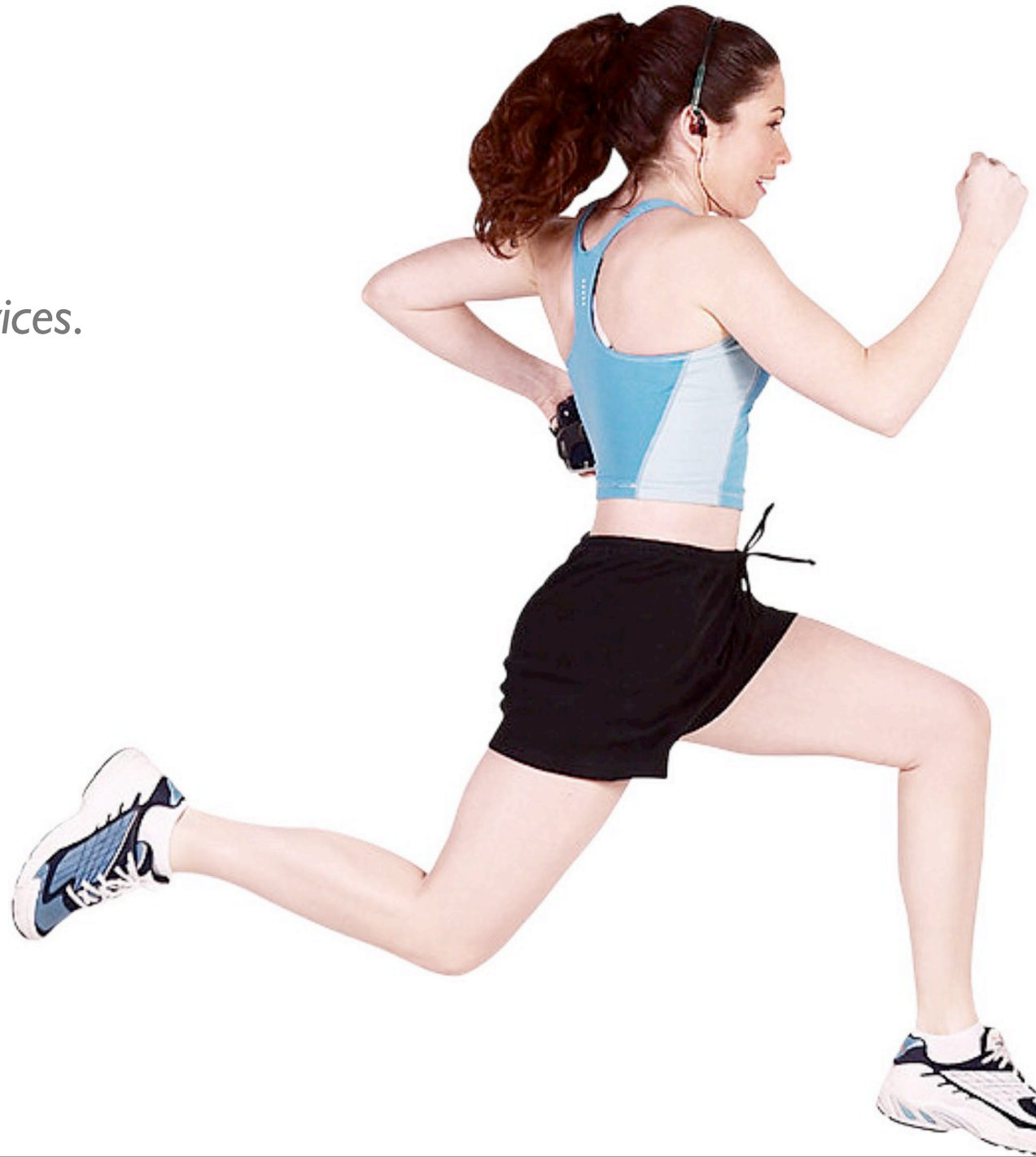
Services are NOT proxies !
They must be freed as soon as possible :
Help GC
Avoid keeping pointers to services when they are gone

Freeing a service is a two steps process. "Nullify" the service pointer, and unget the reference.

```
bundleContext.ungetService(serviceReference);
myService = null;
```

Catch me if you can !

Dealing with the dynamic nature of services.



Dynamic services pitfall #1

Storing a service into a field !

Services may come & go. Calling a service that is "gone" can produce strange results/exceptions.

Store the service reference instead of the service.

Dynamic services pitfall #2

Get a service reference at startup (Activator)

The service may not be present at this time. Dereferencing the reference can return null if service goes. The service reference should be retrieved when needed only.

Store the bundle context instead of the service reference.

Dynamic services pitfall #3

“Race condition” : OSGi is multithreaded

A service may go between the reference and
the service implementation rerieval
(between getServiceReference(...) and getService(ref))
Check that reference AND service are not null.

Dynamic services pitfalls solution

```
BundleContext ctx; // retrieved earlier : pitfall #2

public void doSomethingInABundle() {1
   ServiceReference reference = ctx.getServiceReference(MyInterface.class.getName()); // Use reference : pitfall #1 (~)
   if (reference != null) {   // pitfall #3
      MyInterface service = bundleContext.getService(reference);
      if (service != null) {  // pitfall #3
          // Yipee, call the service now !
          service.doSomethingRealUseful();
      }
   }
}
```

Seems correct, huh ?

But is it?

Well, not really...

What is wrong ?

When the service is there, the service reference returns a service.

OK

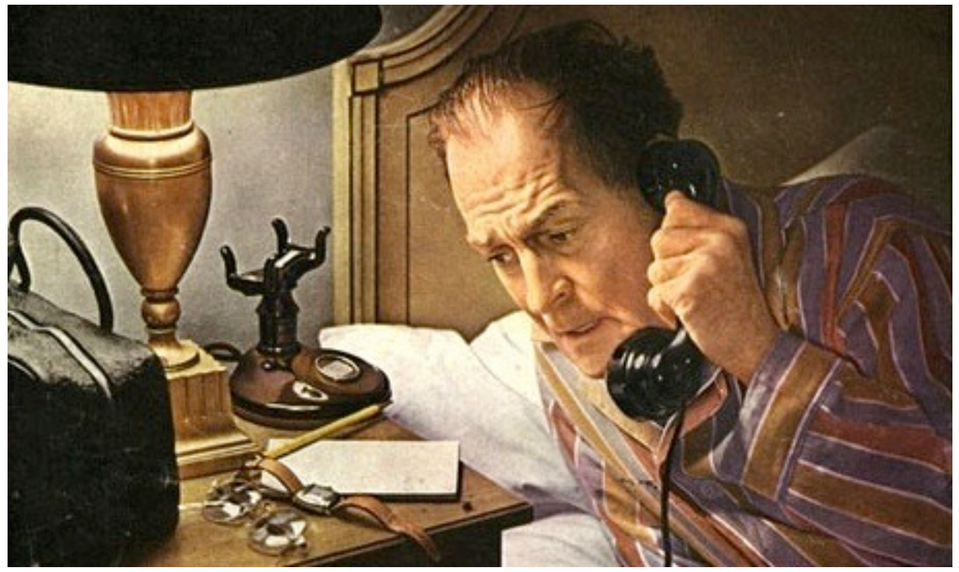
But if service is gone, we get null.

Sad, but OK.

But...
What if another bundle comes and provides a service implementing the same interface ?

Dereferencing the reference still returns null :(
We just missed a service that could have done the job !

What? They changed my service ?



Service Listeners

A service listener listens to service events.

REGISTERED, MODIFIED, UNREGISTERED

```
void serviceChanged(ServiceEvent event);
```

```
bundleContext.addServiceListener(new MyServiceListener(), filter);
```

MyServiceListener implements ServiceListener
The filter (ldap syntax) allows to restrict the listened services.

```
Ex : filter="(objectClass="+MyService.class.getName()+")"
```

With a listener, bundles can handle service dynamic nature.

Now seems cool, huh ?
But is it ?

Well, not really...

Ok, so what's wrong now ?

Well a few things :

1. OSGi is multithreaded. serviceChanged can be called concurrently...
2. OSGi is dynamic. What about the services registered before the listener is added?
3. When several matching services are available, which one should we choose?
4. How exactly do we switch implementation if one goes?

How to solve these points ?

1. synchronize serviceChanged

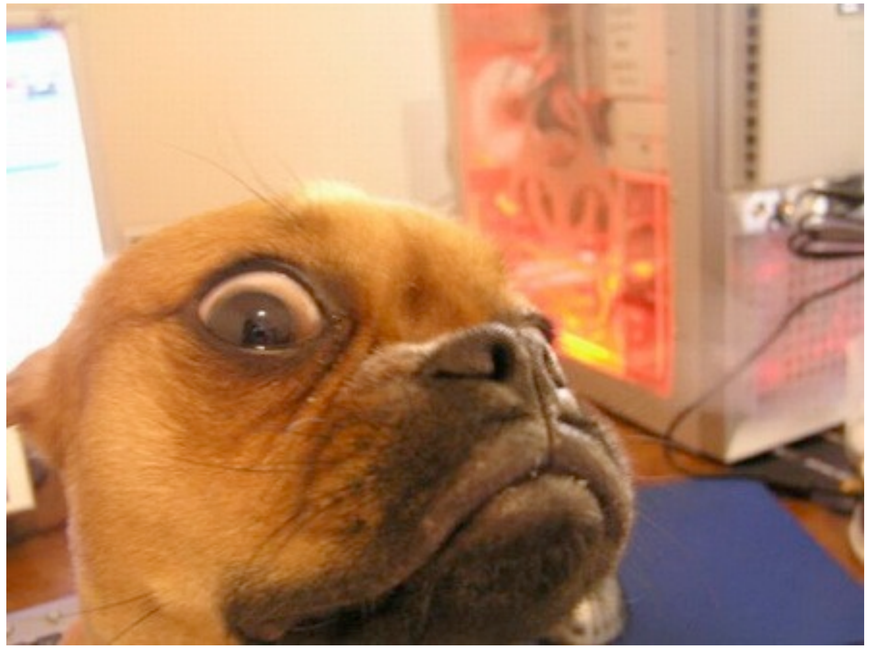
2. send pseudo registration events for already registered services

3. & 4. Keep an ordered set of available services. The natural ordering of services is ranking(highest) + id(lowest).
Allows to select the best service by calling last() on the set,
and to get another service from the set if the current one goes away

That's just a hell of a boilerplate code !
Couldn't anything help me ?

ServiceReference,
implementation,
help GC,
unget service reference,
check nullity of service and service ref,
add a listener,
synchronize,
send pseudo events,
keep a set of services,
change service when it goes,
listen to others,
order them...

I'm watching you !



Service Tracker

ServiceTracker

The service tracker handles all previously mentioned points.
It registers a listener for a service, manages the list of tracked services, handles the service events, returns the highest ranking service when asked...

One can register a service tracker based on a service reference, a service class name, or a filter.

```
// Init
ServiceTracker tracker = new ServiceTracker(bundleContext, MyServiceInteface.class.getName(), null);
tracker.open(); // MANDATORY ! Without this, tracker does not initialize, so getService() always returns null

// Use
tracker.getService();
// Or
tracker.getServices();

// Close
tracker.close(); // When done,
```

ServiceTracker : bonus

The service tracker offers a little bonus : the tracked service object can be customized.

It allows to enforce some business filtering rules, or customize the tracked service, through a ServiceTrackerCustomizer.

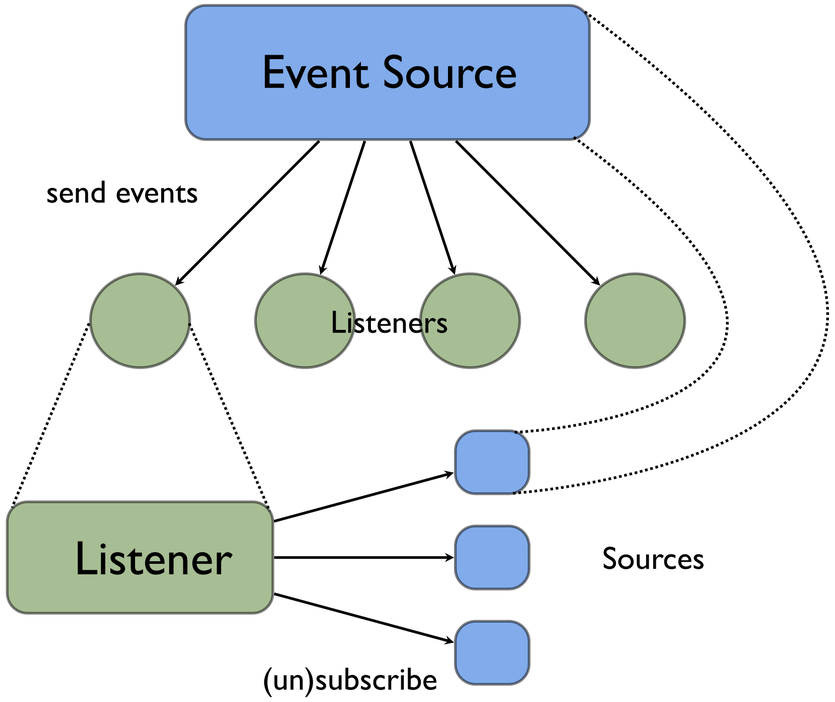
```
public class MyDecorator implements ServiceTrackerCustomizer {
    @Override
    public Object addingService(ServiceReference ref) { // reacts to a new service added
        // create a new implementation of the tracked service
        return new MyServiceInterface() {
           @Override
           public void doIt(String arg) {
               // Customize the tracked service
               ((MyService)bundleContext.getService(ref).doIt("extra text : "+arg));
           }
        };
    }
}

ServiceTracker tracker = new ServiceTracker(bundleContext, MyServiceInteface.class.getName(), new MyDecorator());
```

The Whiteboard pattern

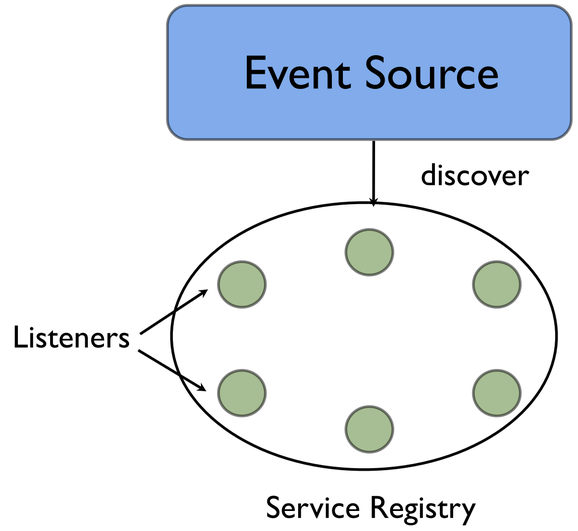
Powered by ServiceTracker

The Whiteboard pattern

A substitute to the listener pattern, using services.
Saves codes managing sources, list of listeners, (un)registration of listeners...
The sender asks OSGi for listeners (matching services).



The Listener pattern



The Whiteboard pattern

ConfigurationAdmin Service

Allows to create and edit services configurations @runtime

Service is configured when a configuration is created.
When configuration changes, service is notified.
Implements : ManagedService

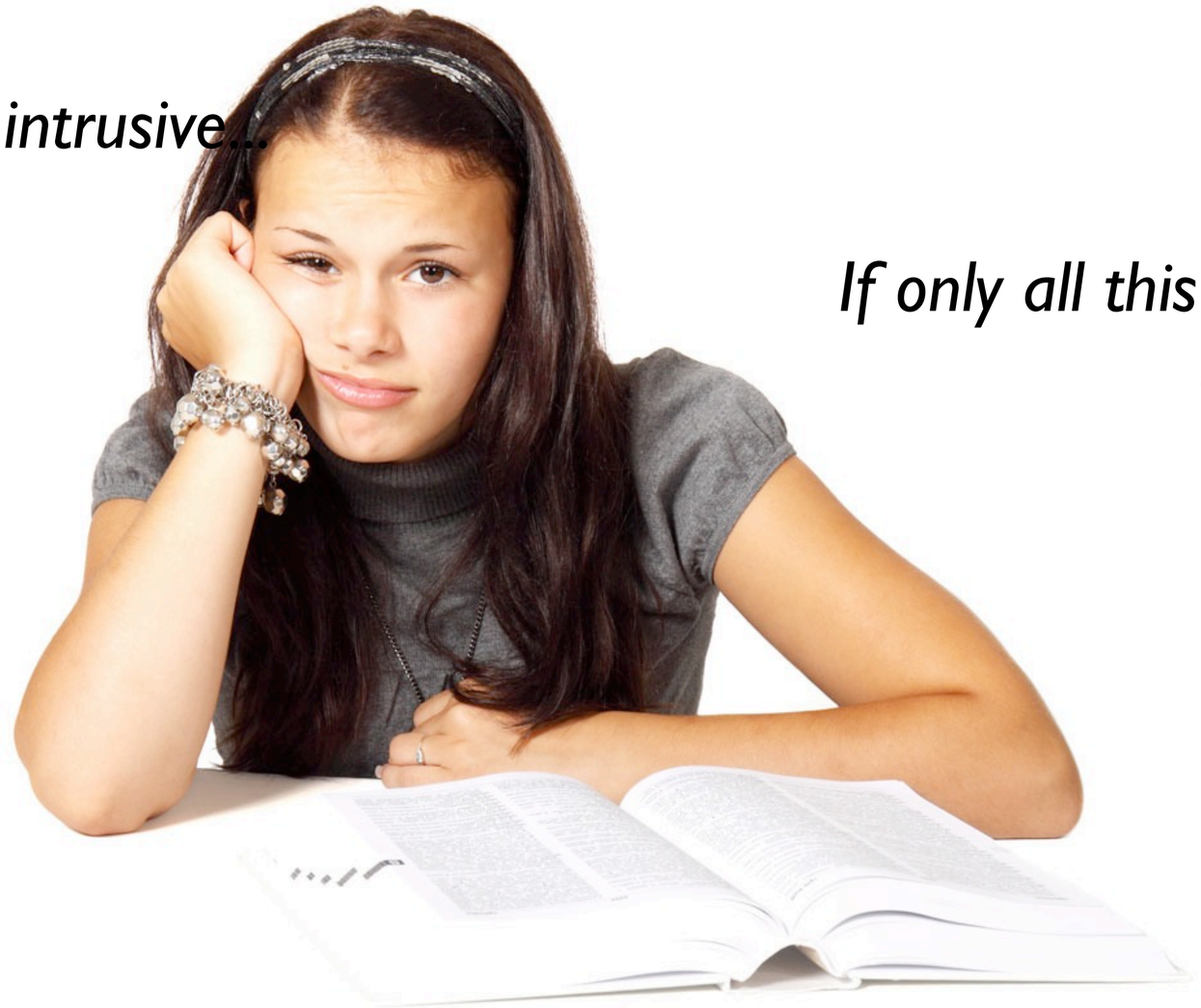
ServiceFactories : Creates an instance of the service for each created configuration.
When configuration changes, service is still notified.
Implements : ManagedServiceFactory

DEMO

managed-service
managed-service-user

service-factory
service-factory-user

Ok OSGi is great, but it's intrusive...



If only all this could be done for me...

Component Frameworks

Creating the magic

OSGi Alliance Declarative Services
Apache iPOJO
Spring OSGi
Spring Dynamic Modules
Eclipse Gemini Blueprint
(OSGi v4.2 BluePrint Service RI)

But that's another story...

OSGi in context

1998 : JSR 8 (Open Service Gateway initiative)

Withdrawn in 1999

2005 : JSR 277 (Java Module System)

Goal : define a module framework, packaging format and a repository system for Java
Reinvents the wheel, ignoring OSGi work

2006

JSR 291 (Dynamic Module System Support for Java)
Goal : Bring OSGi into JCP standardization. Completed quickly, resulted in OSGi R4.1 release.
Eclipse Equinox : JSR 291/OSGi R4.1 RI

JSR 294 (Improved Modularity Support in the Java Programming Language)
Extracted java languages modifications from JSR 277.
Introducing super packages (idea abandonned), then the 'module' keyword

2007

JSR 294 merged back into JSR 277

2008

JSR 277 put on hold indefinitely
JSR 294 extracted again from JSR 277
Sun introduce project JIGSAW (modularization of the JDK)

JSR 277

- Static module system (similar to OSGi Required-Bundle)
- Metadata defined in MODULE-INF/METADATA.module
- Dependencies defined in import clauses with module symbolic name and versions range
- Other clauses list resources and classes visible to other modules
- Modules must be resolved w/ their dependencies before the app can be run
- Modules definitions are obtained from repositories. Resolution based on name+version
- Optionally an Import policy class can define a more complex resolution strategy
- Repositories are accessed as files/urls, hierarchically linked (parent delegation if module not found)
- Modules instantiated before they can be used, associating a classloader to the module
- When all modules are resolved and instantiated, main class is called. Modules cannot be unloaded.

JSR 277

- Simplistic and "toyish" (cf Peter Kriens)
- Ignores 8 years of OSGi work
- Uses a delegating classloader model (like OSGi), without using OSGi experience
- No package sharing
- NO DYNAMICS (no dynamic (un)loading) : any change = reboot
- NO CONSISTENCY CHECK
- Resolution based on name+version only. No class space consistency check
- There is a deepValidate() method in Module class : brute force loading of all classes in all modules checking for ClassNotFoundException/ClassCastException
- Extremely expensive, and only checks loading problem, not consistency problem
- "Required-Bundle" problem : loading ordering (if bundle 2 imports bundle 4, then order is 1243)
- Split packages problem : class form part 1 of package cannot see package-private classes of part 2
- Import cannot control imports : only gives name+version
- Module A has packages pa and pb. Module C depends on both packages
- Module A is refactored : pa in A and pb goes in module B. All modules depending on A must be modified to depend also on B.
- Import policy class : could be nice but executed during resolution so :
- Module is NOT resolved : bound to encounter problems (missing classes, resources...)

JSR 294

Abandonned idea : superpackages

- Super packages : list member (super)packages and exported types
- Public members of member packages are accessible in other member packages (but not outside)
- No versionning
- Declaration of member packages must be exhaustive (no wildcard allowed)
- Exported types must be declared also (wildcard allowed : all or nothing)

```
superpackage my.package {
export my.package.foo.*;
export my.package.bar.MyInterface;
member superpackage my.package.subpack;
member package my.package.toto;
member package my.package.tutu;
}
```

New idea : module keyword

- Adds a module visibility constraint.
- A usable class must be visible. module can prevent it from being visible.

```
module class MyClass {
...
}
// MyClass is accessible from other modules but not visible, so cannot be used
// module can also be applied to fields, methods
```

Details still evolving. Now Peter Kriens is in the expert group.

NOTE : JSR 294 is NOT a modularity framework : it’s an evolution of the language to enforce visibilty rules
Details still evolving today...

JIGSAW

Goal is to modularize the JRE/JDK, not to create a modularity framework.
JIGSAW RI : OpenJDK7 (targets Java 8)

Modules : deployment artifacts (JAR)

Dependency model : based on Maven model (similar to OSGi Required-Bundle)
➠ Transitive dependencies inheritence
A depends on B, which depends on C (which depends on ...) and D (which depends on ...)...
Maven downloads the internet :)
➠ Consistency problem
A depends on B (which depends on D-1.0.0) and C (which depends on D-2.0.0)
Class space inconsistency
➠ “fidelity across all phases” (quoted from Mark Reinhold Jigsaw requirements document)
➠ compile time dependencies must be the same at runtime
➠ implementations must be chosen at compile time
➠ but we only need to *compile* over an API, then *run* over an implementation that is *compatible* with the API
➠ Jigsaw(maven) model makes resolution as simple as : repositoryURL+name+version
➠ But a package-based dependency model requires an indirection mapping between a package
and an artifact providing it, thus allowing different dependencies between compile and runtime.

What modularity framework can i choose ?

OSGi : Yes

JSR 277 : on hold indefinitely, and has severe problems already.

JSR 294 : not a modularity framework

JIGSAW : not a modularity framework

Questions ?

http://www.osgi.org

http://njbartlett.name
http://www.aqute.biz
https://github.com/fesnault/technoshare-OSGi