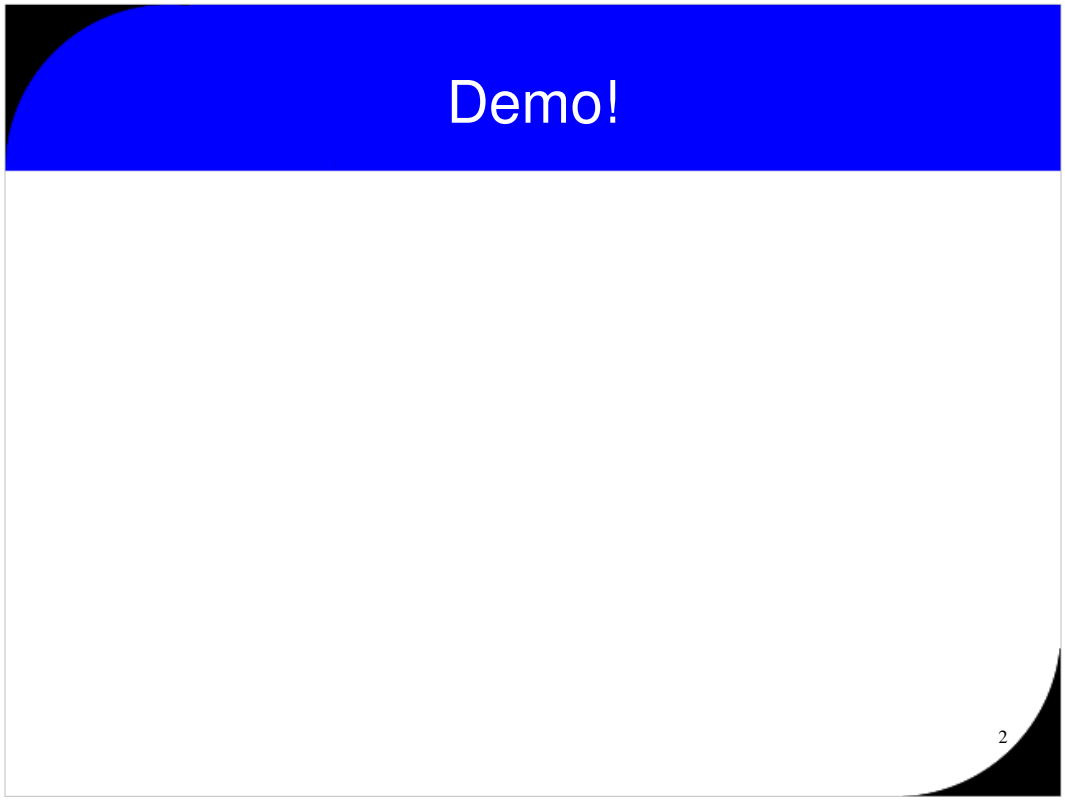
Groovy and OSGi Jumpstart

Hamlet D'Arcy
Pearson

@HamletDRC
hamletdrc@gmail.com
http://hamletdarcy.blogspot.com

OpenOffice Slides: http://is.gd/2Ymlz

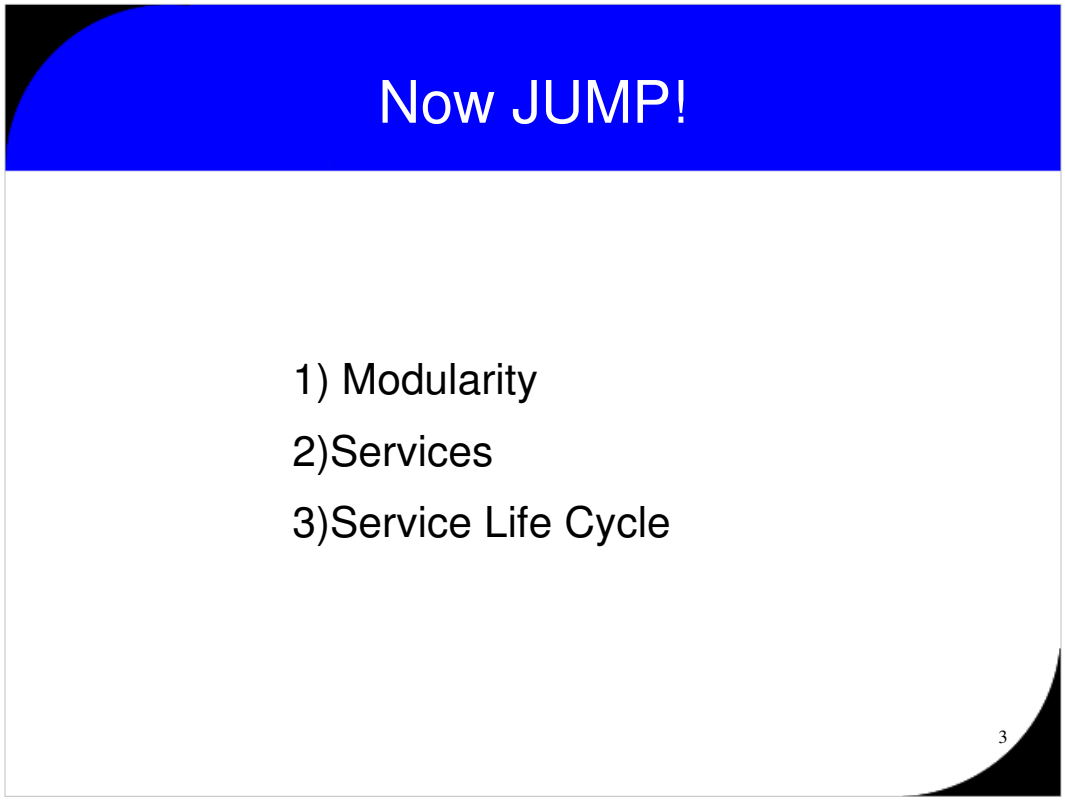


1. Several versions of Groovy are running in same JVM (from same thread?)

2. Groovy libraries are exposed as a services. UI is consuming whatever Groovy services are present

3. UI sees services coming and going and can respond in like

4. 229 Lines of Code (147 of which are Swing). 0 lines of XML. No languages changes or new keywords. No waiting for next JDK/Java. Books published. Open specification. In production. Build tool and IDE support. Many testing solutions.



Concepts of the demo are not hard to understand when put concretely. Why is OSGi so confusing

Oh God! (remember George Burns?)

Worst... Table... of... Contents... Ever...

1) History of OSGi
2) The OSGi Core Specification
3) Handsets and Mobile Devices
4) 10 minutes of how it works on the server
   (if you're lucky)

Modularity

Why Modules

Neil Bartlett on designing a 747:

"The only approach that will allow human beings to design such incredible machines is to break them down into smaller, more understandable modules"

Modularity solves the problems of large code bases.

This makes an assumption: human beings should be actively involved in the design of large projects. But for now we are.

What Problems are We Solving?

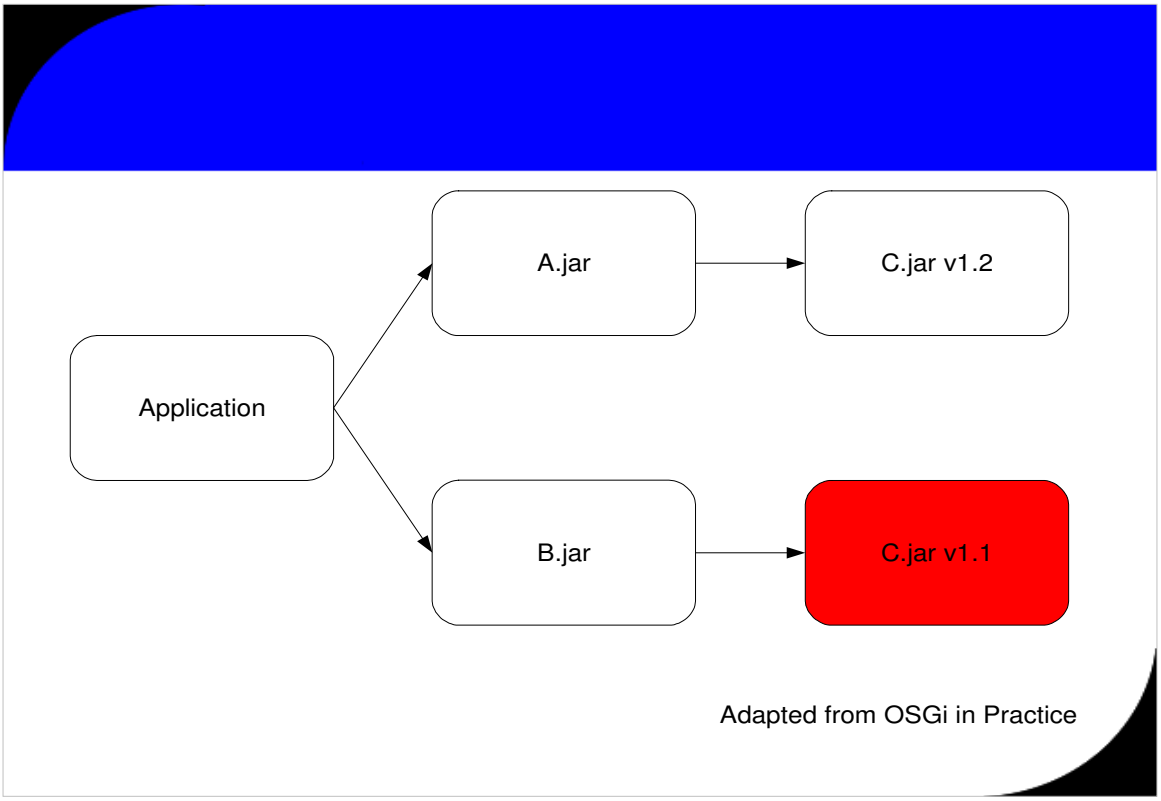
- JAR is not a runtime concept
- JAR contains no standard metadata
- JAR contains no standard version info
- JAR provides no information hiding

The JAR is not a runtime concept – meaningful only at build and deploy time. Contents of all jars are concatenated and treated as a global list. Does not scale, leads to jar-hell: when jars shadow one another.

JARs contain no standard metadata – needed to indicate dependencies. You can do this in Jar metadata (OSGi) or in source (Jigsaw). MANIFEST.MF Class-Path attribute is useless in practice.

JARs contain no standard version information – No standard way to declare your version or the required *version range* of your dependencies.

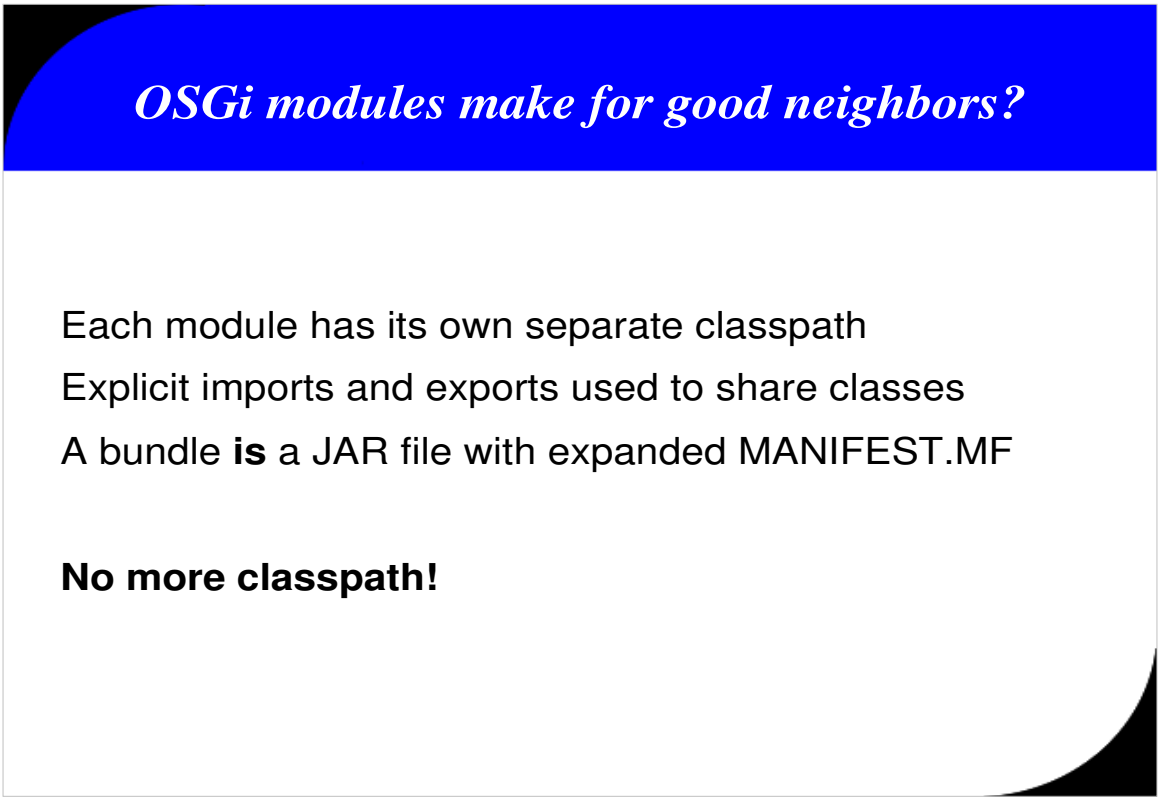
JARs provide no information hiding – public/protected/private indicate visibility across packages, but the unit of deployment is not a package, it is a Jar file. The whole Jar is effectively the public API.


A.jar
C.jar v1.2
Application
B.jar
C.jar v1.1
Adapted from OSGi in Practice

Self-Contained – moved, installed, and uninstalled as a single unit

Highly Cohesive – One purpose

Loosely Coupled – Should not depend on implementation of other modules



Which OSGi bundles are you using today?

OSGi Modules (called a “bundle” or a “plugin” in Eclipse)

Each module has its own classpath, separate from that of other modules.

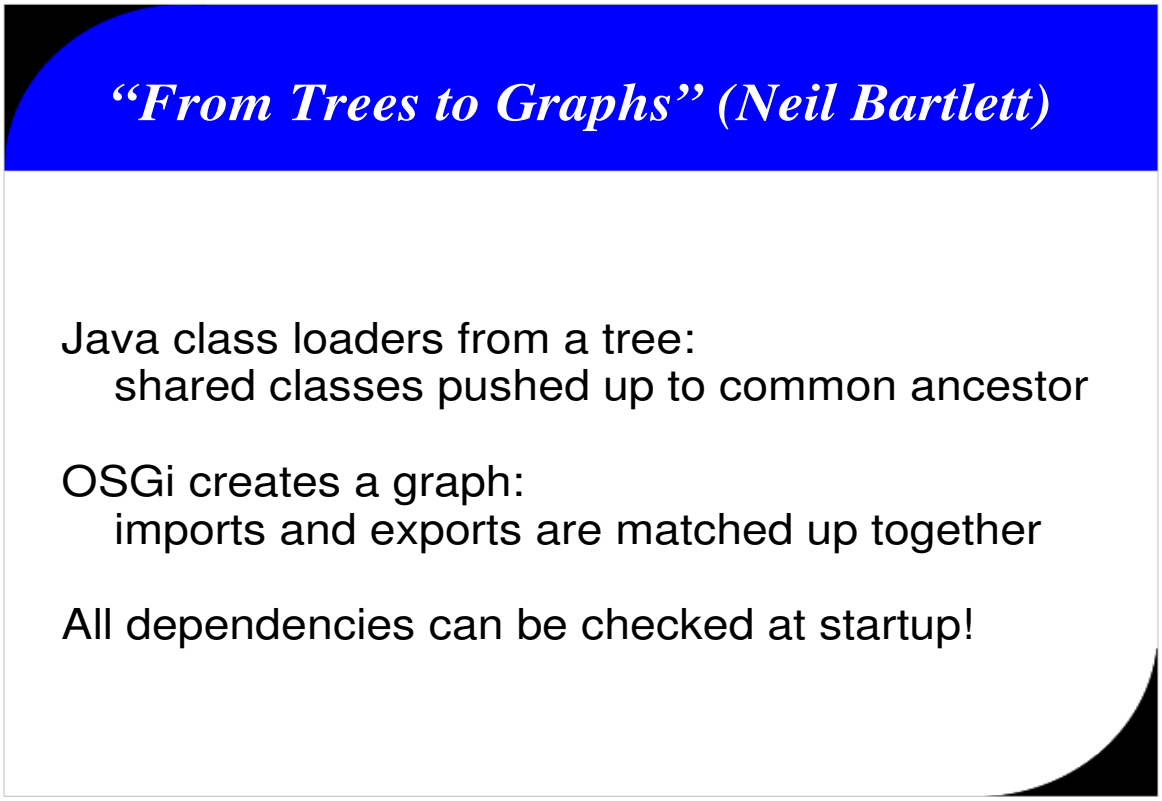
Explicit imports and exports used to share classes across modules

A bundle is a JAR file with with extra data in MANIFEST.MF

Symbolic Name, version, imports, and exports

Bundles can be used anywhere JAR files are used.

No more classpath. Classpath is not part of the Java language or platform. It is a convenience added by compiler writers.



A unique class = qualified class name + classloader
Each bundle has it's own class loader
Therefore, each bundle can have their own versions of the same class.
Java classloaders form a tree: to share a class, it is pushed up to a common ancestor, where everyone sees that class.
The OSGi resolution process creates a graph – imports and exports are matched up together. There is no grand hierarchy or parent/child.
Side Effect: OSGi apps can tell at start-up when all dependencies can't be found. No longer a runtime error.

Each Bundle has Own Class Loader

- Information Hiding – packages are bundle private by default.
- Versioning – different versions of same class side by side
- "ClassCastException: Cannot cast com.example.Service to com.example.Service"

Information Hiding – all packages are bundle private by default. You control at a fine level what users see in your API

Versioning – You can have different versions of the same class side by side in one application

"ClassCastException: Cannot cast com.example.Service to com.example.Service"

"I am Accessible, but am I Visible"

Visibility:

whether one type can see another type

Accessibility:

whether one type can access another type

OSGi provides modularity thru restricted visibility

It is possible for a type to be visible but not accessible.
http://blog.bjhargrave.com/2009/03/i-am-visible-but-am-

Creating a Valid Bundle

Build JAR with code and Groovy-all.jar within

Add Bundle-SymbolicName to MANIFEST.MF

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 10.0-b19 (Sun Microsystems Inc.)
Bundle-SymbolicName:
org.example.ValidBundle
```

example: ./eg/validbundle

Notice: no compile time dependency on OSGi

Creating a Usable Bundle

Build JAR with code and Groovy-all.jar within

Add Exports to MANIFEST.MF

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 10.0-b19 (Sun Microsystems Inc.)
Export-Package: org.example;version="1.0.0",groovy.lan
g;version="1.6.4",org.codehaus.groovy.runtime.callsite;versi
on="1.6.4"
Bundle-ClassPath: .,groovy-all-1.6.4.jar
Bundle-SymbolicName: org.example.UsableBundle
```

Build a Jar containing code and Groovy-all.jar within
Add Bundle-SymbolicName attribute to MANIFEST.MF
Add your package and Groovy packages to Export-Package
Add Groovy-all.jar to Bundle-ClassPath attribute in MANIFEST.MF
Not best way to include dependencies, but an easy one!
You need to export Groovy packages because they are part of the public API of any classes you defined.

Creating a Usable, Modular Bundle

Build JAR with code and Groovy-all.jar within

Add Exports *and Imports* to MANIFEST.MF

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 10.0-b19 (Sun Microsystems Inc.)
Bundle-SymbolicName: org.example.UsableModularBundle
Export-Package: org.example;version="1.0.0",groovy.lang
;version="1.6.4"
Import-Package: groovy.lang;version="1.6.4",org.codehau
s.groovy.runtime;version="1.6.4",org.codehaus.groovy.runtim
e.callsite;version="1.6.4",org.codehaus.groovy.reflection;vers
ion="1.6.4"
```

Build a Jar containing code and Groovy-all.jar within
Add Bundle-SymbolicName attribute to MANIFEST.MF
Add your package and Groovy packages to Export-Package
Add groovy.lang packages to Import-Package
Install Groovy-all.jar into your container
org.example.Greeter is now published!

Consume Your New Bundle!

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 10.0-b19 (Sun Microsystems Inc.)
Bundle-SymbolicName: org.example.UsableModularBundle
Export-Package: org.example;version="1.0.0",groovy.lang
;version="1.6.4"
Import-Package: groovy.lang;version="1.6.4",org.codehau
s.groovy.runtime;version="1.6.4",org.codehaus.groovy.runtim
e.callsite;version="1.6.4",org.codehaus.groovy.reflection;vers
ion="1.6.4"
```

Versioning

Meaning not *marketing*

Major – breaking Change

Minor – backward compatible change

Micro – bug fix

Qualifier – build id

No more letting your marketing department dictate version numbers

A system without unlimited flexibility was chosen to limit complexity (not so in Jigsaw today).

Bundle Identity = Symbolic Name + Version

Both bundles and packages can have version numbers

Versioning

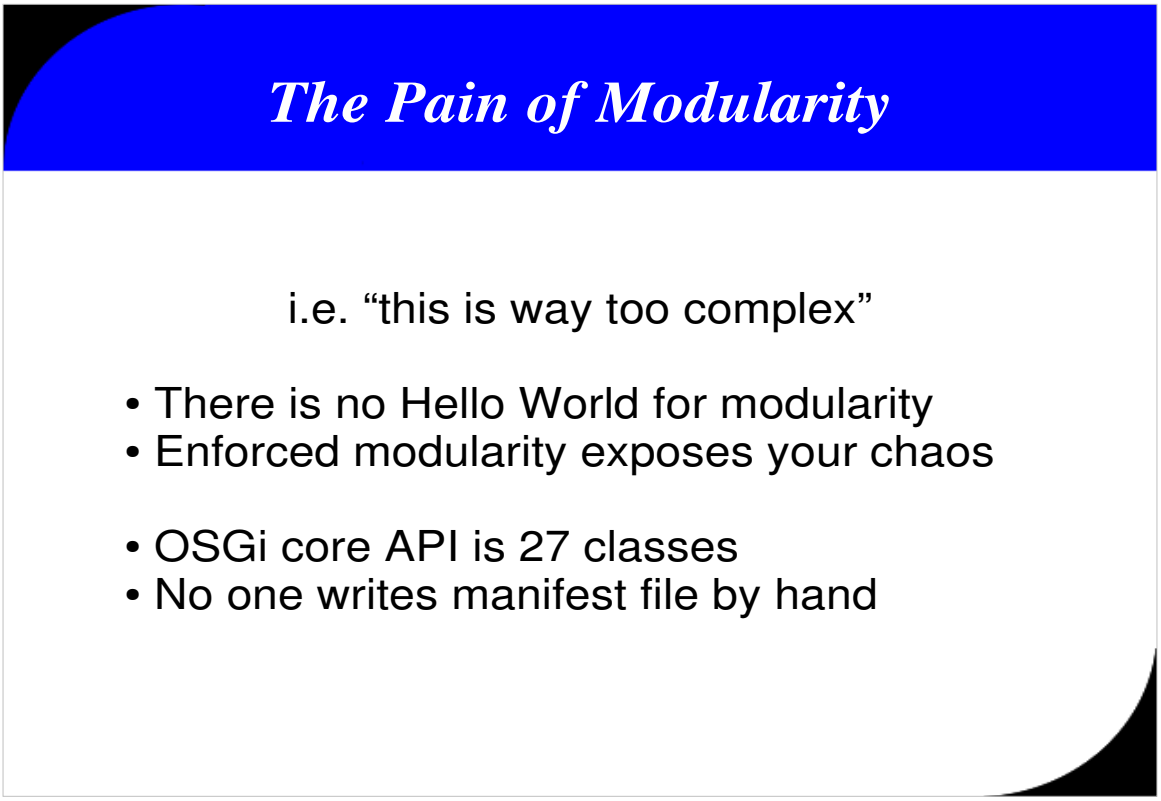
org.example;version="1.6.0.beta-2"

org.example;version="[1.0,2.0)"

org.example;version="[1.0,1.9]"

Bracket is inclusive, paren exclusive

Minor will show up in minimum version to ensure an API exists



You cannot demonstrate modularity with a Hello World because modularity solves the problem of large evolving code bases. Peter Kriens

Kriens - enforced modularity is painful because it confronts you with all the entanglements in your code, and even worse, the hacks and shortcuts in the libraries you use. Strong modularity puts your code in a straightjacket, and often that is no fun when you have legacy code that enjoys the anarchy of the Java class path.

Kriens - OSGi core API is 27 classes. That is a all. Security, Module layer, Life cycle layer, and Service Layer. Exceptions, permissions, and interfaces. And one of them is even deprecated!

Kriens - About the manifest, well, it is years ago I wrote a manifest. Tools like bnd make this file disappear and take the drudgery out of writing the manifest by calculating many of its values. Complaining about the manifest is like complaining about writing the class file format, you should never have to do that.

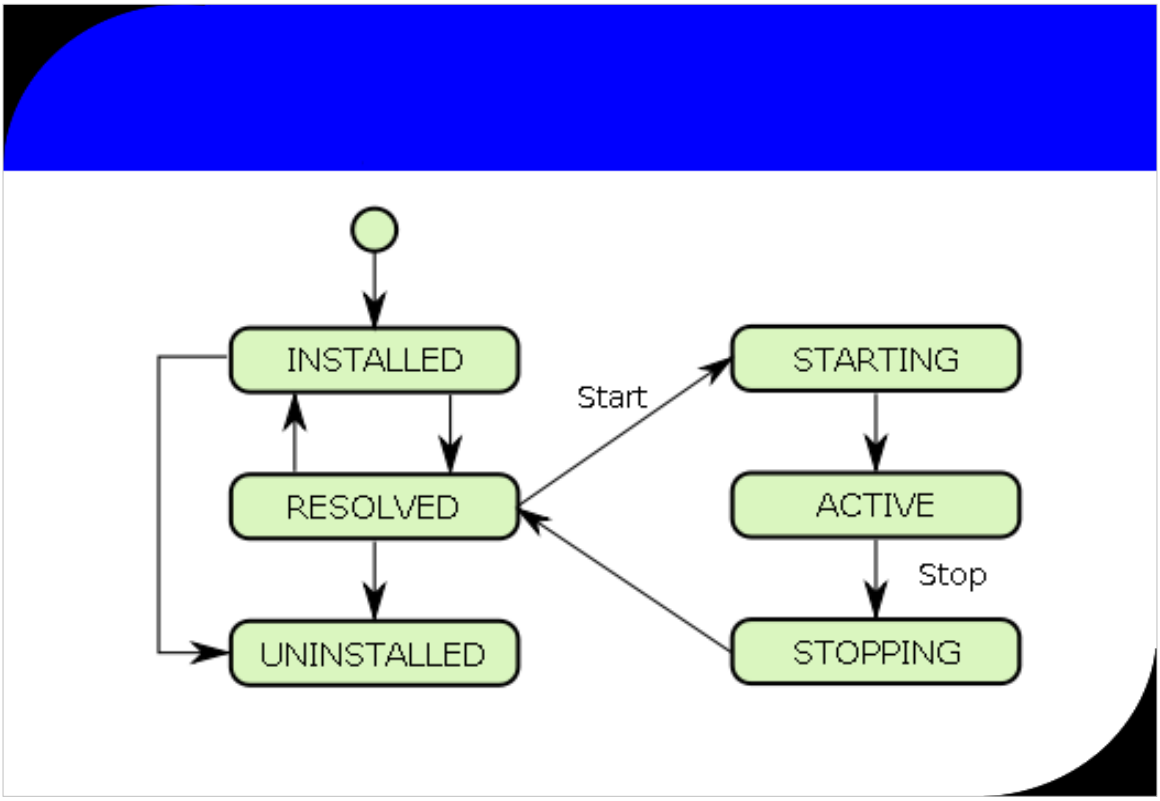
Service Lifecycle

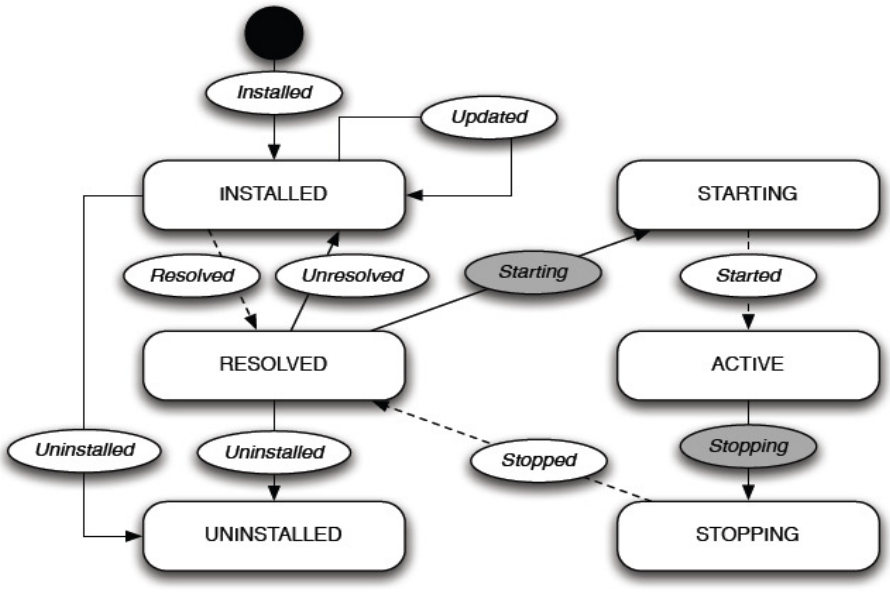
The Lifecycle

1) Install
2) Start
3) Stop
4) Uninstall

… what, you wanted a diagram?

Installed and started are important. Installed means that the bundle is available in the container. When a bundle moves to started is when dependencies are resolved. If there isn't a valid version of all dependencies installed then the start fails. There is no runtime failure of missing classes or question of which jars are in use.


INSTALLED
RESOLVED
UNINSTALLED
Start
STARTING
ACTIVE
Stop
STOPPING



Ah, that's better. (from OSGi in Practice)

Static vs. Dynamic Services

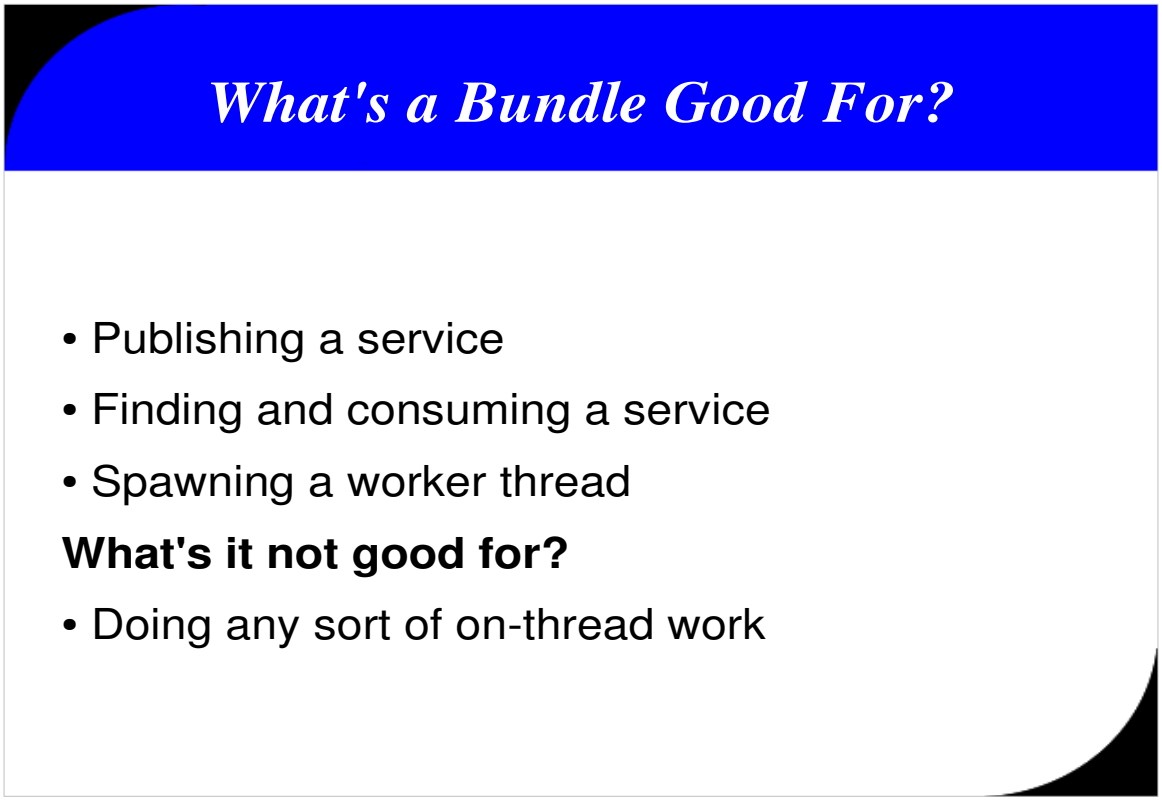
A static service:

```
public static void main(String[] args)
```

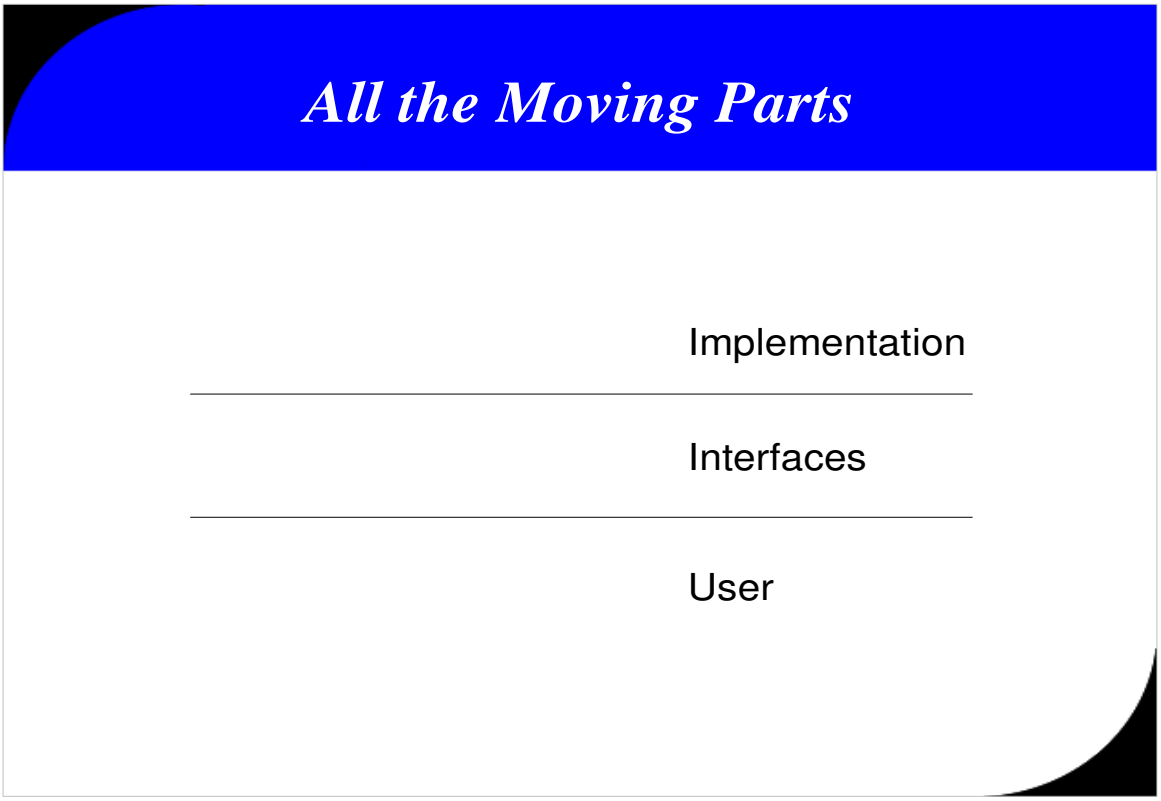
A dynamic service:

```
interface BundleActivator {
    public void start(BundleContext);
    public void stop(BundleContext);
}
```

Dynamic Modules – Bundles can be installed, updated, and uninstalled without taking down application. Services are just published and listened for, that's all.

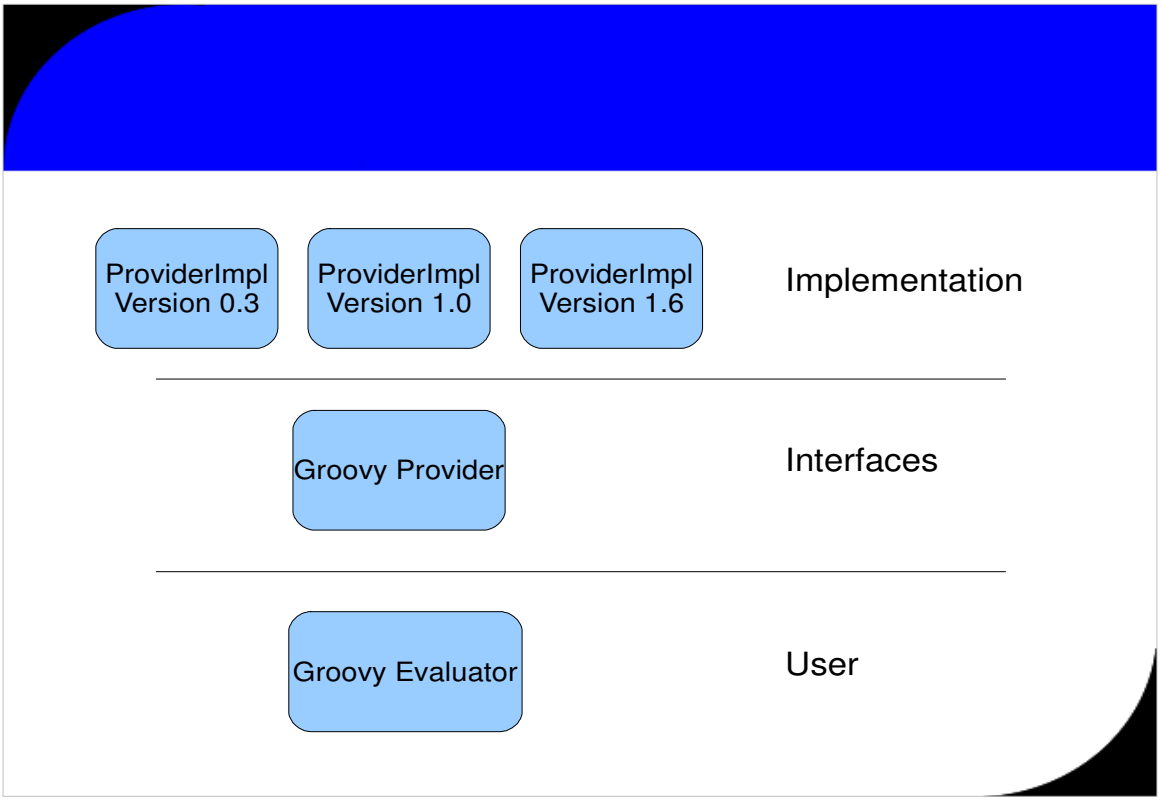


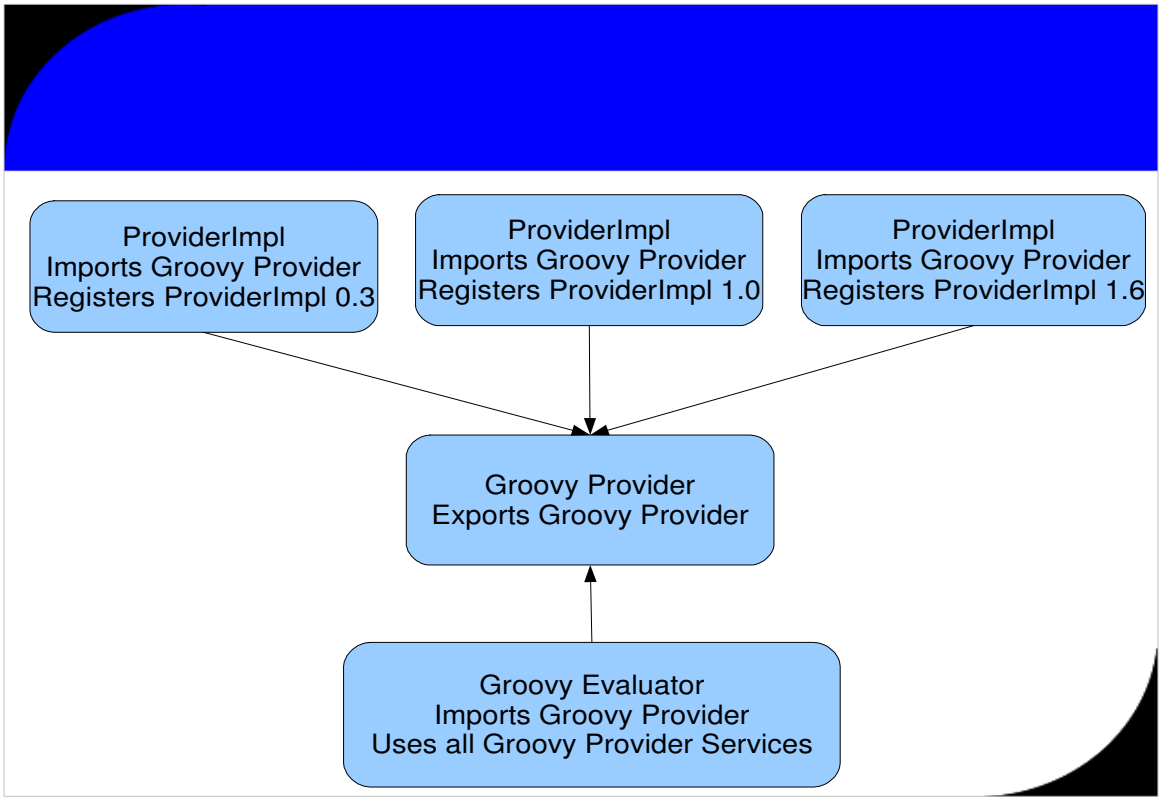
Making a bundle is easy. Publishing a service is easy. Consuming a service is hard. You will want to read up on the concurrency issues



Services need 3 bundles: the interfaces, the implementation, the user of the service

(These samples are from the Groovy sample OSGi project)


ProviderImpl Version 0.3
ProviderImpl Version 1.0
ProviderImpl Version 1.6
Implementation
Groovy Provider
Interfaces
Groovy Evaluator
User


ProviderImpl
Imports Groovy Provider
Registers ProviderImpl 0.3
ProviderImpl
Imports Groovy Provider
Registers ProviderImpl 1.0
ProviderImpl
Imports Groovy Provider
Registers ProviderImpl 1.6
Groovy Provider
Exports Groovy Provider
Groovy Evaluator
Imports Groovy Provider
Uses all Groovy Provider Services

Publishing a Java Service

```
import org.osgi.framework.*;

class Activator implements BundleActivator {
  private ServiceRegistration reg;

  void start(BundleContext ctx) {
    MyService service = new MyService();
    String name = MyService.class.getName();
    reg = ctx.registerService(name, service, null);
  }

  void stop(BundleContext ctx) {
    reg.unregister();
  }
}
```

Start is called when your bundle is “started”. This occurs only after the bundle is installed. Stop is called when the bundle is stopped. It is important to clean up properly after yourself. This ensures that all events are all published on the event bus properly and others can clean up as well.

Note: a service is registered by its type: class.getName()

Publishing a Groovy Service

```
import org.osgi.framework.*

class Activator implements BundleActivator {

  ServiceRegistration registration

  void start(BundleContext context) {
    ClassLoader original = Thread.currentThread().contextClassLoader
    try {
      def newLoader = getClass().classLoader
      Thread.currentThread().contextClassLoader = newLoader
      def service = new MyService()
      def name = MyService.class.getName()
      registration = context.registerService(name, service, null)
    } finally {
      Thread.currentThread().contextClassLoader = original
    }
  }
  void stop(BundleContext context) {
    registration.unregister()
  }
}
```

I used to be able to explain why you need this but I've forgotten. I think it's a coping mechanism.

Consuming a Groovy Service

- Groovy services can be consumed from Java bundles with no changes

But

- Program to interfaces
- Keep Groovy out of interface type signatures

Groovy should be an implementation detail. Putting getMetaClass() in a service class forces the consumer to import the Groovy bundle.

Finding Installed Services

```
class MyActivator implements BundleActivator {

  void start(BundleContext ctx) {
    def name = MyService.class.name
    def refs = ctx.getAllServiceReferences(name, null)

    refs?.each { ServiceReference ref ->
      MyService service = ctx.getService(ref)
      service.doSomething()
    }
  }

  void stop(BundleContext context) { }
}
```

Service Properties

```
def refs = context.getAllServiceReferences(
            name,
            ['route'   : 'Kessel Run',
            'distance': '12-parsecs'])
```

Service Filtering

```
def filter =
      '(&(route=Kessel Run) (distance=12-parsecs))'


def refs = context.getServiceReferences(
            name,
            filter)
```

Service Filtering with filter4osgi

```
import static org.filter4osgi.builder.FilterBuilder.*

...

def filter = and(
              eq('route', 'Kessel Run'),
              eq('distance', '12-parsecs')
             ).toString()

def refs = context.getServiceReferences(name, filter)
```

filter4osgi bunus: paid bug reports!

Bonus of filter4osgi: bug reports are paid!

(Disclaimer: it is my project)

Challenges of Locating Services

Examples assumed static services, not dynamic

How to code a system where

… bundles come and go at runtime?

… bundle start order is unspecified?

"Services have to discover each other and cannot depend on one being installed and started before the other". In Real Life™, services follow a blackboard pattern: one entity registers services in a shared location and another entity checks for the existence of services in that location.

http://osgi.mjahn.net/2009/07/01/osgi-vs-jigsaw-why-cant-we-talk/ :

OSGi provides a pretty strict module system, which when running is well defined. Unfortunately, there is a problem with how to get to this state or even knowing when this state is achieved. The idea behind its model is that using services, there should not be any dependency on the start order, because everything can change at any time. This is a nice idea, but in real world it is impossible to achieve.

It is just not standardized how to ensure that certain bundles have to be loaded before everything else in order to start a particular runtime with security on. OSGi limits itself to define the runtime behavior, leaving out configuration issues when moving from one container to another. Basically the configuration for the start levels of each bundle getting loaded is an implementation detail - doesn't this look familiar to you when using JEE ;-) "eventually we got different class loading solutions for each J2EE vendor, because this part of the specification was left out."

ServiceTracker

- Well documented
- Implement ServiceTrackerCustomizer or subclass ServiceTracker
- Notifies of past and future services coming online

Show example from demo: groovy-evaluator - Activator.java

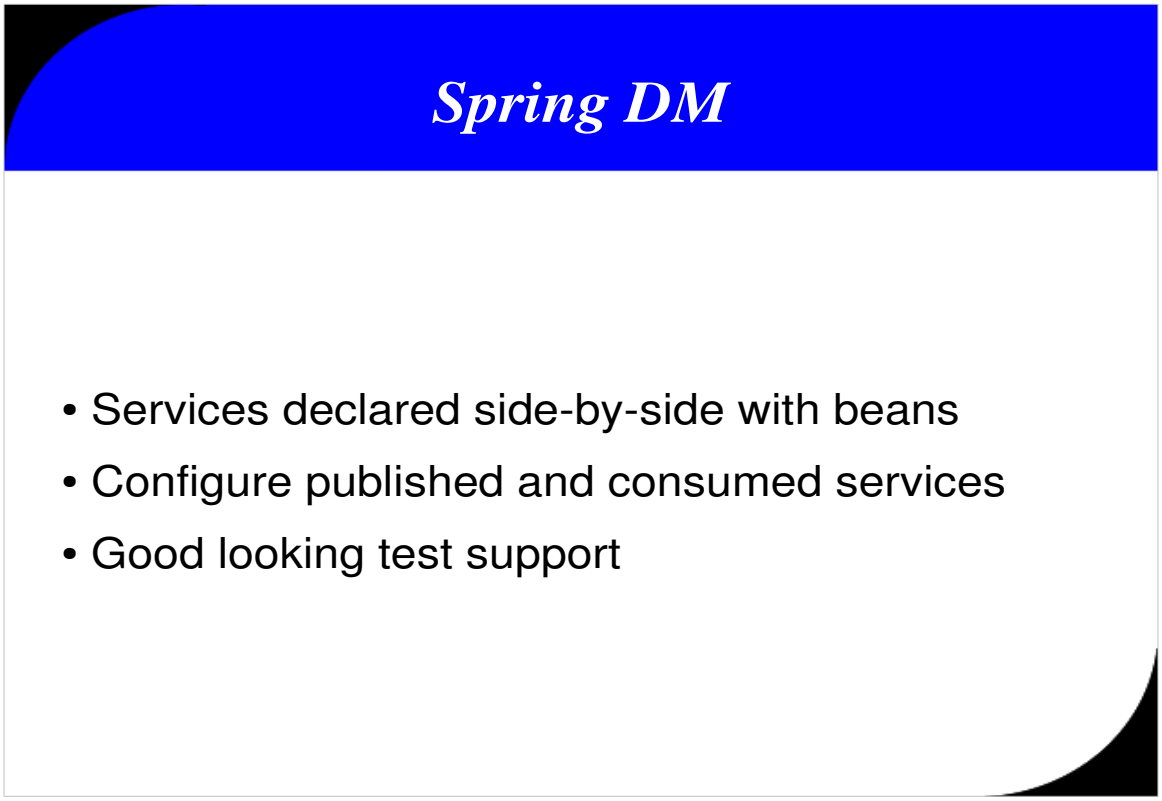
Declarative Services

ServiceTracker requires:

- boilerplate code
- cautious approach to threading

DS offers better solution:

- Service declared in XML in JAR OSGI-INF folder
- Consumers specify a “cardinality” for service



Spring DM is a form of declarative services for OSGi

Disclaimer: I'm an official Spring Fan Boy (not sure about VMWare though)

Good looking testing support with `AbstractConfigurableBundleCreatorTests`

Testing support looks great: can pick which containers to test on, specify bundle dependencies, and specify custom JAR manifests.

Tools

OSGi Implementations

Equinox – Widely used, open source, many online examples

Knopflerfish – Good testing support, open source

Felix – Apache Group's compact, embeddable release

Concierge – very small, very optimized. Meant for handsets

Writing Manifest Files

bnd – App for command line, Eclipse, Maven, and Ant

Gradle OSGi plugin – Wraps bnd

Bundlor – SpringSource's bnd alternative

Eclipse PDE – Plugin Development Environment

Manifest First (PDE) - tools to write manifest files (will mimic what happens at runtime)

Template Driven (BND) - tools to generate manifest files

IDE Shootout: IntelliJ IDEA

- Available in IDEA 9 or as Osmorc Plugin in 8
- Generates bundles and helps with manifests
- Launch, deploy, debug within containers
- Runs testing tools like Pax Exam

IDE Shootout: Eclipse PDE

Same as IDEA but adds...

- Detects versioning requirements, compatibility issues, breaking API changes, API changes after version freeze, and API leaks
- Suggests versions, validates @since tags, performs usage scans
- Supports @noimplement, @noextend, @noinstantiate, @nooverride, & @noreference
- Enforces execution environments
- ... tons of online examples

Slide left intentionally busy.

Claims to have better i18n tooling and better support for declarative services. I haven't figured out what this means yet.

Testing

- Just run JUnit outside OSGi container!
- Or run JUnit tests inside a container
- junit4osgi – adds OSGiTestCase
- Pax Exam and Spring DS Support

Junit works fine outside the container.

You can run Junit within a container by using Junit bundle from Knopflerfish

Apache Felix junit4osgi – mixes the two

Pax Exam & DS classes seem to have similar intent

Pax Exam

- Control of loading dependent bundles
- Control for executing on many/all containers
- Nice IDEA support

Demo the filter4osgi tests
Builds off Pax Runner, which is a tool you can use to launch a suite of bundles as an application.

Project Jigsaw

- Whatever happened to “Super Packages”?
- Jigsaw is a module system plus more
- JSR 294 - language/VM changes for modules
- Modularizes JDK, but we can use it
- Not part of Java SE 7 Platform Specification

Super Packages grew to become something very large
“Plus more” refers to native packaging system.
Jigsaw requires JSR 294, language changes to support modularity.
Buckley: “JSR 294 does not depend on Jigsaw. Rather, Jigsaw and OSGi are likely to exploit language/VM features defined by JSR 294.”
Confusing: “the OpenJDK project called Jigsaw is both the Reference Implementation of JSR 294 and the design+implementation of the Jigsaw module system.”

Striving for “No Black Holes” in simple module system.

The Jigsaw "Controversy"

- Module data in MANIFEST.MF or Java source?
- What exactly are the requirements?
- Are there "black holes"?
- Can Jigsaw do it better?

"Controversy" = most parties are downplaying the differences and trying to get along. But these were criticized JSRs and expert group even before Jigsaw dropped in December 2008.

Buckley: If you believe that the ultimate component is a service, then a bundle is a dull unit of packaging whose production should be automated with tools like bnd." "If you believe that the ultimate component is a logical module with an exported API on which other modules should depend, then explicit declarations of module relationships, written by a programmer, are natural" Sounds like a build time vs. runtime debate.

Requirements: Partial packages (splitting java.lang) is a requirement. Timing might be a requirement. An alliance probably can't gain consensus in time for JDK7 Release. Opponents say that splitting packages on performance effects is almost always a bad idea. Apache Harmony has not demonstrated that OSGi can be used to modularize the JDK (but I don't remember details, it's in Java Posse #259)

Kriens "This requirement is begging for more complexity and the payoff seems very slim. Splitting packages along performance boundaries will require great foresight to have any performance effects and will in almost all cases be in conflict with minimizing coupling. It is a classic example of coupling of two unrelated concepts (performance, low coupling) into a single design concept (module). In practice, it always results in systems that are not good in either performance nor in decreasing the coupling. Though performance is a crucial aspect of the VM (and some fantastic work has been done in Hotspot, even without modularity), it is important not to mix concepts that have very different optimization axes. Every time when I see that happening, both axes have to be compromised."

A benefit of Jigsaw: native packaging (which OSGi isn't trying to solve!). Another: Jigsaw offers more control over visibility. In OSGi, Bundle A looks up a service from Bundle B. Bundle A cannot see class of service implementation but can call service.getClass() and get a reference to public fields/methods in Bundle B and then invoke them. Jigsaws module keyword will be able to prevent this. Gives bundle authors more control and encapsulation.

Conclusion

- OSGi isn't hard. Modularity is hard
- OSGi exists today (several implementations!)
- OSGi works for for Java 1.3 and greater
- Is modularity the only answer to complexity?

We did not talk about security or execution environments.

Bartlett on 747s: “The only approach that will allow human beings to design such incredible machines is to break them down into smaller, more understandable modules.

Gabriel on ULS: “ULS systems will be designed beyond human comprehension by design methods that we don't completely understand."

Additional Resources

Java Posse #245 - OSGi Interview (Kriens & Hargrave)
Java Posse #259 - Jigsaw and JSR 294 Interview (Reinhold & Buckley)
OSGi in Practice – Neil Bartlett's free-as-in-vivre Creative Commons book
EclipseSource Webinars - http://eclipsesource.com/en/about/events/webinar-osgi-and-equinox-jumpstart/
Peter Krien's Blog - http://www.osgi.org/blog/
Modular Java - Craig Walls, Prag Press
Equinox and OSGi: The Power Behind Eclipse (Safari Rough Cuts)
OSGi in Practice – Stuart McCulloch, Manning
Pro Spring Dynamic Modules for OSGi – Daniel Rubio, Apress
Neil Bartlett – OSGi for Application Developers - http://www.infoq.com/presentations/OSGi-for-Application-Developers-Neil-Bartlett
Getting Started with OSGi – Neil Bartlett http://neilbartlett.name/blog/osgi-articles/
Getting Started with Spring DM – Dzone Refcard by Craig Walls