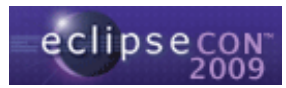
Component Oriented Development in OSGi with Declarative Services, Spring Dynamic Modules and Apache iPOJO

Neil Bartlett[1] Heiko Seeberger[2]

[1]Weigle Wilczek UK

[2]Weigle Wilczek GmbH

March 24, 2009



Part I

Introduction

Contents

The (Partial) Failure of Object Oriented

- One of the primary goals of object oriented programming (OOP) was, and still is, **re-use**.
- It has **mostly failed** in that goal.

The (Partial) Failure of Object Oriented

- One of the primary goals of object oriented programming (OOP) was, and still is, **re-use**.
- It has **mostly failed** in that goal.

The (Partial) Failure of Object Oriented

Peter Kriens

"Object Oriented technology was going to change the world... we would have all these objects in our library and building a new system would be a snap. Just get a few classes, bunch them together... and *voila*!"

What Went Wrong??

- *Coupling*
- Classes can almost never be used in isolation – they depend on other classes.
- Those classes depend on other packages, which depend on other JARs. . .

What Went Wrong??

- *Coupling*
- Classes can almost never be used in isolation – they depend on other classes.
- Those classes depend on other packages, which depend on other JARs. . .

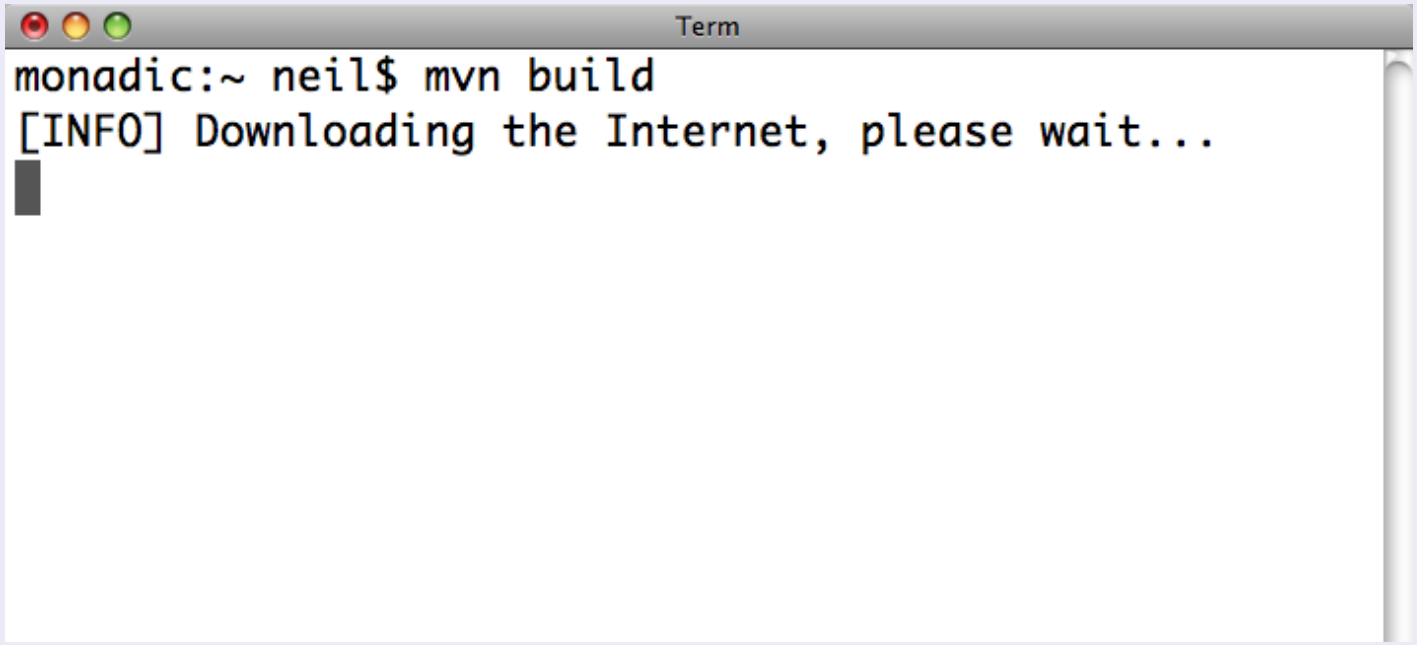
What Went Wrong??

- *Coupling*
- Classes can almost never be used in isolation – they depend on other classes.
- Those classes depend on other packages, which depend on other JARs. . .

What Went Wrong??

Eventually just to use one small class we end up doing this:

Maven



A Solution?

- Re-use of classes outside their original context is hard, so...
- Give up!
- Leave them where they are, and call over the network!
- This is sometimes called "SOA".

A Better Solution

- *Component Oriented Programming*
- Builds on OOP.
- OOP is not *bad*, it's just not the whole answer.

Contents

Components

But wait, what *is* a "component", vs an object?

- Good question! Many attempts have been made to define the term.
- The following is not a formal academic definition, just a working definition that we have found useful in practice.

Components

But wait, what *is* a "component", vs an object?

- Good question! Many attempts have been made to define the term.
- The following is not a formal academic definition, just a working definition that we have found useful in practice.

The Most Common Analogy

Lego?

Lego is a Poor Analogy

- Just dead lumps of plastic.
- All look and act the same.
- Not even very re-usable (try using a Duplo block in a Lego Technics model!).

A Better Analogy

Bees/Animals

Components Are:. . .

- *Active* participants in the system.
- *Aware* of and *adapt* to their environment.
- May provide services to other components and use services from other components.
- Have a life cycle.

Components Are:. . .

- *Active* participants in the system.
- *Aware* of and *adapt* to their environment.
- May provide services to other components and use services from other components.
- Have a life cycle.

Components Are:. . .

- *Active* participants in the system.
- *Aware* of and *adapt* to their environment.
- May provide services to other components and use services from other components.
- Have a life cycle.

Components Are:. . .

- *Active* participants in the system.
- *Aware* of and *adapt* to their environment.
- May provide services to other components and use services from other components.
- Have a life cycle.

What Does the "Environment" Mean?

- Services provided by other components.
- Resource, devices, etc.

Adaptation

- When the environment is good, the component *flourishes*.
- When the environment is harsh, the component *survives*.
- When very harsh, the component *sleeps or dies*.

Contents

Implementing Components

- OSGi is the *perfect* environment for implementing components.
- The module layer allow us to minimise our static dependencies.
- Fewer static dependencies means less stuff that *must* be present for our component to work.
- Services allow our component to interact with other components.

Implementing Components

POJOs

Components should be implemented as POJOs (Plain Old Java Objects) "glued" together with OSGi services.

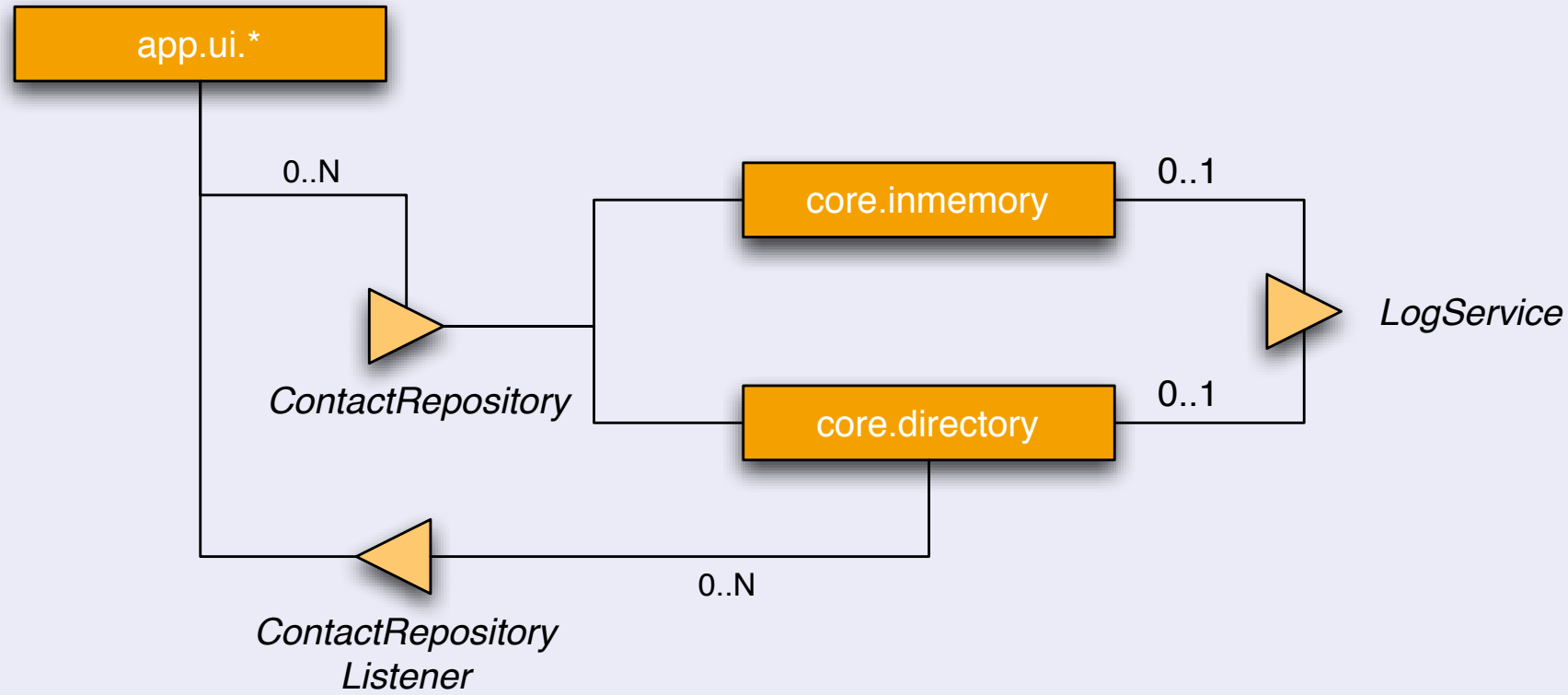
Contents

The Example Application

- The example is a simplistic “contact manager” application.
- Capable of displaying the content of multiple contact “repositories”.
- Each repository appears in its own tab.
- Repositories are implemented as OSGi services.

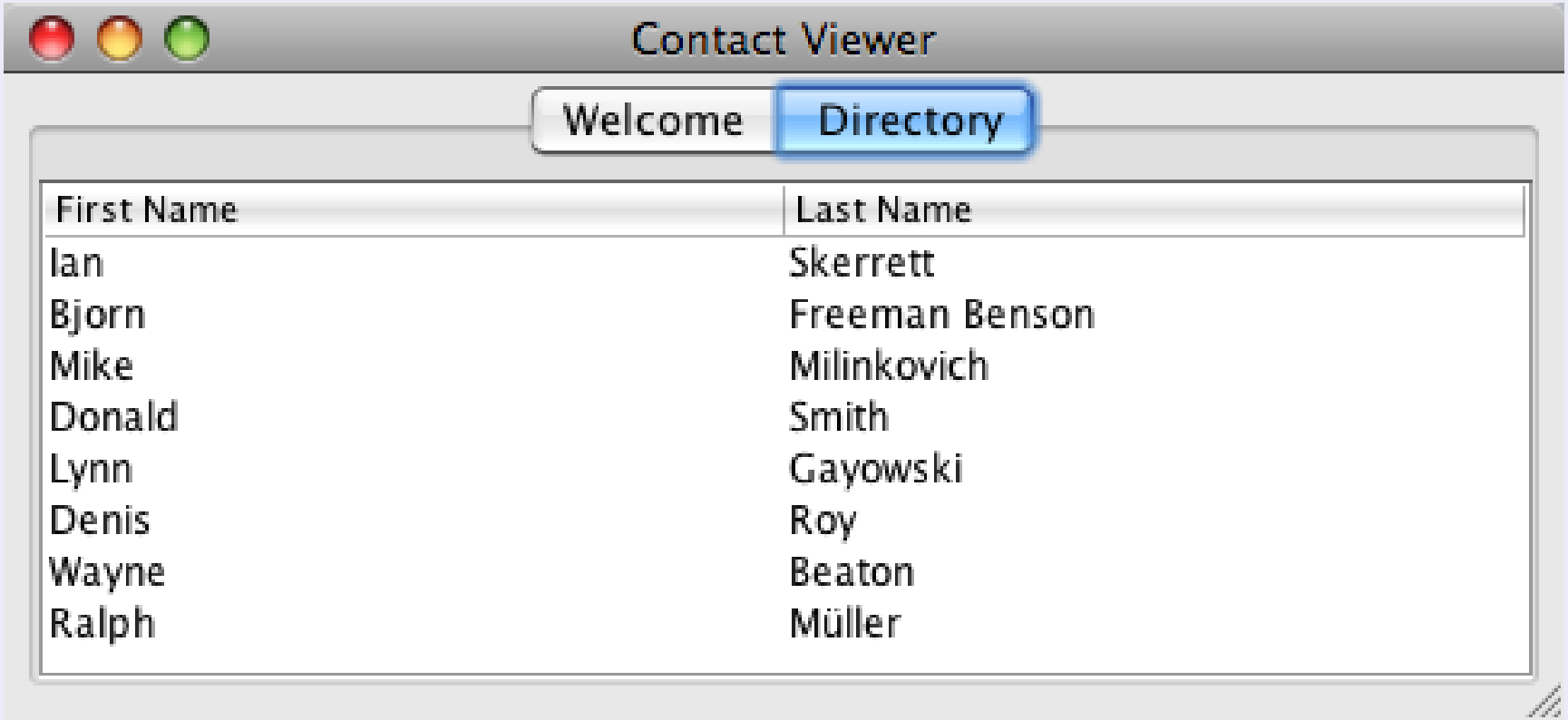
Example Architecture

Services Diagram



The Example Application

Contact Repository Viewer



Running the Example Application

First start Equinox using the "Core" launcher from Eclipse's Run menu.

Equinox Console

```
osgi> install file:contacts_api.jar
Bundle id is 3

osgi> install file:swingui.jar
Bundle id is 4

osgi> start 4
```

The UI should now appear, but with no contact repository tabs.

Installing a Basic Contact Repository

Equinox Console

```
osgi> install file:../ComponentsRaw/contacts.core.inmemory_raw.jar
Bundle id is 5

osgi> start 5
```

You should see two repositories: “Some Dummies” and “OSGi Nerds”.

The "Raw" Components

- The `ComponentsRaw` project contains two repository implementations.
- They are implemented using "raw" OSGi APIs, i.e. `ServiceTracker`.

Implementing Components with Raw OSGi

- It *is* possible to implement components using just the raw OSGi APIs. But there are several challenges.
- We need to be cautious to separate component code from OSGi "glue". If OSGi APIs are used in our components it makes them difficult to test and non-POJO.
- It is much harder to access services and configuration using raw APIs.
- Small changes in desired behaviour – e.g. a switch from mandatory to optional dependence on a service – require large changes in our glue code.
- Much of the glue code is repetitive – mostly the same for many different components.

Component Frameworks

- We would like to use a framework to ease implementation of components in OSGi.
- There is more than one to choose from!
- In this tutorial we look at three popular choices: Declarative Services; Spring Dynamic Modules; and Apache iPOJO.

A More Interesting Component

For most of this tutorial we will focus on a more interesting and complex example.

- The `DirectoryRepository` component implements a contact repository backed by files in a directory.
- It is *active*: it creates a thread to monitor the directory.
- It needs configuration: the directory to monitor.
- It consumes services provided by other components: the log service and some listeners.
- It provides a service to other components: the contact repository functionality.

Getting Stuff

- **Exercises**:
  http://neilbartlett.name/downloads/eclipsecon2009/labs.zip
- **Solutions**:
  http://neilbartlett.name/downloads/eclipsecon2009/solutions.zip
- **These Slides**:
  http://neilbartlett.name/downloads/eclipsecon2009/slides.pdf

NB these are temporary URLs until the files have been uploaded to `eclipsecon.org`.

Part II

Declarative Services

Contents

Declarative Services

"Declarative Services" (DS) is a specification from the OSGi Compendium, section 112. It was introduced in Release 4.0 and is based on the extender model.

- Like all extenders, DS performs tasks *on behalf of* other bundles.
- The DS spec defines this extender and it is implemented by frameworks. The extender bundle itself is called the "Service Component Runtime" or SCR.

The terms DS and SCR are sometimes confused. Remember, DS is the specification, SCR is the actual bundle that implements the specification.

Declarative Services

- There are *significant* improvements in DS in OSGi R4.2.
- Many of these changes are supported in Equinox 3.5M5+.
- We will use the R4.2 features, but mention when we do so.

Declarative Services

What does the SCR extender bundle do on our behalf?

1. Creates Components.
2. "Binds" them to services and configuration.
3. Manages the component's lifecycle in response to bound services coming and going.
4. Optionally, publishes our components as services themselves.

Declarative Services

What does the SCR extender bundle do on our behalf?

1. Creates Components.
2. "Binds" them to services and configuration.
3. Manages the component's lifecycle in response to bound services coming and going.
4. Optionally, publishes our components as services themselves.

Declarative Services

What does the SCR extender bundle do on our behalf?

1. Creates Components.
2. "Binds" them to services and configuration.
3. Manages the component's lifecycle in response to bound services coming and going.
4. Optionally, publishes our components as services themselves.

Declarative Services

What does the SCR extender bundle do on our behalf?

1. Creates Components.
2. "Binds" them to services and configuration.
3. Manages the component's lifecycle in response to bound services coming and going.
4. Optionally, publishes our components as services themselves.

Declarative Services

What does the SCR extender bundle do on our behalf?

1. Creates Components.
2. "Binds" them to services and configuration.
3. Manages the component's lifecycle in response to bound services coming and going.
4. Optionally, publishes our components as services themselves.

Contents

A Minimal Component Declaration

minimal.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">

  <implementation class="org.example.osgi.ds.HelloComponent"/>

</scr:component>
```

NB: the namespace is required in order to use R4.2 features. If not included then SCR will default to the prior version.

A Minimal Component Declaration

- This declaration requests SCR to simply instantiate the named class, which must be visible to the declaring bundle's classloader.
- We need to tell the SCR extender bundle about this file.
- We use a manifest header, `Service-Component`.
- A comma-separated list of XML declaration files.

A Minimal Component Declaration

MANIFEST.MF

```
Bundle-SymbolicName: mybundle
Bundle-Version: 1.0.0
Service-Component: OSGI-INF/minimal.xml
...
```

A Minimal Component Declaration

HelloComponent

```
package org.example.osgi.ds;

public class HelloComponent {
    public HelloComponent() {
        System.out.println("HelloComponent created");
    }

    // ...
}
```

Note: a Plain Old Java Object (POJO)! No OSGi API dependencies.

Building the Bundle

Internal Bundle Structure

```
minimal_ds.jar/
    META-INF/
        MANIFEST.MF
    OSGI-INF/
        minimal.xml
    org/
        example/
            osgi/
                ds/
                    HelloComponent.class
```

Running the Example

We need to install the SCR bundle from Equinox, and also a utility bundle used by Equinox's SCR:

- `org.eclipse.equinox.ds_<version>.jar`
- `org.eclipse.equinox.util_<version>.jar`

The SCR bundle needs to be started, but the `util` bundle does not.

Running the Example

Console

```
osgi> install file:org.eclipse.equinox.ds_1.1.0.v20090112-1800.jar
Bundle id is 2

osgi> install file:org.eclipse.equinox.util_1.0.1.v20081205-180...
Bundle id is 3

osgi> start 2

osgi> ss

Framework is launched.

id  State       Bundle
0   ACTIVE      org.eclipse.osgi_3.4.2.R34x_v20080826-1230
1   RESOLVED    org.eclipse.osgi.services_3.2.0.v20081205-1800
2   ACTIVE      org.eclipse.equinox.ds_1.1.0.v20090112-1800
3   RESOLVED    org.eclipse.equinox.util_1.0.1.v20081205-1800

osgi>
```

Running the Example

Now install and start the example bundle:

Console

```
osgi> install file:minimal_ds.jar
Bundle id is 4

osgi> start 4
HelloComponent created

osgi>
```

Note that SCR ignores our bundle until it is in ACTIVE state. DS bundles must be activated even though they often have no bundle activator! This is quite different from Eclipse Extensions.

Contents

Activation and Deactivation

- That was a very long-winded way to merely instantiate a class!
- This component is not useful because it cannot even *do* anything.
- However, DS allows us to define lifecycle methods.
- We can be notified when the component starts and stops.
- This allows us to do interesting things like start threads, open sockets, etc.

An Active Component

PollingComponent

```
public class PollingComponent {

    private static final int DEFAULT_PERIOD = 2000;
    private PollingThread thread;

    protected void activate(Map<String,Object> config) {
        System.out.println("Polling Component Activated");
        Integer period = (Integer) config.get("period");
        thread = new PollingThread(
                period != null ? period : DEFAULT_PERIOD);
        thread.start();
    }
    protected void deactivate() {
        System.out.println("Polling Component Deactivated");
        thread.interrupt();
    }
}
```

Activation and Deactivation

- SCR will find our activate/deactivate methods automatically.
- We can call them something else by adding attributes to the XML declaration.

NB: Before R4.2, the method names could not be changed and they had to take a parameter of type `ComponentContext`, from the DS API. This broke the pojocity of the component.

Activation and Deactivation

Why didn't we just write this as a bundle activator??

- This is a POJO!
- Easier access to configuration – the `Map` parameter to `activate`.
- Easier access to service registry – we will see this soon.

Contents

References to Services

- Using lower level APIs we must write a lot of "glue" code to bind to services.
- DS replaces the glue code with simple declarations.
- We declare *references* to services.

References to Services

Example reference Element

```
<reference name="LOG"
    interface="org.osgi.service.log.LogService"
    bind="setLog" unbind="unsetLog"/>
```

Reference Attributes

name	The name of the reference.
interface	The service interface name.
bind	The name of the "set" method associated with the reference.
unbind	The name of the "unset" method associated with the reference.

Contents

Optional vs Mandatory References

- We previously discussed the idea of *optional* and *mandatory* service dependencies.
- To model these with `ServiceTracker` required completely different code patterns.
- With DS we can simply change a single attribute on the `reference` element to switch between optional and mandatory.

Optional Service Reference

Example `reference` Element

```
<reference name="LOG"
    interface="org.osgi.service.log.LogService"
    bind="setLog" unbind="unsetLog"
    cardinality="0..1"/>
```

- The default cardinality was "`1..1`" meaning that we must have exactly one instance, i.e. the reference is mandatory.
- A cardinality of "`0..1`" indicates that either zero or one instance is okay, i.e. the reference is optional.
- Yes, "`0..n`" and "`1..n`" do exist. We will look at these later.

Lab 1 : Building the Directory Repository (Setup)

1. Install and start the `commands.jar` bundle which gives us a useful `installDir` command.

```
osgi> install file:commands.jar
Bundle id is 3

osgi> start 3
```

2. Install the Contacts API bundle, and the DS bundles:

```
osgi> install file:contacts_api.jar
Bundle id is 4

osgi> installDir ../ComponentsDS/bundles
```

Lab 1 : Building the Directory Repository

1. Complete the implementation of the `DirectoryRepositoryDS` class – TODO markers explain what is needed.
2. Complete the DS XML descriptor `directoryRepository.xml` in the `dsxml` directory. Remember to add at least:
   - A `property` element to set the `dirName` property to the name of the `contacts` directory.
   - An optional reference to the OSGi log service (`org.osgi.service.log.LogService`).
   - A zero-to-many reference to the repository listeners.

Lab 1 : Testing the Component

1. Build the bundle using the supplied ANT build.
2. Install and start the bundle and also start the SCR extender bundle (`org.eclipse.equinox.ds...`).
3. Type the `log` command to see messages from the component. Observe what happens when `.contact` files are added/removed in the `contacts` directory.

Contents

Static vs Dynamic Components

If we stop the log bundle (`org.eclipse.equinox.log...`) we should see something like this:

```
osgi> stop 6
DirectoryRepositoryDS constructed
```

Static vs Dynamic Components

. . . and when restarting the log bundle :

```
osgi> start 6
DirectoryRepositoryDS constructed
```

It seems our service is being destroyed and recreated each time the referent goes away or comes back. Why??

Static Policy

- By default, DS uses a *"static policy"* for updating references to services.
- This means it hides the complexity of dealing with dynamically changing references. We need not worry about service references being replaced while we are using them or using `synchronized`.
- It does this simply by destroying and recreating the component instance each time the references change.

Efficiency of the Static Policy

- *Much of the time*, this isn't too expensive, and it's a reasonable trade off.

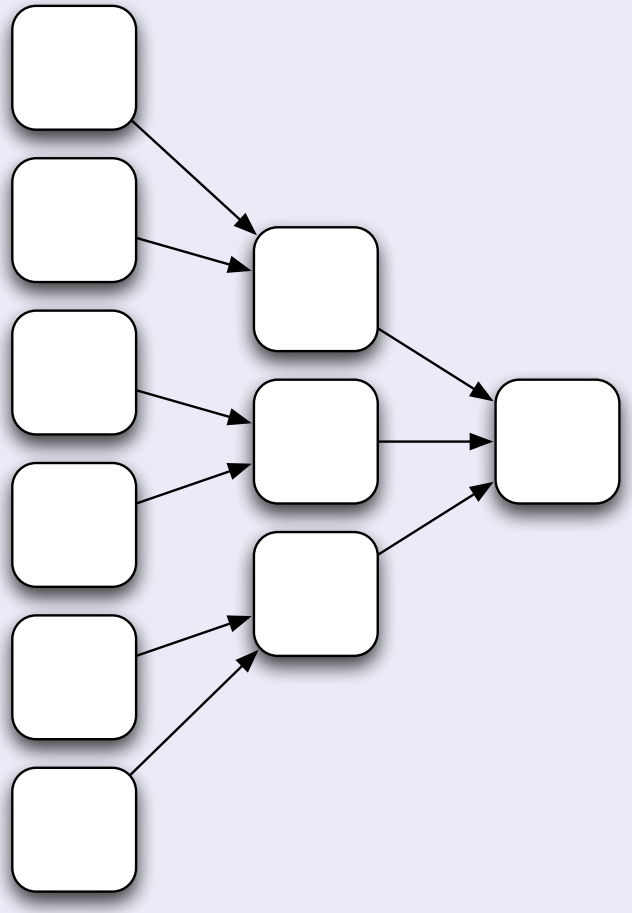
Donald Knuth

"Premature optimisation is the root of all evil."

- Object instantiation in Java is cheap!
- However, sometimes this is a bad idea.

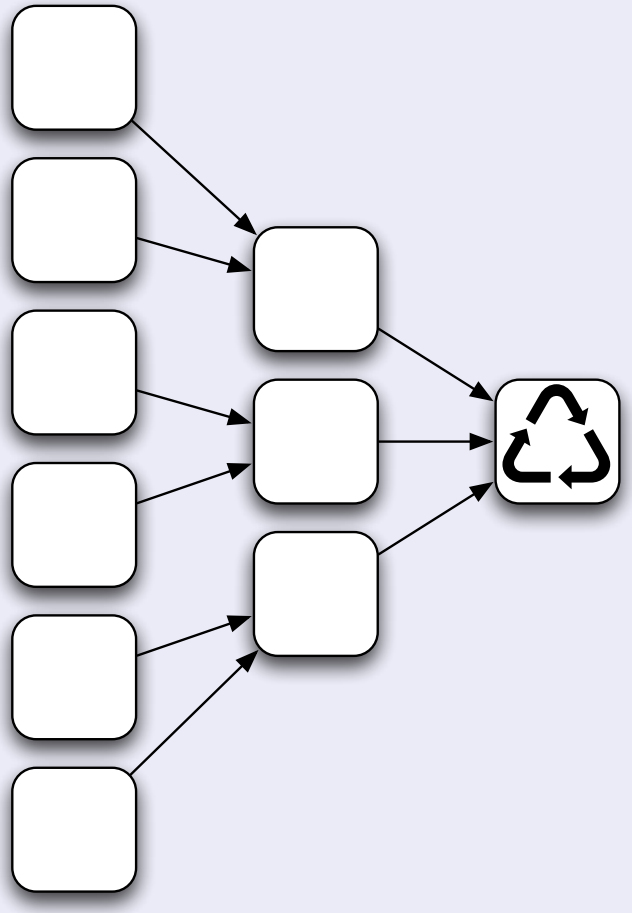
Efficiency of the Static Policy

Dependent Services Graph



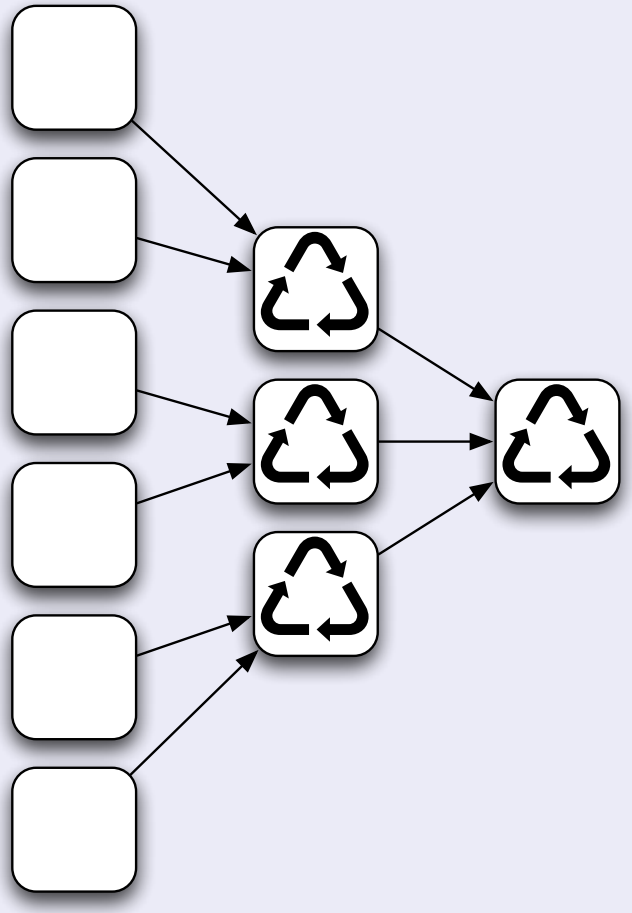
Efficiency of the Static Policy

Dependent Services Graph



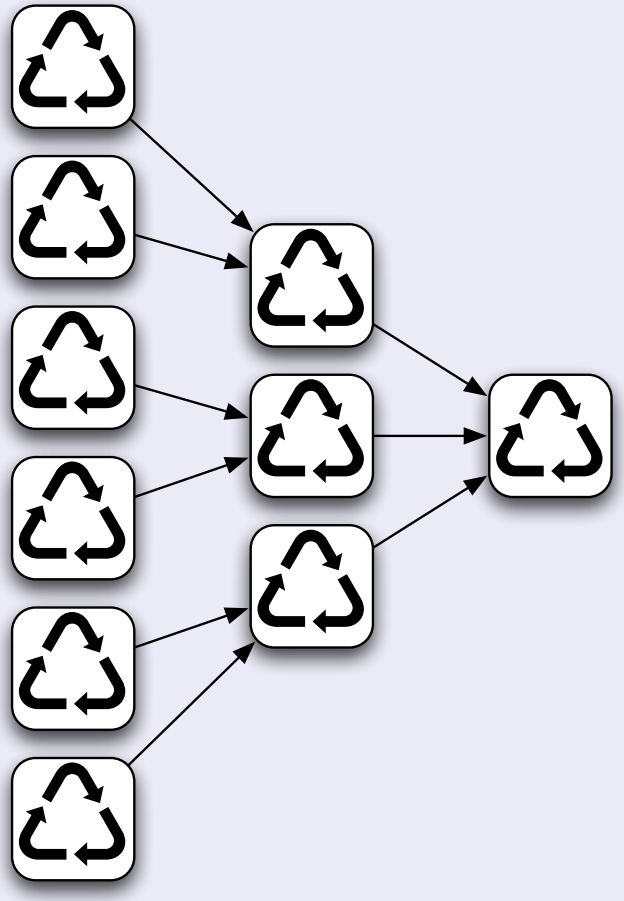
Efficiency of the Static Policy

Dependent Services Graph



Efficiency of the Static Policy

Dependent Services Graph



Dynamic Policy

- Fortunately "static" is only the default. We can change it simply by declaring `policy="dynamic"`.

Example `reference` Element

```
<reference name="LOG"
    interface="org.example.osgi.log.Log"
    bind="setLog" unbind="unsetLog"
    cardinality="0..1"
    policy="dynamic"/>
```

- But now we need to worry about being "dynamic-safe".
- The bind/unbind method might be called while a service method that uses the referent is executing.

Coping with Dynamic Effects

- We already made our `log` field `volatile` for visibility purposes. But this is not enough to ensure we can cope with changes to the referred service.

Accessing the Referred Service

```
if(log != null) {
    log.log(Log.INFO, "This is a message");
}
```

- This is a pattern (or anti-pattern!) known as “check-then-act”.
- It fails because the `log` reference may be non-null on the first line and then be set to null before the second line executes.
- Result: `NullPointerException`.

Coping with Dynamic Effects

- A possible solution is to wrap this code in a `synchronized` block:

Accessing the Referred Service

```
synchronized(this) {
    if(log != null) {
        log.log(Log.INFO, "This is a message");
    }
}
```

- Note we would also have to make the bind and unbind methods `synchronized`.
- However this reduces the concurrency of our program, and it is unwise to hold a lock when calling a service.

Coping with Dynamic Effects

- It is better to use a small `synchronized` block just to read the field, then act outside the block:

Accessing the Referred Service Outside `synchronized`

```
Log log = null;
synchronized(this) {
    log = this.log;
}
if(log != null) {
    log.log(Log.INFO, "This is a message");
}
```

Coping with Dynamic Effects

- A lightweight alternative solution is to use an `AtomicReference`.

Accessing the Referred Service

```
AtomicReference<Log> logRef;

// ...

Log log = logRef.get()
if(log != null) {
    log.log(Log.INFO, "This is a message");
}
```

Dynamic and Mandatory

What is the effect of dynamic policy with a mandatory reference?

- When a mandatory reference is no longer satisfied, the component must be discarded. When the reference becomes satisfied again, the component is recreated.
- This is the case for both static and dynamic policy. So does dynamic policy make sense for a mandatory reference?
- Yes. With dynamic policy, the service reference can be *replaced* if there is another immediately available.

Dynamic and Mandatory

What is the effect of dynamic policy with a mandatory reference?

- When a mandatory reference is no longer satisfied, the component must be discarded. When the reference becomes satisfied again, the component is recreated.
- This is the case for both static and dynamic policy. So does dynamic policy make sense for a mandatory reference?
- Yes. With dynamic policy, the service reference can be *replaced* if there is another immediately available.

Dynamic and Mandatory

What is the effect of dynamic policy with a mandatory reference?

- When a mandatory reference is no longer satisfied, the component must be discarded. When the reference becomes satisfied again, the component is recreated.
- This is the case for both static and dynamic policy. So does dynamic policy make sense for a mandatory reference?
- Yes. With dynamic policy, the service reference can be *replaced* if there is another immediately available.

Dynamic and Mandatory

What is the effect of dynamic policy with a mandatory reference?

- When a mandatory reference is no longer satisfied, the component must be discarded. When the reference becomes satisfied again, the component is recreated.
- This is the case for both static and dynamic policy. So does dynamic policy make sense for a mandatory reference?
- Yes. With dynamic policy, the service reference can be *replaced* if there is another immediately available.

Dynamic Service Replacement

Beware!

When a service is dynamically replaced, SCR **first** calls the bind method with the new service, **then** calls the unbind method with the old service. Therefore the naïve unbind method as follows is **wrong**:

```
protected synchronized void unbind(Log log) {
    this.log = null;
}
```

Dynamic Service Replacement

- To do this correctly in a synchronized block we would have to check the value as follows:

Correct Unbinding with synchronized

```
protected synchronized void unsetLog(Log log) {
    if(this.log == log) this.log = null;
}
```

Dynamic Service Replacement

- With `AtomicReference` it can be done more elegantly:

Correct Unbinding with `AtomicReference`

```
protected void unsetLog(Log log) {
    logRef.compareAndSet(log, null);
}
```

- This atomically sets the value of `logRef` to `null` iff the current value is `log`.

Multiple References

- So far we have looked only at unary references, `0..1` or `1..1`.
- We noted that `0..n` and `1..n` also exist.
- At this point there is nothing difficult or surprising in going from unary to multiple references.
- With a multiple reference the bind/unbind methods are called for each individual service instance.

Multiple References

- So far we have looked only at unary references, `0..1` or `1..1`.
- We noted that `0..n` and `1..n` also exist.
- At this point there is nothing difficult or surprising in going from unary to multiple references.
- With a multiple reference the bind/unbind methods are called for each individual service instance.

Multiple References

- So far we have looked only at unary references, `0..1` or `1..1`.
- We noted that `0..n` and `1..n` also exist.
- At this point there is nothing difficult or surprising in going from unary to multiple references.
- With a multiple reference the bind/unbind methods are called for each individual service instance.

Multiple References

- So far we have looked only at unary references, `0..1` or `1..1`.
- We noted that `0..n` and `1..n` also exist.
- At this point there is nothing difficult or surprising in going from unary to multiple references.
- With a multiple reference the bind/unbind methods are called for each individual service instance.

Multiple References

Important

Multiple references **must** use the dynamic policy. Static policy simply makes no sense. Thread safety is therefore important and unavoidable.

Contents

Publishing a Service

- The final piece of the puzzle is publishing our component as a service itself.
- This is done with the `<service>` element in our XML descriptor.

Service Element

```
<service>
    <provide interface="net...ContactRepository"/>
</service>
```

- Provide multiple services simply by adding additional `<provide>` elements.

Service Properties

We can specify service properties using the <property> element. These properties are passed to the component in activation *and* published to the service registry.

Property Element

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">

   <implementation class="..."/>

   <property name="foo" value="bar"/>
   ....
```

Lab 2 : Using the Dynamic Policy

1. Switch both references in the `directoryRepository.xml` to use dynamic policy.
2. Fix the component implementation to properly reflect dynamics.
3. Repeat the tests from the previous exercise.

Lab 2 : Publishing as a Service

1. Publish the component under the `ContactRepository` interface.
2. Set the service property `name` to the value `Directory`.
3. Check the output of the `services` command in the console.

Expected Results

```
osgi> start 7

osgi> services
...
{net.eclipsetraining.osgi.contacts.api.ContactRepository}=
    {name=Directory, dirName=contacts,
    component.name=DirectoryRepository,
    component.id=1, service.id=35}
Registered by bundle: contacts.core.directory_ds_0.0.0 [7]
No bundles using service.
...
```

Two Important Points

1. The service appears to have been registered by our bundle even though it was *really* registered by SCR. Other bundles just see a normal service, they are not aware we are using DS.
2. Something was missing in the output on the previous slide... the print statement from the constructor, "DirectoryRepositoryDS constructed". Why?

Contents

Lazy Service Creation

- By default, SCR creates “delayed services”.
- These are services that are registered in the service registry but the implementation object has not yet been instantiated.
- It will be instantiated “on-demand” when the first client attempts to actually use the service.

The Problem with Eager Service Registration

- Programmatic registration from a bundle activator results in services being registered *eagerly*.
- As soon as our bundle starts, our service is created.
- But we have no idea if any consumer will ever need our service!

The Problem with Eager Service Registration

- Programmatic registration from a bundle activator results in services being registered *eagerly*.
- As soon as our bundle starts, our service is created.
- But we have no idea if any consumer will ever need our service!

The Problem with Eager Service Registration

- Programmatic registration from a bundle activator results in services being registered *eagerly*.
- As soon as our bundle starts, our service is created.
- But we have no idea if any consumer will ever need our service!

The Cost of Eager Service Registration

- Most simple service objects are cheap to create. . .
- . . . but the class loader for the bundle is not quite so cheap!
- Hundreds of bundles providing services that are never used creates significant classloading overhead.
- If our bundle has no activator and uses "delayed" services, there is no need for the framework to create a classloader until the service is *actually used*.

Immediate Services

Delayed creation is the default for services, but it can be turned off.

- Service components can be declared with `immediate="true"` to make SCR instantiate them immediately (assuming their dependencies are satisfied).
- Non-service components are always “immediate”.

Part III

Spring-DM

Contents

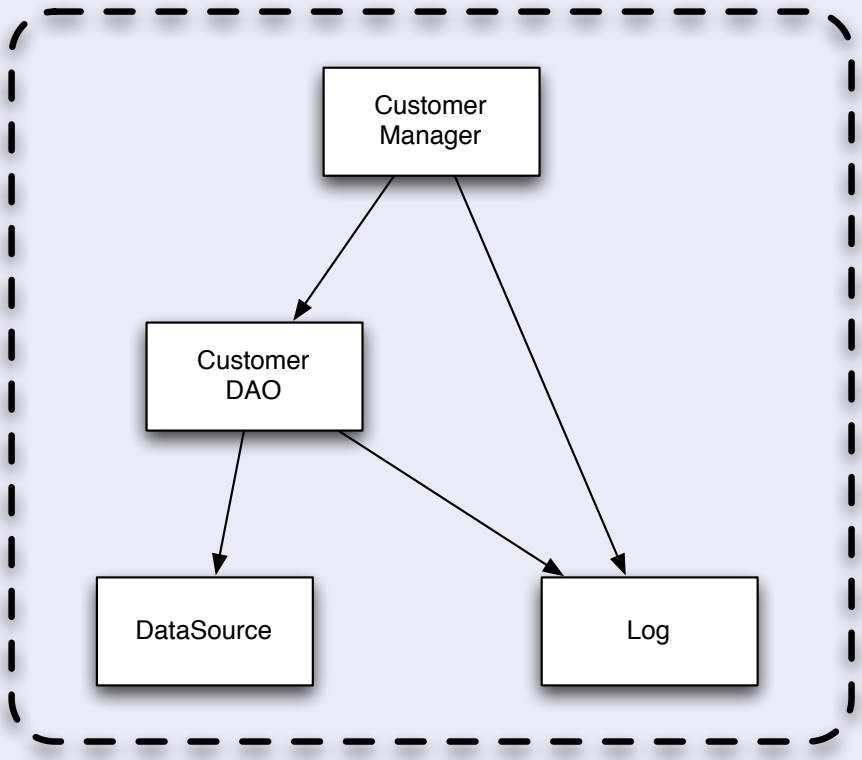
Spring Dynamic Modules (DM)

- The Spring Framework is an extremely popular Dependency Injection (DI) framework.
- It is widely used in many Java deployment scenarios, but particularly in J2EE.
- It has a strong focus on POJOs, interface-based development, and testability.

The Spring Framework

Spring creates a container called the Application Context. Within the container are multiple *beans*.

Spring Application Context



Spring Beans

Beans in Spring tend to be pure POJOs following the JavaBean pattern.

CustomerDAO

```
public class CustomerDAO {

    private DataSource dataSource;
    private Log log;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public void setLog(Log log) {
        this.log = log;
    }
    // ...
}
```

Spring XML Configuration

Beans are instantiated, configured and "wired" together using an XML declaration file.

Spring XML

```
<beans>
    <bean id="customerDb" class="org.spring...DbDataSource">
        <property name="url" value="jdbc:hsqldb:hsql://localhost"/>
        <property name="username" value="sa"/>
    </bean>

    <bean id="customerDao" class="org.example...CustomerDAO">
        <property name="dataSource" ref="customerDb"/>
    </bean>
</beans>
```

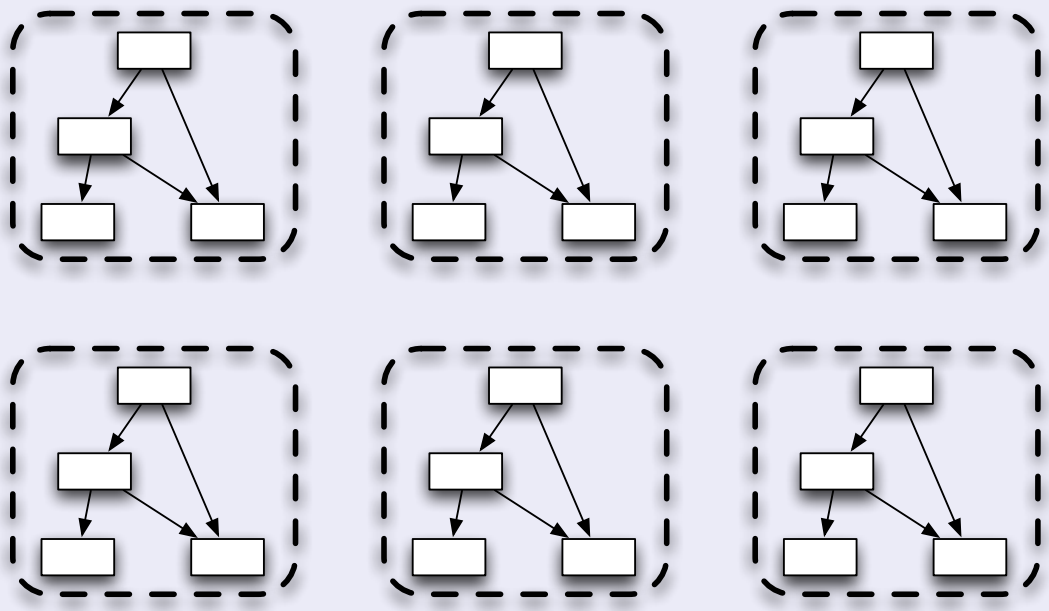
Spring Framework Limitations

- Spring works well and a huge number of components and patterns are available.
- However it can be painful to scale up to large complex applications.
- Thousands of beans in a flat namespace.
- Start-up ordering concerns.
- *Lack of modularity*.

Spring Dynamic Modules

- Spring-DM uses OSGi to modularise large Spring applications.
- Does not fundamentally change the Spring container model... just allows the creation of multiple interoperating containers.

Spring-DM

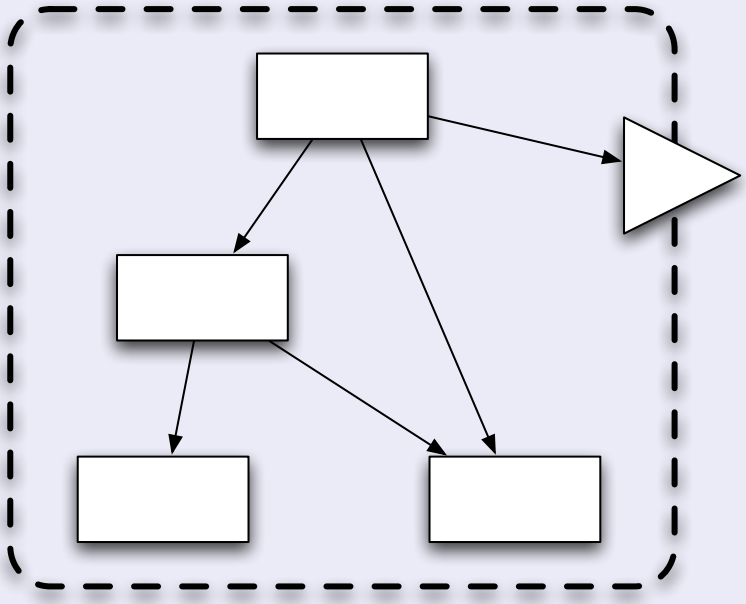


Bridging Application Contexts

- Spring-DM allows a large Application Context to be broken down into many small ones.
- It also offers "glue" between App Contexts based on *importing* and *exporting* beans.

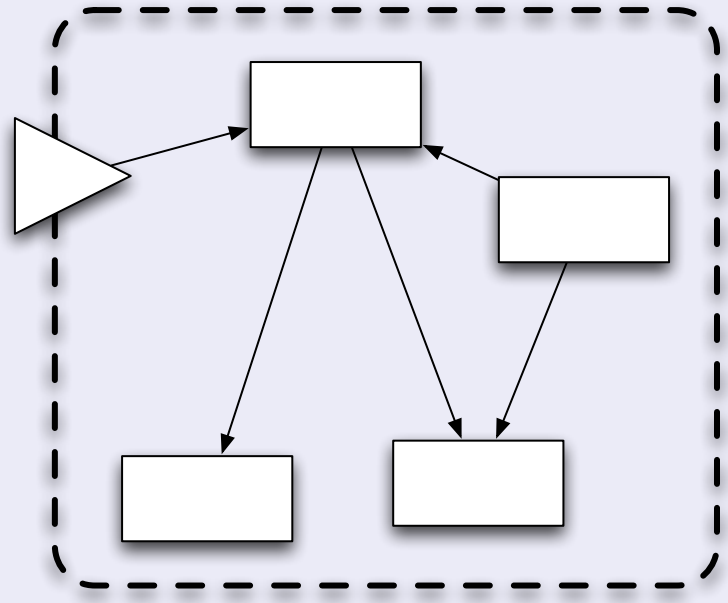
Bridging Application Contexts

Exporting a Bean



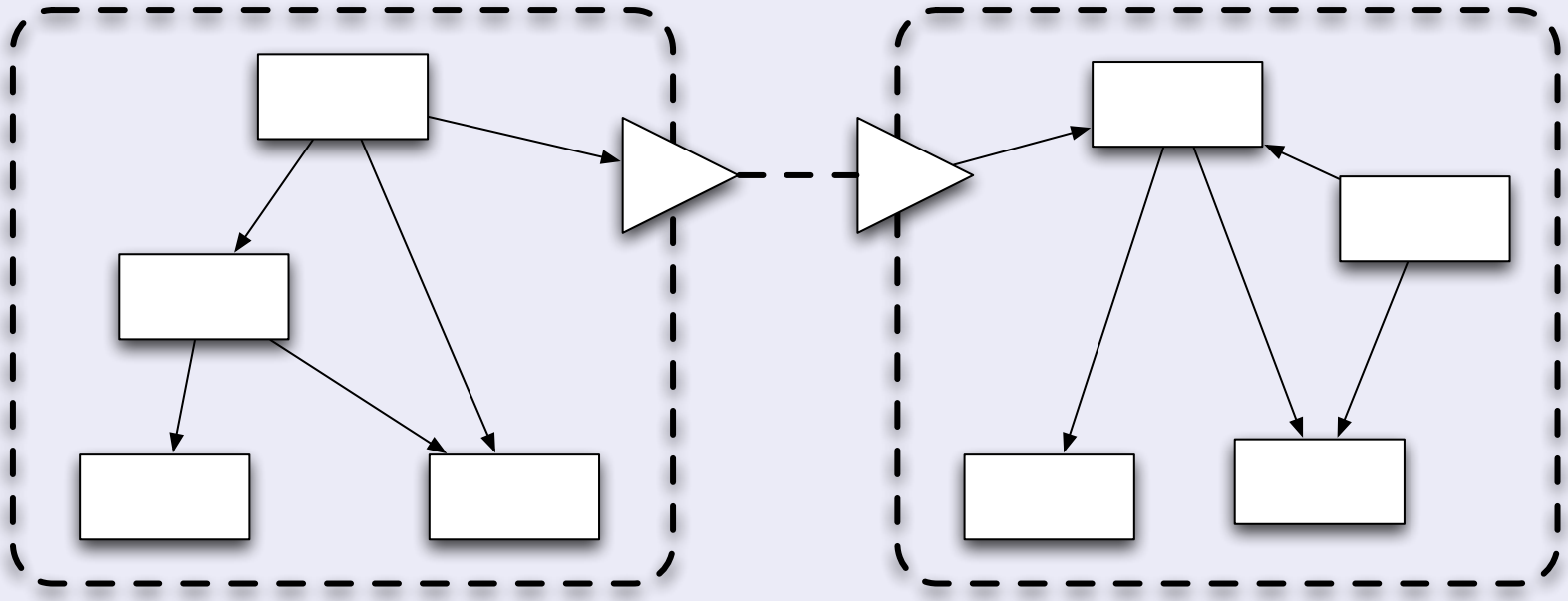
Bridging Application Contexts

Importing a Bean



Bridging Application Contexts

Exporting and Importing a Bean



Contents

A Minimal Example

minimal.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"/>

    <bean id="hello" class="org.example.osgi.ds.HelloComponent"/>

</beans>
```

Spring-DM Application Context

- Like DS, Spring-DM uses an extender bundle which looks for XML files.
- Placed in a common location (`META-INF/spring/*.xml`) or referenced by a `Spring-Context` manifest header.

MANIFEST.MF

```
Spring-Context: config/context.xml, config/context-osgi.xml
```

Running the Example

- We need to install rather more bundles for Spring-DM to work... 12!
- One of them is the extender bundle and must be started:
- `org.springframework.osgi.extender_<version>.jar`

Starting Spring-DM

```
osgi> ss
...
17 RESOLVED     org.springframework.osgi.extender_1.2.0.m2
...

osgi> start 17
INFO [org.spring...] - Starting [...extender] bundle v.[1.2.0.m2]
INFO [org.spring...] - No custom extender configuration detected...
INFO [org.spring...] - Initializing Timer
```

Spring-DM Application Context

- Unlike DS, each XML file is *not* a single component.
- Spring-DM creates *one* Application Context for the whole bundle. The contents is the union of all the listed XML files.
- Some parts of this XML will be *OSGi-specific*.

Best Practice

OSGi-specific parts of the Spring XML should be placed in a separate file. This enables re-use of the generics parts when used in a non-OSGi runtime.

Contents

Activation and Deactivation

There are no default method names for activation and deactivation, we must specify them via the bean declaration:

Init and Destroy Methods

```
<bean id="polling" class="org.example.osgi.dm.PollingComponent"
        init-method="activate"
        destroy-method="deactivate">
    <property name="period" value="2000"/>
</bean>
```

These methods are always zero-arg. Configuration is supplied via our setter methods, or constructor injection.

Contents

References to Services

- "Importing" a bean from another context means referencing it as a service.

Service Imports

```
<osgi:reference id="importedLogSvc"
    interface="org.osgi.service.log.LogService"/>

<bean id="loggingPollingBean" class="org.example...">
    <property name="log" ref="importedLogSvc"/>
</bean>
```

- The reference element looks just like another bean, and can be referenced by other beans using its ID as normal.

Proxy Beans

- In fact the reference really *is* just another bean. It is a generated proxy that continuously tracks the backing service.
- The proxy is injected *once* into each bean that refers to it.

Important!

Spring-DM does *not* directly inject service objects into its beans, but injects a proxy object. The reference from the bean to the proxy remains constant, so the bean does not have to deal with the dynamics.

Dynamics and Thread Safety

- This swaps one problem – thread safety – for another: the beans using the service have no idea whether the service is presently available.
- What happens when a bean calls the proxy and no backing service is present?
- DM uses an (optional) timeout to wait for a suitable service to appear.
- Ultimately it must throw an unchecked exception, `ServiceUnavilableException`.

Listening to Services

- This is nice and simple, but sometimes we really need to know the state of the service. For example we might be able to offer alternative functionality when the backing service is unavailable.
- E.g. when there is no `LogService`, print to the console.
- For this we can nominate a bean as a *service listener*.

Listening to Services

Service Listener

```
<osgi:reference id="importedLogSvc"
        interface="org.osgi.service.log.LogService">
    <listener ref="log" bind-method="setLog"
                        unbind-method="unsetLog"/>
</osgi:reference>

<bean id="log" class="org...MyLog"/>
```

Note that the `log` bean declaration could be in the non-OSGi part of our application context.

Listening to Services

- Just like DS with dynamic policy, the bind/unbind methods can be called concurrently with other methods accessing the bean.
- Therefore beans configured as listeners must be thread-safe.
- There is no equivalent of static policy.
- NB: when service replacement occurs, unbind is not called at all. Unbind is *only* called when the service is going away and there is no available replacement.

Multiple References

- The <reference> element is used for references to single services. For multiple services, there are two choices: <list> and <set>.
- Each of these gives us a *managed collection* object which can be injected as a constant bean into other beans.

List and Sets

```
<osgi:list id="listenerList"
    interface="org.osgi.service.log.LogListener"/>

<osgi:set id="listenerSet"
    interface="org.osgi.service.log.LogListener"/>

<bean id="log" class="org...MyLog">
    <property name="listeners" ref="listenerList"/>
</bean>
```

Multiple References

- Spring-DM adds and removes service objects in this collection transparently.
- A listener bean can be nominated, as with singular references.
- Iterators over these collections are *stable*. If a call to `hasNext` returns true then the subsequent call to `next` is guaranteed to return non-`null`.

Contents

Mandatory Dependencies

- Like DS, Spring-DM supports both optional and mandatory services via a simple declaration.

Optional Service Reference

```
<osgi:reference id="logSvc" interface="org...log.LogService"
        cardinality="0..1"/>

<osgi:list id="listeners" interface="org...log.LogListener"
        cardinality="0..N"/>
```

- The default cardinality can be controlled in the outer <beans> element, but the default default is 1..1 or 1..N, i.e. mandatory.

Mandatory Dependencies

- If there are *any* unsatisfied mandatory references, the entire application context will not start.
- Obviously we cannot have a mandatory ref to a service exported by the same application context!
- If a mandatory reference becomes unsatisfied, the application context is **not** deactivated and no active components will be deactivated.
- Instead any service exports that depend on the reference (whether directly or transitively) will be unregistered.

Contents

Publishing a Service

- Any bean within the application context can be published as a service.

Simple Publication

```
<osgi:service ref="contacts" interface="net...ContactRepository"/>
```

More Publication Options

```
<osgi:service ref="contacts">
    <osgi:interfaces>
        <value>net...ContactRepository</value>
        <value>net...EventHandler</value>
    </osgi:interfaces>
</osgi:service>

<osgi:service ref="contacts" auto-export="interfaces"/>

<osgi:service ref="contacts" auto-export="interfaces">
    <osgi:service-properties>
        <entry key="foo" value="bar"/>
    </osgi:service-properties>
</osgi:service>
```

Lab 3 : Building the Directory Repository (Setup)

1. Install the Spring-DM bundles:

```
osgi> installDir ../ComponentsSpringDM/bundles
```

2. Start the Spring-DM Extender bundle (`org.springframework.osgi.extender...`)

```
osgi> start 17
INFO [org.spring...] - Starting...
```

Lab 3 : Building the Directory Repository

1. Complete the implementation of the `DirectoryRepositoryDM` class – TODO markers explain what is needed.
2. Complete the Spring XML descriptor `osgi-context.xml` in the `spring` directory. Add references to the log and listener services.
3. Complete `context.xml` in the same directory. Add properties wired to the service reference beans. Declare the init and destroy methods on the main component bean.

Lab 3 : Publishing as a Service

1. Publish the component under the `ContactRepository` interface.
2. Add the property `name=Directory`.
3. Test as before with the `services` command and in the Swing GUI.

Contents

Advantages of Spring-DM

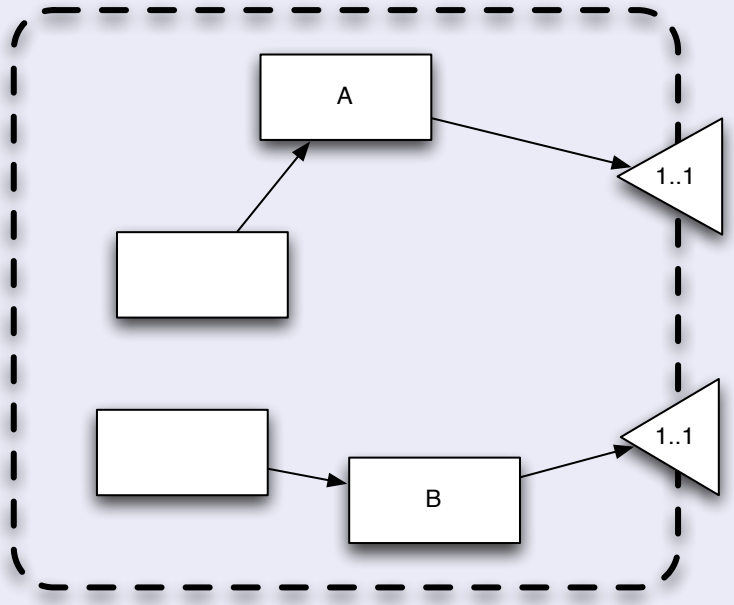
- Spring-DM is clearly the best approach for modularising an *existing* Spring-based application.
- It provides a more mature Dependency Injection framework than DS with features not present in DS (e.g. constructor injection).
- Stronger focus on POJOs than DS.
- Standardised in the OSGi compendium in R4.2 (the Blueprint Service).

Disadvantages of Spring-DM

- Heavy! Twelve JARs totalling 2.1Mb (DS is 170k)
- The "application context" idea has been adapted to fit the OSGi model and it feels like legacy. The result seems to be higher complexity.
- No separate bean life-cycle.
- Lack of laziness.
- A "standard" but will we ever see a competing implementation?

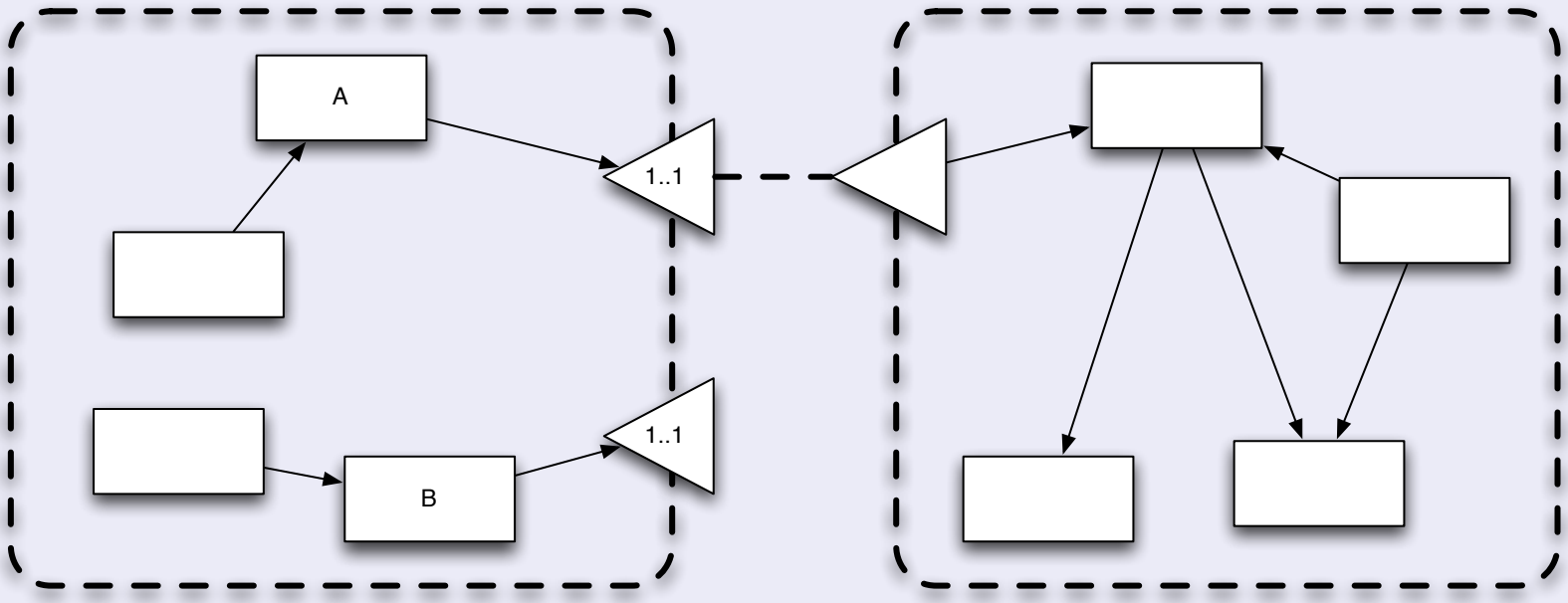
Lack Of Bean Life-cycle

Bundle with 2 Mandatory References



Lack Of Bean Life-cycle

Partially Satisfied



- The App Context on the left will not be activated because one mandatory reference is missing.
- However the individual bean *A* has no missing references.
- Why not activate just *A* and the beans that use it?

Lack of Bean Life-cycle

- When mandatory references become unsatisfied, Spring-DM avoids destroying the application context.
- Beans that are "active" will *not* be deactivated... even if their "mandatory" references are unsatisfied!
- Have to use service listeners if there is active state (e.g. a polling thread) which must be deactivated when a referenced service goes away.

Why does Spring-DM not deactivate the context? Probably because it is expensive to recreate, rather than because it is the right thing to do.

Lack of Laziness

- The application context is created by the Spring-DM extender bundle as soon as our bundle enters `ACTIVE` state.
- This causes classloading events, forcing the framework to create a classloader for our bundle.
- There does not appear to be a mechanism for deferring this activity until the published services are needed by a consumer.

Part IV

Apache iPOJO

Contents

Apache iPOJO

- iPOJO is a sub-project of Apache Felix, however it works on other R4.1 frameworks.
- Like Spring-DM it focuses on creating a pure POJO programming model for components.
- It supports individual “bean” component lifecycle.
- Supports both XML and annotation-based metadata.

Contents

A Minimal Example

Minimal iPOJO Declaration

```
<ipojo>
    <component name="hello" classname="org...HelloComponent"/>
    <instance component="hello"/>
</ipojo>
```

- iPOJO separates the concepts of a component *definition* and component *instances*.
- The former is like a factory.

Running the Example

iPOJO is just one extender bundle, which must be installed and started:

- `org.apache.felix.ipojo-<version>.jar`

BUT we need an additional build step – an ANT task which post-processes the bundle JAR. We sill see why this is necessary shortly.

Contents

Activation and Deactivation

Validate/Invalidate Callbacks

```
<ipojo>
    <component classname="org...PollingComponent">
        <callback transition="validate" method="start"/>
        <callback transition="invalidate" method="stop"/>
    </component>

    <instance component="org...PollingComponent"/>
</ipojo>
```

Like Spring-DM, the activation method is zero-arg, so how are configuration properties set?

Configuration Properties

Properties

```
<ipojo>
    <component classname="org...PollingComponent">
        <callback transition="validate" method="start"/>
        <callback transition="invalidate" method="stop"/>
        <properties>
            <property name="period" field="period"/>
        </properties>
    </component>

    <instance component="org...PollingComponent">
        <property name="period" value="2000"/>
    </instance>
</ipojo>
```

Component Implementation

Receiving a Property

```
public class PollingComponent {

    private int period;

    protected void start() {
        // ...
    }

    protected void stop() {
        // ...
    }
}
```

Direct injection into the private field!

Contents

References to Services

Referring to a service is done with similar magic. Just declare a private field with the type of the service, and use it!

Receiving a Service

```
public class PollingComponent {

    private int period;
    private LogService log;

    protected void start() {
        if(log != null)
            log.log(LogService.LOG_INFO, "Starting polling");
        // ...
    }
```

Declaring a Service Reference

```
<component classname="org...PollingComponent">
    <requires field="log"/>

    ...

</component>
```

Optional Service References

References are mandatory by default, to make optional:

```
<component classname="org...PollingComponent">
    <requires field="log" optional="true"/>

    ...

</component>
```

Contents

Dynamic Services

Notice when we used the optional log service:

```
if(log != null)
    log.log(LogService.LOG_INFO, "Starting polling");
```

This is classic check-then-act, and looks completely unsafe!

Thread Safety

- iPOJO "manages" the required synchronisation for us.
- As soon as a method "touches" a dependency (i.e. an injected field) , iPOJO ensures that these objects are kept until the end of the method.
- All iPOJO components therefore follow a dynamic policy as defined by DS.
- But we can write our code without regard for thread-safety.

Bytecode Instrumentation

Perhaps now you see what the additional build step is for.

- iPOJO makes heavy use of *bytecode instrumentation*.
- Methods are generated giving access to private fields.
- Methods are inspected for access to shared variables, and synchronization instructions inserted.

Contents

Publishing Services

Publishing is done with the `<provides>` element. In the simplest cases, just adding this element is all that is necessary:

Provides Element

```
<component classname="org...PollingComponent">
    <provides/>
</component>
```

This is like Spring-DM's `auto-export="interfaces"`. Reflection is used to publish under all interfaces implemented by the component.

Service Properties

Service properties are added to the `provides` element as follows:

Properties on Provides

```
<component classname="org...PollingComponent">
    <provides>
        <property name="foo" type="string" value="bar"/>
        <property name="period" field="period"/>
    </provides>
</component>
```

Properties can be explicitly provided here, or sourced from fields in the component.

Lab 4 : Building the Directory Repository (Setup)

1. Install the iPOJO bundle:

```
osgi> installDir ../ComponentsiPOJO/bundles
```

2. Start the iPOJO bundle (`org.apache.felix.ipojo...`)

```
osgi> start 5
```

Lab 4 : Building the Directory Repository

1. Complete the implementation of the `DirectoryRepositoryiPOJO` class – TODO markers explain what is needed.
2. Complete the XML descriptor `core.contacts.directory_ipojo.xml`. Add the following:
   - An optional reference to the log service.
   - A zero-to-many reference to the listener service.
   - Callbacks for the activate/deactivate methods.

Lab 4 : Publishing as a Service

1. Publish the component under the `ContactRepository` interface.
2. Add the service property `name=Directory`.
3. Test as before with the `services` command and in the Swing GUI.

Contents

Annotation-Based Declarations

- As an alternative to XML, iPOJO supports annotations directly in the component source code.
- These annotations are retained in the compiled bytecode but not the runtime VM.

Annotation-Based Declarations

MyCustomerDAO

```
@Component
@Provides
public class MyCustomerDAO implements CustomerDAO {
    @Requires
    private DataSource dataSource;

    public Customer lookupCustomer(String name) {
        Connection conn = dataSource.getConnection();
        // ...
    }

    @Validate
    public void starting() { /* ... */ }

    @Invalidate
    public void stopping() { /* ... */ }
}
```

Annotation-Based Declarations

- We can use annotations together with XML.
- The annotations control the component definition (i.e. `<component>` element).
- XML would then be used to produce instances.

Contents

iPOJO Handlers

- DS and Spring-DM use a "monolithic container", i.e. all the aspects of component control (service refs, service provides, configuration, life-cycle) are implemented in one indivisible runtime (the SCR or Spring-DM extender).
- iPOJO separates these tasks into individual *handlers*.
- Handlers are provided out of the box for the common use-cases (refs, provides, etc.).
- We can extend the behaviour of iPOJO with custom handlers.
- Example: JMX instrumentation of component state.

Contents

Advantages of iPOJO

- Very clean POJO coding style for components.
- Extreme flexibility through handlers.

Disadvantages of iPOJO

- Bytecode engineering and extensive annotations: are these really still POJOs??
- Is the blanket promise to "handle concurrency" really credible?
- Additional build steps.

Thread-Safety

- Yes, thread-safety *is* easy to screw up, and Java offers weak support for getting it right. Worse, too many Java programmers are unaccustomed to concurrent programming. Remember, J(2)EE literally *forbids* the use of threads.
- Is the solution to delegate to a container?
- No. Truly re-usable components should not rely on a special container to make them robust, they should be internally robust.
- Well-written DS components are *safe in almost any context*. iPOJO components are only safe in iPOJO or a non-threaded environment.

Thread-Safety

- Yes, thread-safety *is* easy to screw up, and Java offers weak support for getting it right. Worse, too many Java programmers are unaccustomed to concurrent programming. Remember, J(2)EE literally *forbids* the use of threads.
- Is the solution to delegate to a container?
- No. Truly re-usable components should not rely on a special container to make them robust, they should be internally robust.
- Well-written DS components are *safe in almost any context*. iPOJO components are only safe in iPOJO or a non-threaded environment.

Thread-Safety

- Yes, thread-safety *is* easy to screw up, and Java offers weak support for getting it right. Worse, too many Java programmers are unaccustomed to concurrent programming. Remember, J(2)EE literally *forbids* the use of threads.
- Is the solution to delegate to a container?
- No. Truly re-usable components should not rely on a special container to make them robust, they should be internally robust.
- Well-written DS components are *safe in almost any context*. iPOJO components are only safe in iPOJO or a non-threaded environment.

Thread-Safety

- Yes, thread-safety *is* easy to screw up, and Java offers weak support for getting it right. Worse, too many Java programmers are unaccustomed to concurrent programming. Remember, J(2)EE literally *forbids* the use of threads.
- Is the solution to delegate to a container?
- No. Truly re-usable components should not rely on a special container to make them robust, they should be internally robust.
- Well-written DS components are *safe in almost any context*. iPOJO components are only safe in iPOJO or a non-threaded environment.

Part V

Conclusion

Contents

Component Model Interoperability

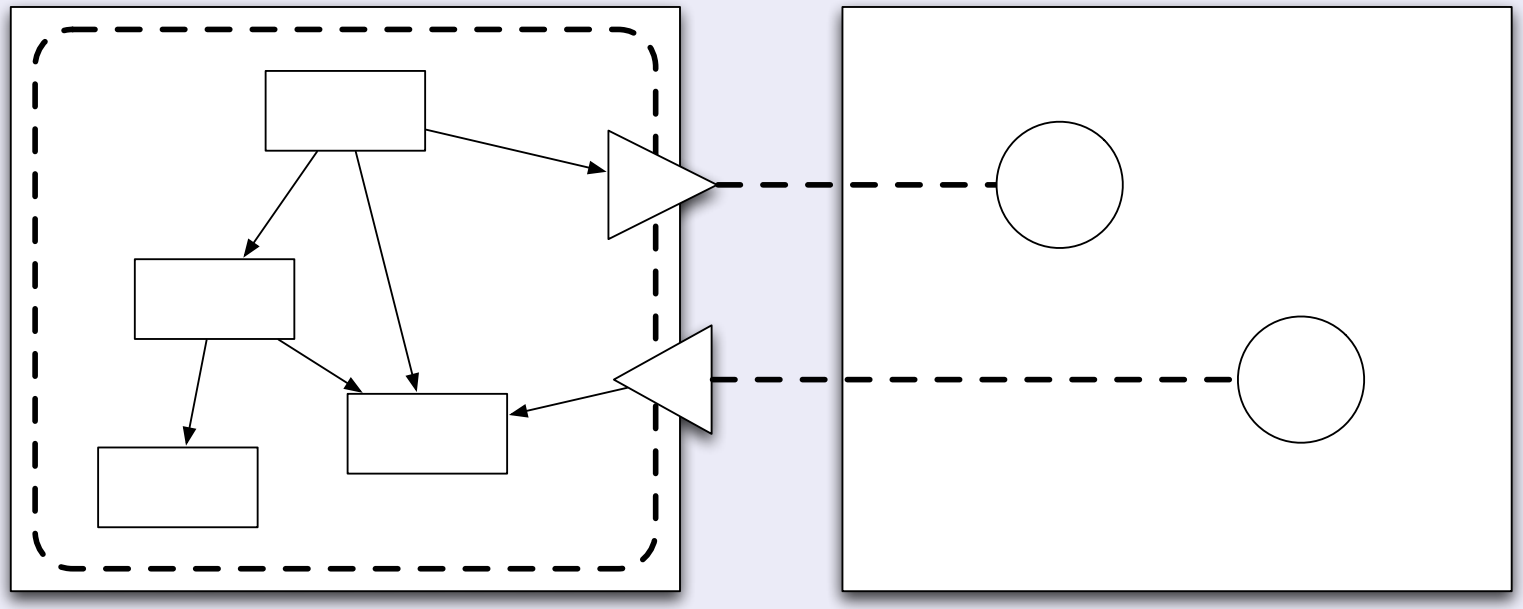
Great News!

All of the discussed component models for OSGi (and others e.g. Peaberry for Google Guice) interoperate *perfectly* at the services level!

Component Model Interoperability

We can easily use a Spring-DM with DS or iPOJO bundles on either side:

Exporting and Importing a Bean



Component Model Interoperability

- In all cases, the imported and exported artefacts are *just* services.
- The use of DS, Spring-DM etc. can be considered merely an implementation detail of each bundle.
- We are not stuck with our choice of component model until the end of time!
- We can use third party components developed using other component models.