OSGi

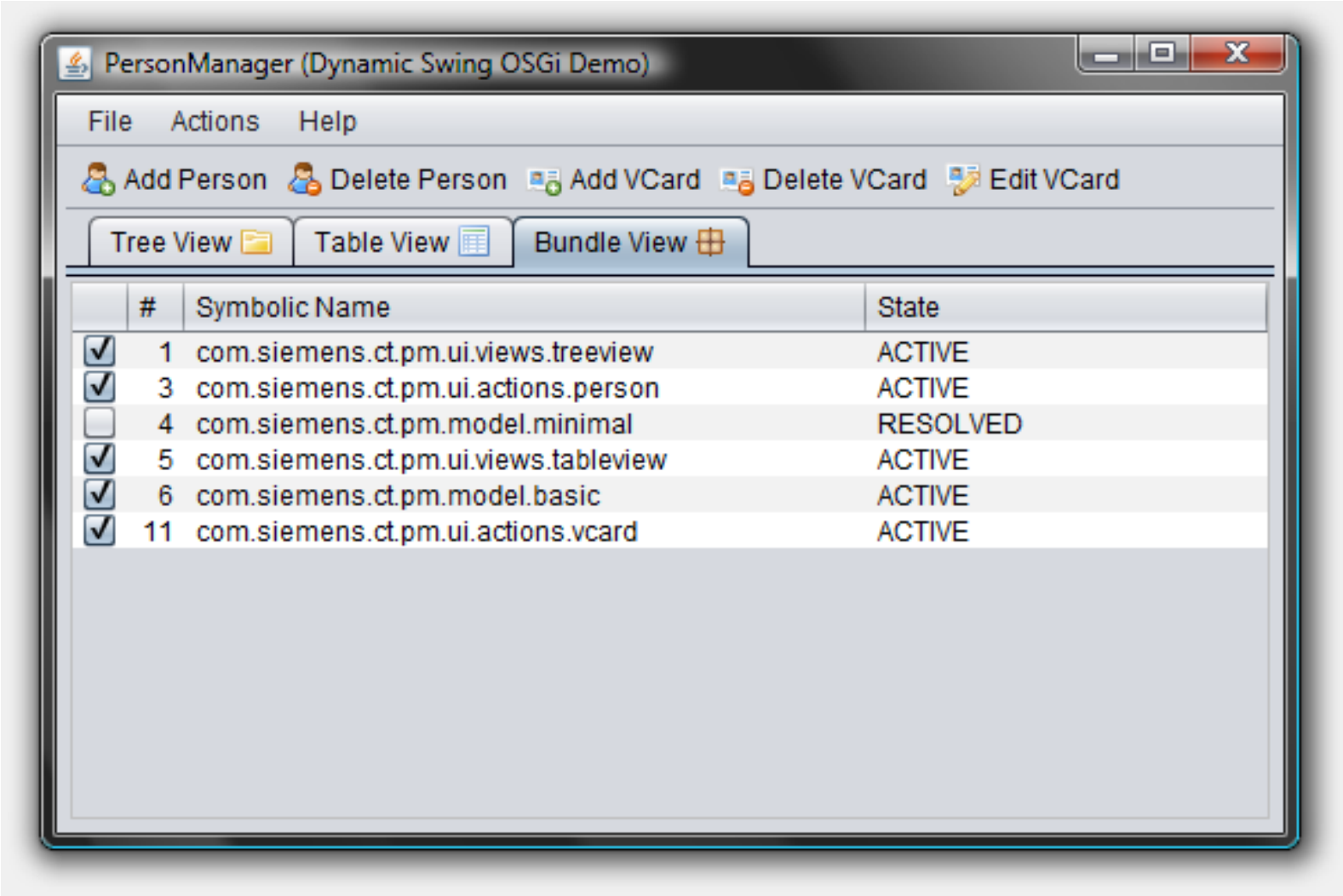
Declarative Services versus Spring Dynamic Modules

Heiko Seeberger
WeigleWilczek
Kai Tödter
Siemens Corporate Technology

Agenda

- Demo: A Swing App Framework based dynamic OSGi application
- OSGi Services
  - Providing and consuming Services
  - Service Tracker
- Overview Declarative Services & Spring Dynamic Modules
- Declarative Services in Detail
- Spring Dynamic Modules in Detail
- Comparison (DS vs. DM)
- Conclusion & Best Practices
- Discussion

Demo



How to get the Demo?

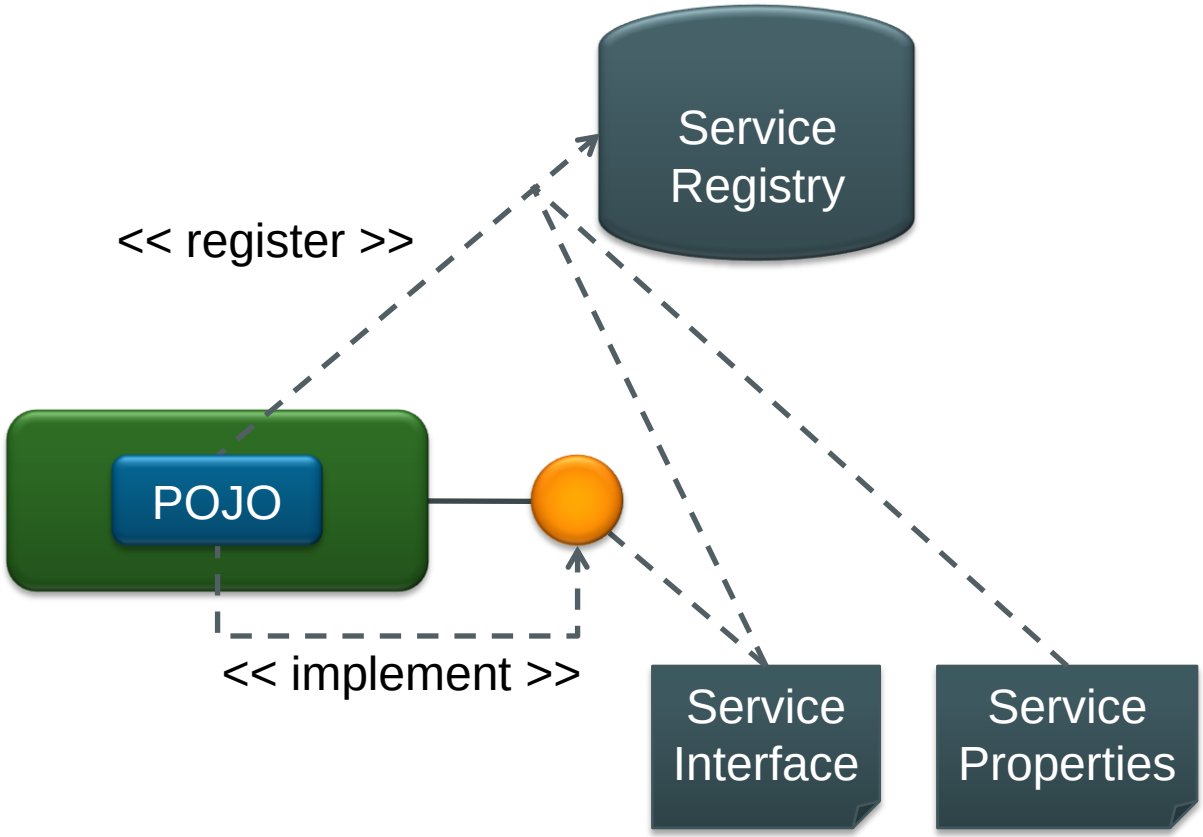
- The PM Demo project home page is: http://max-server.myftp.org/trac/pm
- There you find
  - Wiki with some documentation
  - Anonymous Subversion access
  - Trac issue tracking
- Licenses
  - All PM project sources are licensed under EPL
  - Swing Application Framework (JSR 296) implementation is licensed under LGPL
  - Swing Worker is licensed under LGPL
  - The nice icons from FamFamFam are licensed under the Creative Commons Attribution 2.5 License

Agenda

- Demo: A Swing App Framework based dynamic OSGi application
- OSGi Services
  - Providing and consuming Services
  - Service Tracker
- Overview Declarative Services & Spring Dynamic Modules
- Declarative Services in Detail
- Spring Dynamic Modules in Detail
- Comparison (DS vs. DM)
- Conclusion & Best Practices
- Discussion

OSGi is Service-Oriented!

Providing a Service (1)

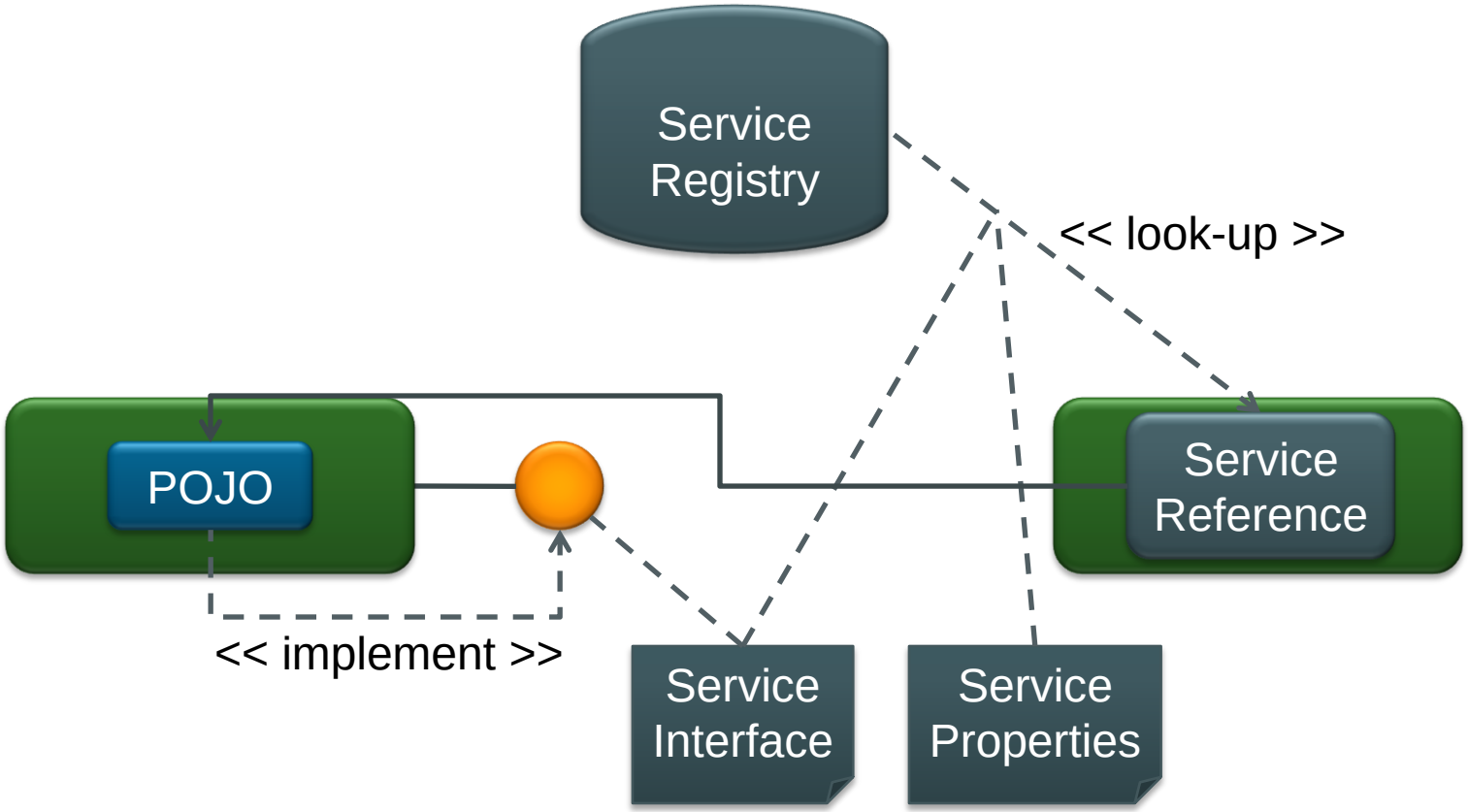


Providing a Service (2)

- An OSGi service is a POJO ...
- ... registered under one ore more service interface names ...
- ... and optional service properties

BundleContext
+ registerService(String, Object, Dictionary): *ServiceRegistration* + registerService(String[], Object, Dictionary): *ServiceRegistration*

Consuming a Service (1)



Consuming a Service (2)

- First a service reference is looked-up by the Service Interface name
- Second the service object is obtained using the Service Reference
  - The Service Registry keeps track of obtained service objects
- When no longer needed, a service object must be returned

BundleContext
+ getServiceReference(String): *ServiceReference* + getService(*ServiceReference*): Object + ungetService(*ServiceReference*): boolean

How to deal with Service Dynamics?

Services might come and go

- Service Look-ups might fail, because ...
  - ... bundle(s) providing service not started yet
  - ... service(s) not registered yet
  - ... bundle(s) providing service stopped
  - ...

Best practice:
Better listen to service events than looking up services.

Service Listeners ...

- ... will receive all service events
  - REGISTERED, MODIFIED, UNREGISTERING
- ... or a filtered subset

BundleContext
+ addServiceListener(*ServiceListener*): void + addServiceListener(*ServiceListener*, String): void

Dynamic Services are difficult to handle (1)

Typical task:
Track all services of a certain type, i.e. all that are already there as well as all that will come and go.

- Possible solution:
  - (1) Frist look-up all existing services
  - (2) Then register a Service Listener to stay tuned
- Problem: Possibility to miss out on some services registered or unregistered between (1) and (2)

Dynamic Services are difficult to handle (2)

- Alternative solution: Reverse the steps
  - (1) First register a Service Listener to stay tuned
  - (2) Then look-up all existing services
- Problem: Watch out for duplication!

Best practice:
Service Listeners are low-level and their usage is error-prone.
Better use Service Trackers.

The Service Tracker

- Utility class specified in OSGi Service Compendium
  - Package: “org.osgi.util.tracker”
  - Based on Service Listeners and “a lot of knowledge”
- Services to be tracked may be specified by ...
  - ... a Service Interface name
  - ... a filter
- Must be opened to start and closed to stop tracking
- Handles service events via the methods
  - `addingService()`
  - `modifiedService()`
  - `removedService()`

Services in very dynamic Scenarios

- Programmatic use of dynamic services works, but has some drawbacks:
  - Eager creation of services leads to
    - Large memory footprint
    - Long startup time
  - Increasing complexity with decreasing robustness

- But is there anything out there that helps here?
- Yes, services and dependencies can be specified declaratively!
  - Declarative Services
  - Spring Dynamic Modules
  - Others (e.g. iPOJO)

Agenda

- Demo: A Swing App Framework based dynamic OSGi application
- OSGi Services
  - Providing and consuming Services
  - Service Tracker
- Overview Declarative Services & Spring Dynamic Modules
- Declarative Services in Detail
- Spring Dynamic Modules in Detail
- Comparison (DS vs. DM)
- Conclusion & Best Practices
- Discussion

Overview DS & DM

- The next 4 slides show just a very brief overview and examples
- Details follow in the rest of this presentation

Declarative Services (DS)

- DS is part of the OSGi R4 spec (Service Compendium)
- DS let you declare components in XML
- The declarations live in OSGI-INF/<component>.xml
- Components can provide services
- Components can depend on other services
  - Dependency injection for references to other services:
  - References may be bound to bind()-/unbind()-methods in components
  - A cardinality and a creation policy can be defined
- Components need the bundle manifest header *Service-Component*
  e.g. **`Service-Component: OSGI-INF/personManager.xml`**

PM Example DS Component

```
<component name="pm.ui.actions.person.ActionContribution">
    <implementation
        class="pm.ui.actions.person.ActionContribution"/>
    <service>
       <provide interface=
            "pm.application.service.IActionContribution"/>
    </service>
    <reference name="PersonManager"
        interface="pm.model.IPersonManager"
        bind="setPersonManager"
        unbind="removePersonManager"
        cardinality="0..1"
        policy="dynamic"/>
</component>
```

Spring Dynamic Modules (DM)

- Integration of Spring DI and OSGi
- XML files are placed in META-INF/spring directory, usually
  - One XML file to define a Spring bean
  - One XML file to map this bean to an OSGi service
- Uses Spring dependency injection for references to other services and POJOs
- Similar but more flexible/powerful approach compared with DS
  - But needs around 12 additional Spring and logging bundles to run

PM Example Spring DM Component

XML for Spring Bean:

```
<beans (schema attributes omitted)>
  <bean name="savePerson"
        class="pm.ui.actions.save.ActionContribution"/>
</beans>
```

XML for OSGi service mapping:

```
<beans (schema attributes omitted)>
<osgi:service id="savePersonOSGi" ref="savePerson"
   interface="pm.application.service.IActionContribution"/>
</beans>
```

Agenda

- Demo: A Swing App Framework based dynamic OSGi application
- OSGi Services
  - Providing and consuming Services
  - Service Tracker
- Overview Declarative Services & Spring Dynamic Modules
- Declarative Services in Detail
- Spring Dynamic Modules in Detail
- Comparison (DS vs. DM)
- Conclusion & Best Practices
- Discussion

Service Component Runtime

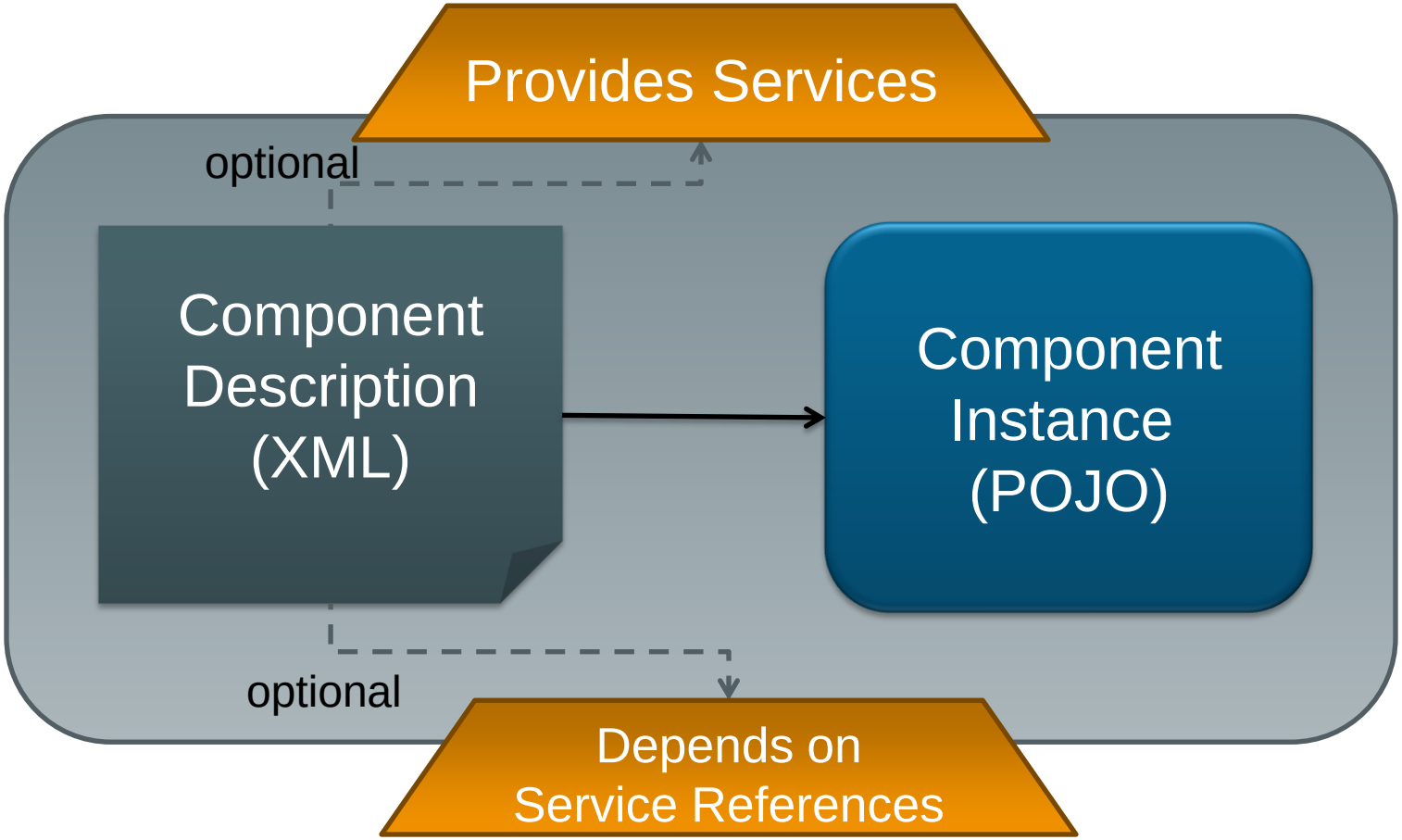
- Providing a service means
  1. Providing an implementation of a service interface
  2. Register the implementation with the service registry
- In programmatic approaches, this is usually done by the same bundle
- In DS, the Service Component Runtime (SCR) takes care of
  - Registering/Unregistering the service implementation
  - Managing the lifecycle
- DS introduces the notion of a Component that
  - Can provide services
  - Can depend on other services

Needed OSGi Bundles

To run DS with Equinox, the following bundles are needed:

- org.eclipse.osgi
- org.eclipse.osgi.services
- org.eclipse.equinox.ds
- org.eclipse.equinox.util

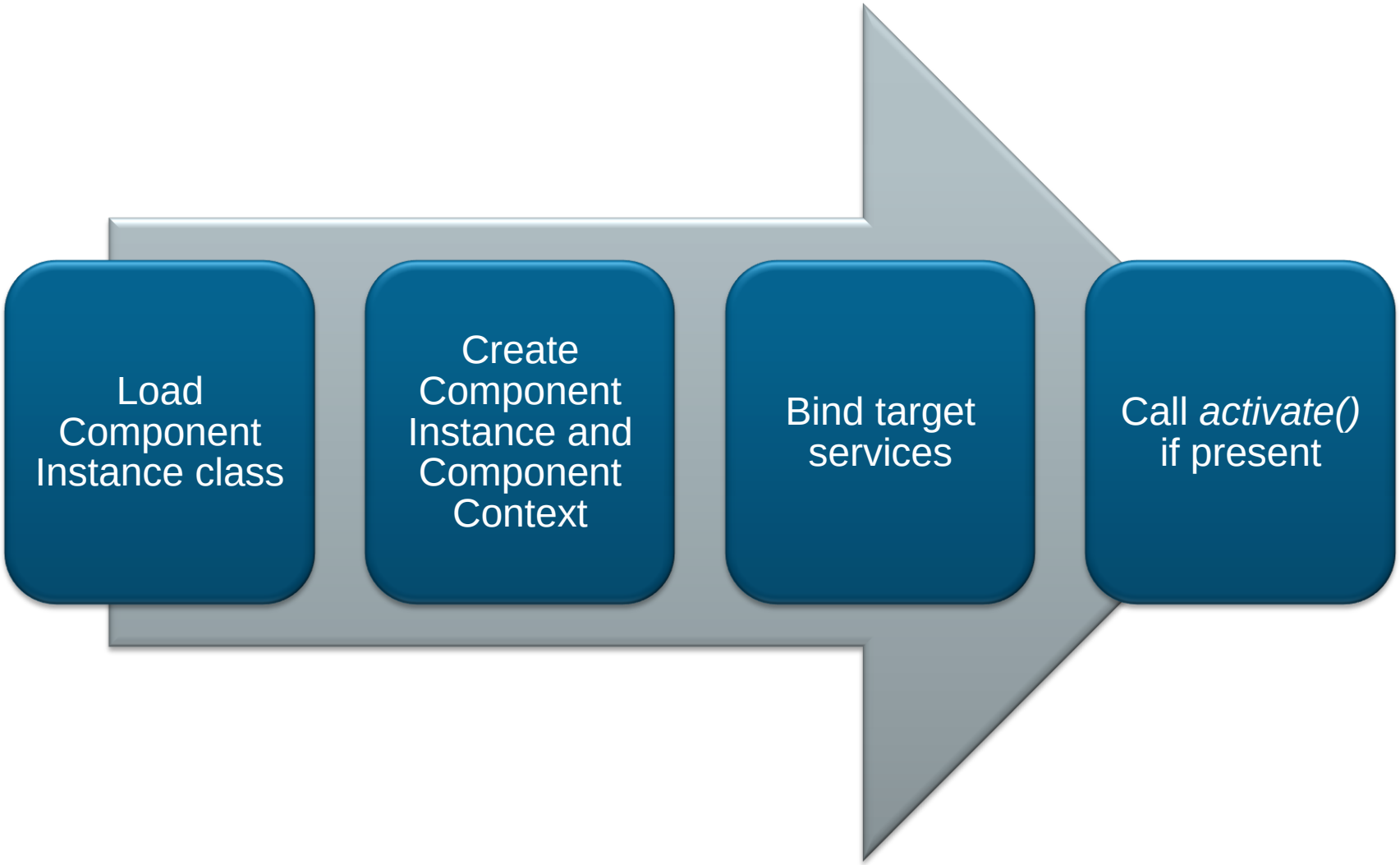
DS Components



Component Activation

- Lazy Activation (Delayed Component)
  - All dependencies must be resolved
  - On first request of the service interface the component is activated
  - This is the default behavior for components providing services
- Eager Activation (Immediate Component):
  - As all dependencies can be resolved the component is created
  - This is expressed in XML by the component attribute **`immediate="true"`**
  - This should be used only if there is a good reason for!
  - All components not providing services must be immediate components!

Component Activation (1)



Component Activation (2)

- Component Instance class needs a default constructor
- *activate()* and *deactivate()* may be supplied
  - Methods are found by reflection
  - May also be defined in super classes

```
protected void activate(ComponentContext context)
protected void deactivate(ComponentContext context)
```

Defining a DS Component

- The **`<component>.xml`** file
  - Has to be placed in a directory OSGI-INF
  - Contains the XML code

```
<component name="ct...person.ActionContribution">
   <implementation
        class="ct...person.ActionContribution"/>
</component>
```

  - Optional component attributes
    - **`immediate`**: if set true, the component is activated eagerly
  - Optional tags (see next slides)
    - service
    - reference
- Components need the bundle manifest header Service-Component
  e.g. **`Service-Component: OSGI-INF/<component>.xml`**

Services and Properties

» Specify the interface the component implements as a service, e.g.

```
<service>
   <provide interface=
      "pm.application.service.IActionContribution"/>
</service>
```

» Also properties can be defined declaratively, e.g.

```
<property name="action.name"
          type="String" value="SavePersonAction"/>
```

» The defined service will have the component's properties

Defining References

- DS uses dependency injection for references to other services
- The SCR resolves all the references automatically
  - Only, if all references with respect to the cardinality (see next slide) can be resolved, the component will be activated

```
<reference name="PersonManager"
   interface="pm.model.IPersonManager"
   ...
/>
```

Reference Cardinality

Cardinality	Meaning
0..1	Unary and optional
1..1	Unary and mandatory
0..n	Multiple and optional
1..n	Multiple and mandatory

Example

```
<reference name="PersonManager"
      ...
      cardinality="0..1"
/>
```

Accessing Services – Event Strategy

- On (un)binding a referenced service a (un)bind method is called
- Methods are found by reflection
- Method names are specified in XML

```
<reference name="PersonManager"
    ...
    bind="setPersonManager"
    unbind="removePersonManager"
    …
/>
```

- Parameters for bind/unbind method may be the service type or a ServiceReference, e.g.

```
public synchronized void
        setPersonManager(IPersonManager personManager)
```

Reference Policy

- How does unbinding of target services influence the life cycle?
- Static policy:
  - Service Component is deactivated and ...
  - ... activated again if other service(s) available
- Dynamic policy:
  - Life cycle is not affected, as long as references are satisfied

```
<reference name="PersonManager"
  …
  policy="dynamic"/>
```

DS Conclusion

+ Very easy to define components, services and references declaratively
+ SCR takes care of life cycle management and dependency resolution

But

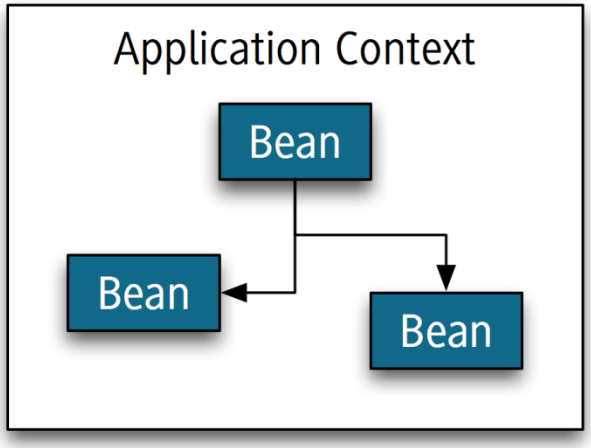
– Dependency injection only works for other services, not for POJOs or DS components
– Properties cannot be declared mandatory

Agenda

- Demo: A Swing App Framework based dynamic OSGi application
- OSGi Services
  - Providing and consuming Services
  - Service Tracker
- Overview Declarative Services & Spring Dynamic Modules
- Declarative Services in Detail
- Spring Dynamic Modules in Detail
- Comparison (DS vs. DM)
- Conclusion & Best Practices
- Discussion

Spring Dependency Injection

- Application Context: Lightweight container
- Beans: Non-invasive programming model
- Dependency Injection



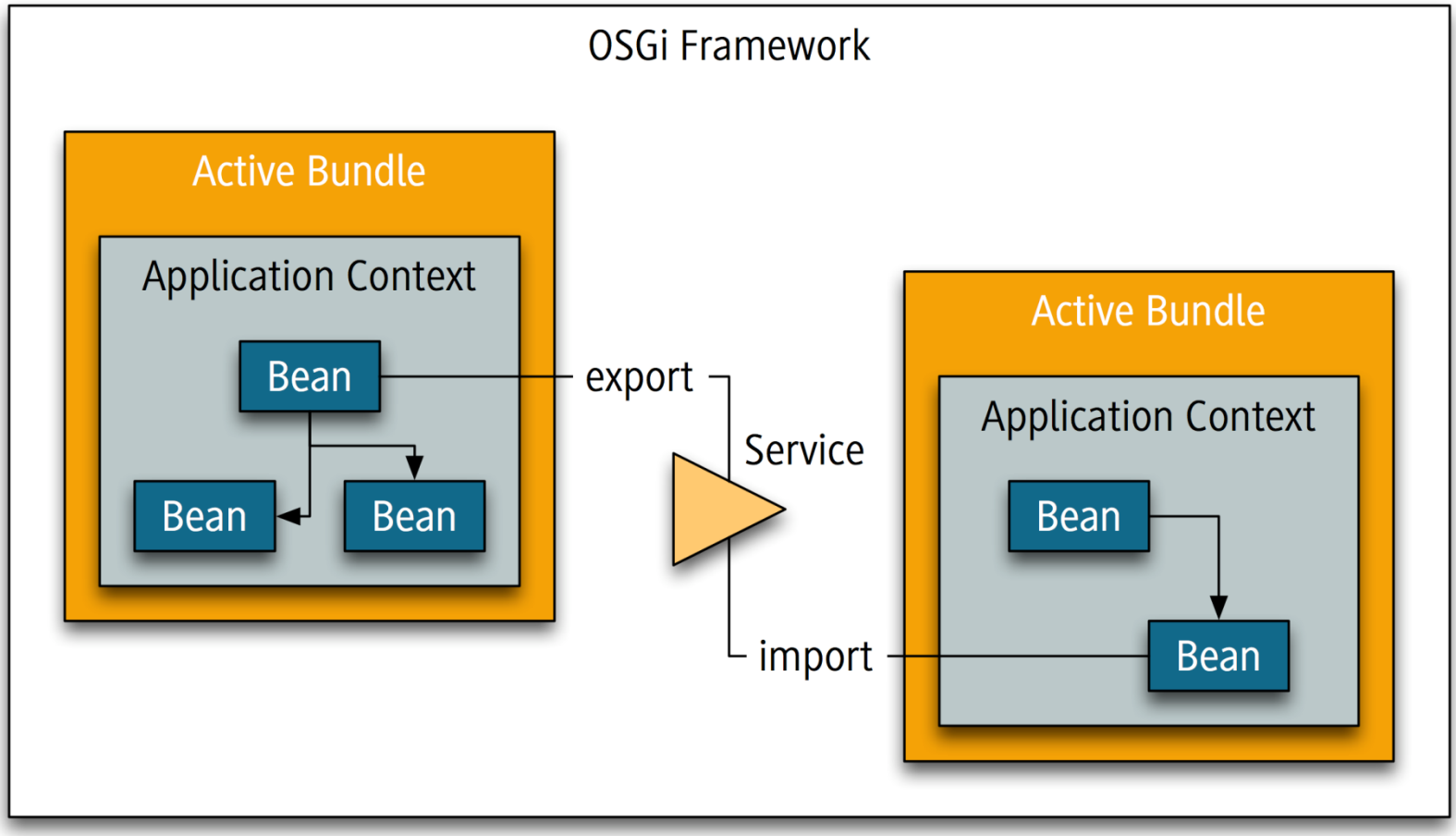
What is Spring DM?

Spring Dynamic Modules focuses on integrating Spring Framework powerful, non-invasive programming model and concepts with the dynamics and modularity of OSGi platform.

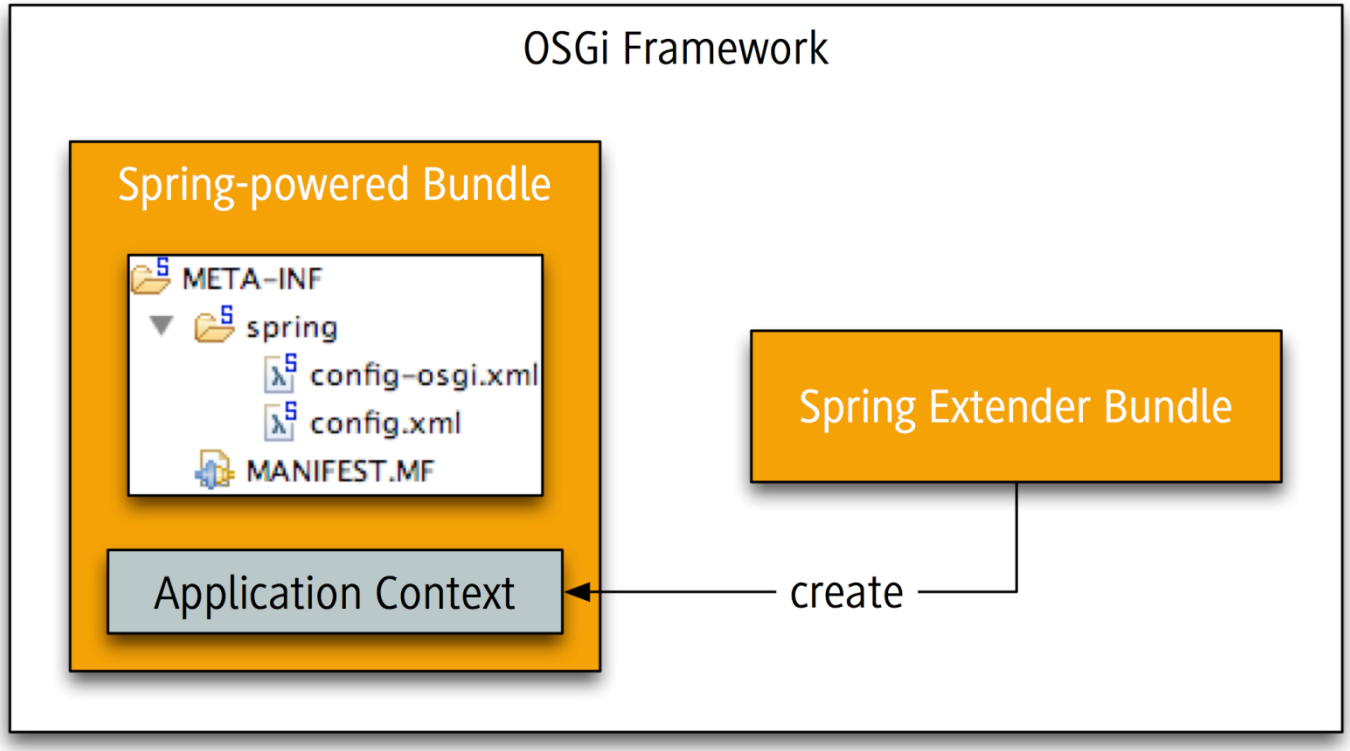
It allows transparent exporting and importing of OSGi services, life cycle management and control.

(Spring DM Ref. Guide, 1.0)

Spring DM at a Glance

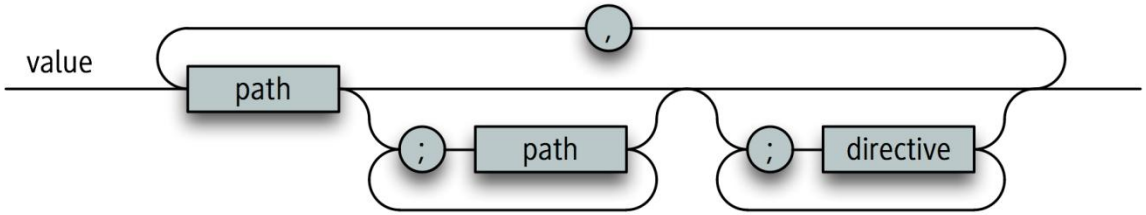


Applying the Extender Model



The Spring-Context Manifest Header

Spring-Context Manifest Header Syntax



Directives

- create-asynchronously
- wait-for-dependencies
- timeout
- publish-context

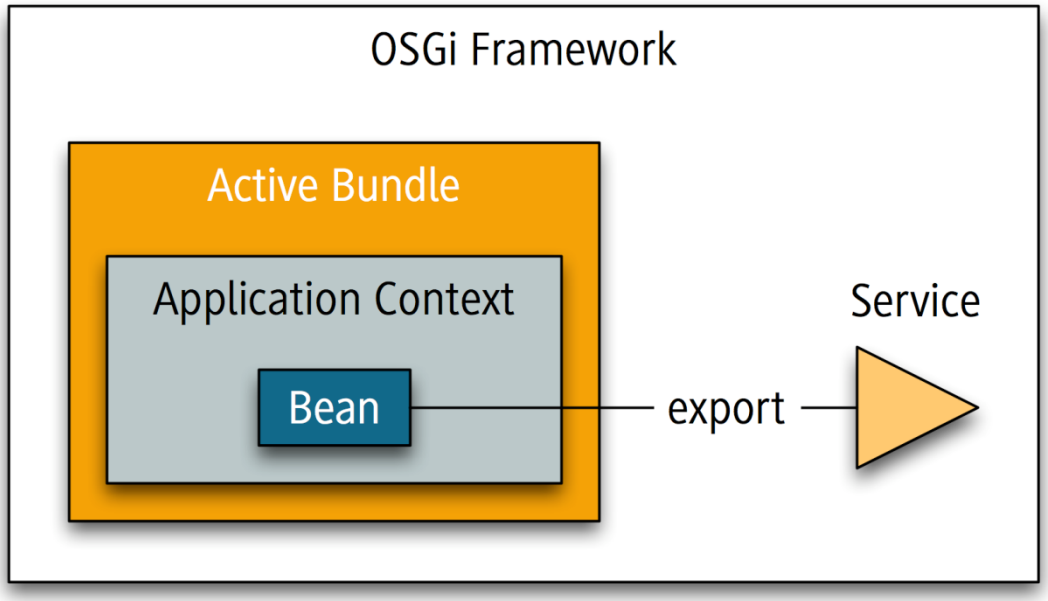
Examples

- Spring-Context: *
- Spring-Context: config.xml,config-osgi.xml
- Spring-Context: *;create-asynchronously:=false

Application Context Creation

Directive	Effect
create-asynchronously	Create asynchronously as to OSGi thread? Default is *true*
wait-for-dependencies	Wait for mandatory dependencies? Default is *true*
timeout	Time in seconds to wait for mandatory dependencies before failing. Default is 300
publish-context	Publish the Application Context as OSGi service? Default is *true*

Exporting Beans as OSGi Services (1)

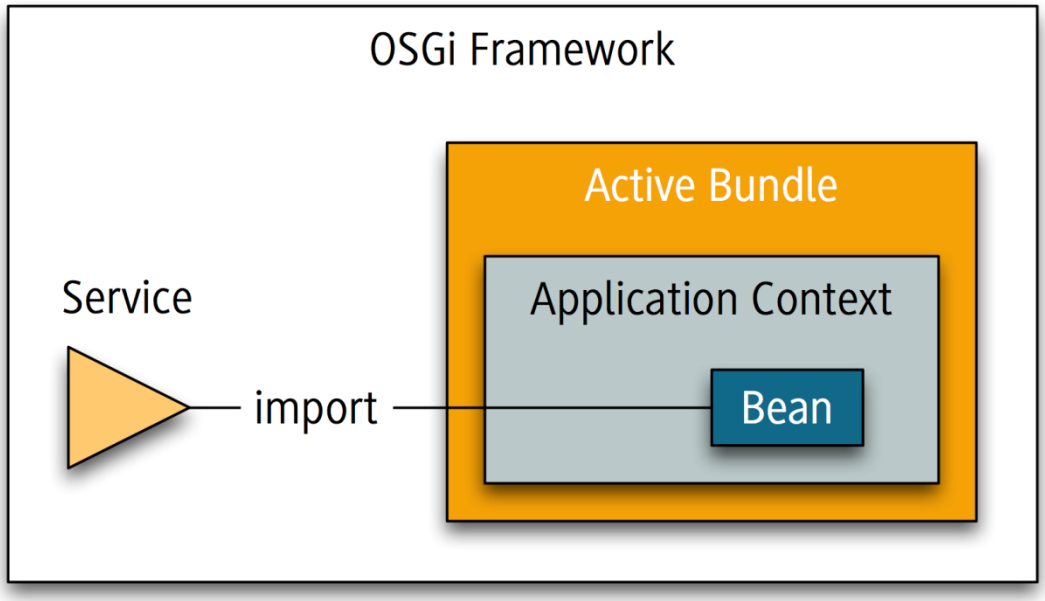



```
<service ref="..." interface="...">
```

Exporting Beans as OSGi Services (2)

service Attributes	Meaning
ref	Bean to be exported as OSGi service
interface	FQCN of service interface. Alternatives: *auto-export* or *interfaces* sub-element
auto-export	Automatically manage service interfaces. *disabled*, *interfaces*, *class-hierarchy*, *all-classes*
ranking	Service property *service.ranking*

Importing Beans as OSGi Services (1)




```
<reference id="..." interface="..." cardinality="...">

<set|list id="..." interface="..." cardinality="...">
```

Importing Beans as OSGi Services (2)

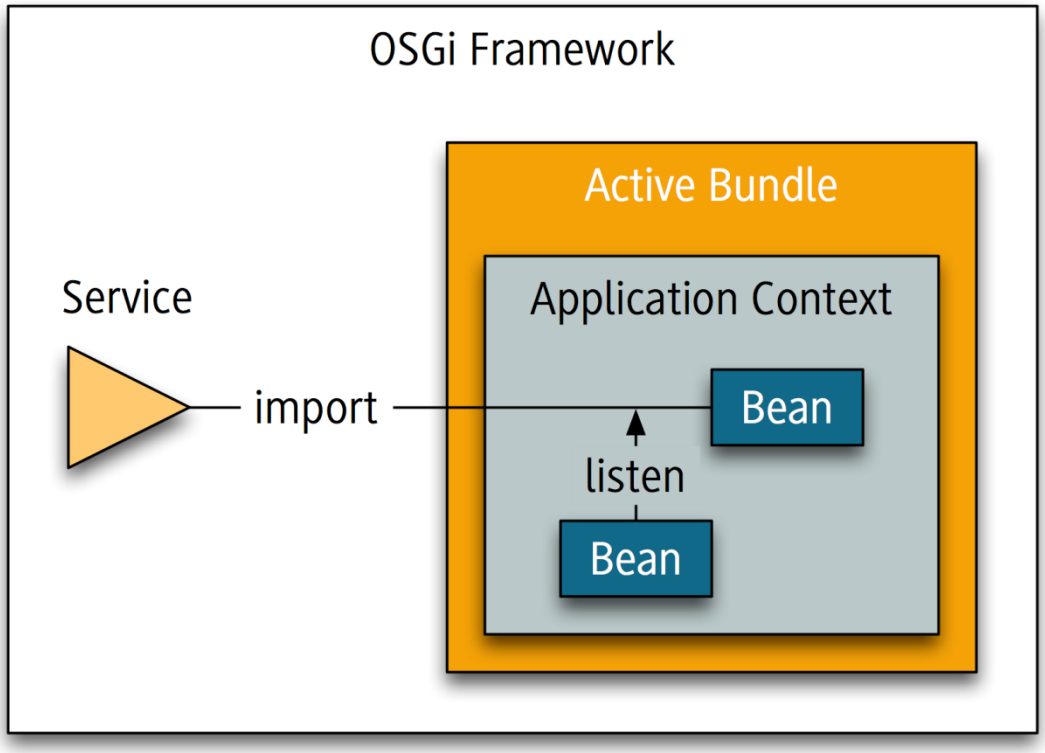
Attributes	Meaning
id	Name of bean for referenced OSGi service
interface	FQCN of service interface. Alternative: *interfaces* sub-element
filter	OSGi filter expression to constrain service look-up
cardinality	*reference*: 0..1, 1..1 *set*|*list*: 0..N, 1..N

And what about Service Dynamics?

- *ServiceUnavailableException* for invocations on stale references
- Unary references (*reference*) are dynamic-aware
  - Re-binding with matching service when bound service goes away
- Multiple references (*set*|*list*) are dynamic-aware
  - *Collection*s and *Iterator*s always up-to-date with service registry

- Services with unsatisfied references are unregistered …
- … and re-registered when satisfied again

Listening to Service Events




```
<reference id="..." interface="...">
  <listener bind-method="..." unbind-method="...">
    <beans:bean class="..."/>
    ...
```

Needed OSGi Bundles

To run Spring DM the following additional bundles are needed:

- spring-osgi-core-1.1.2.jar
- spring-osgi-extender-1.1.2.jar
- spring-osgi-io-1.1.2.jar
- com.springsource.org.aopalliance-1.0.0.jar
- spring-aop-2.5.5.jar
- spring-beans-2.5.5.jar
- spring-context-2.5.5.jar
- spring-core-2.5.5.jar
- *com.springsource.slf4j.api-1.5.0.jar*
- *com.springsource.slf4j.log4j-1.5.0.jar*
- *com.springsource.slf4j.org.apache.commons.logging-1.5.0.jar*
- *log4j.osgi-1.2.15-SNAPSHOT.jar*

Spring DM Conclusion

+ Very easy to define Spring beans, OSGi services and references declaratively
+ Spring DM takes care of life cycle management and dependency resolution
+ Very powerful Dependency Injection

But

– Properties also cannot be declared mandatory

Agenda

- Demo: A Swing App Framework based dynamic OSGi application
- OSGi Services
  - Providing and consuming Services
  - Service Tracker
- Overview Declarative Services & Spring Dynamic Modules
- Declarative Services in Detail
- Spring Dynamic Modules in Detail
- Comparison (DS vs. DM)
- Conclusion & Best Practices
- Discussion

DS versus DM

Topic	OSGi DS	Spring DM
Ease of use	++	+
Reducing complexity	+	++
Lazy service activation/creation	++	--
Compatible with procedural OSGi service model	++	++
Dependency Injection of Services	+	+
Dependency Injection of POJOs	--	++
Current tooling support	+	+

Conclusion & Recommendation

- We recommend to use either DS or DM rather than manually programming all the service infrastructure and life cycles
- If the capabilities of DS are ok with you (or lazy activation is important), use DS
- If you need DI or want to use Spring anyway, use Spring DM

Further Information

- OSGi Alliance Specifications: Service Compendium, 112 Declarative Services Specification
  - Component Description schema detailed
  - Enabling Service Components
  - Factory Components
- Spring Dynamic Modules
- Other Service Component models
  - Apache Felix iPOJO

Discussion