OSGi Alliance Community Event

iPOJO: The Simple Life

Richard S. Hall

Agenda

- Introduction & Background
- iPOJO Overview
- The Basics
- Providing Services
- Using Services
- Configuring Instances
- Creating Composites
- Conclusion

Introduction & Background

Introduction

- OSGi technology provides an interesting platform for creating ***dynamically extensible*** applications
- However, dealing with dynamism is a pain...
  - ...raising the abstraction is necessary
  - Developers should concentrate on application logic, not low-level OSGi mechanisms
- Solving these issues is necessary if we want to move into innovative areas
  - Context awareness, pervasive computing, autonomic computing, etc.

Background

2000

- I started implementing the OSGi specification so that I could investigate dynamic assembly of applications
  - The framework was called Oscar
  - Unfortunately, I have not stopped implementing it

Background

2001

- Humberto Cervantes and I started to discuss OSGi component models
  - Humberto finished a prototype of Beanome in 2002
    - Defined a simple OSGi-based component model with factories and simple service dependency management

Background

2002

- Humberto and I started working on a general approach to simplify OSGi dynamic service dependency management
  - Humberto developed Service Binder as part of his PhD
    - Included all of the standard mechanisms (e.g., optionality, aggregation, binding policies, etc.)

Background

2004

- Declarative Services was being defined, incorporating ideas from Service Binder
- Experimental versions of Service Binder introduced a composite service model
  - However, it proved insufficient

Background

2005

- I started thinking about an improved way to create composite services using byte code generation
- Peter Kriens starts thinking about using byte code manipulation to further simplify (***de-cruft***) dynamic dependency management
  - Peter gets into the details and presents a prototype in October 2005

Background

2006

- Around 2006, Clement Escoffier and I started working together to create iPOJO
  - Combine Peter's "***de-crufting***" effort with the ability to create dynamic composite services
  - Clement does all of the hard work, I just complain about it
- Which leads us to now...

iPOJO Overview

iPOJO Overview

- Approach
  - Make things simple
    - Simple things should be simple
    - Not so simple things should still be reasonably simple
    - Complicated things should be possible
  - Follow POJO philosophy
    - Use plain old Java objects
    - Avoid tying application logic to component framework
  - Employ byte code manipulation techniques
    - Intercept access to component member fields
    - Monitor component method entry, exit, and exceptions
  - Be as lazy as possible

iPOJO Overview

- Usage
  - Part of the build process, specifically packaging
    - First, compile component source
    - Then, package component as bundle (perhaps using **BND**)
    - Finally, process bundle with iPOJO manipulator
  - Supports Maven, Ant, and Eclipse
    - Example Ant task definition:

```
<target name=”post-package” depends=”package”>
  <ipojo input="${output.dir}/${project.name}.jar"
         metadata="metadata.xml"/>
</target>
```

iPOJO Overview

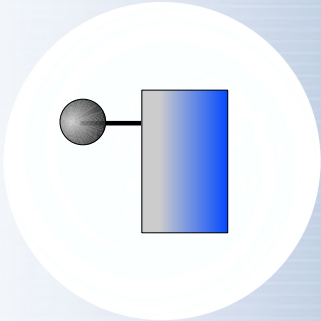
- Issues
  - POJO-ness
    - iPOJO does not support constructor injection
      - Possible, but it is anathema to dynamism
    - iPOJO requires an empty constructor or a constructor with bundle context argument
      - Service dependencies are usable in the constructor
      - Possible to have constructor with other arguments, but iPOJO will not use it
    - iPOJO supports annotations
      - The examples in this presentation use annotations
      - However, annotations are still a form of API and reduce POJO-ness
      - iPOJO supports other metadata annotation approaches

The Basics

The Basics

- Assume we have the following service interface

```
public interface Printer {
  void print(String s);
}
```



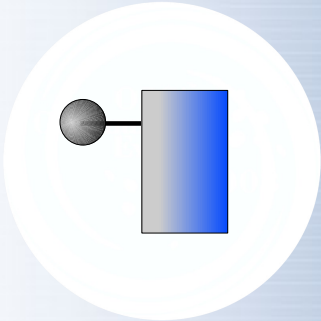
The Basics

- Assume we have the following service interface

```
public interface Printer {
  void print(String s);
}
```

- Here is an iPOJO component providing the service

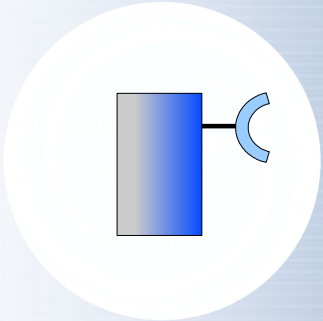
```
@Component
@Provides
public class PrinterImpl implements Printer {
  public void print(String s) {
    // Print the string somehow...
  }
}
```



The Basics

- Assume we define a text editor like this

```
public interface TextEditor {
  public void print();
  public String getText();
  public void setText(String s);
  public int getSelectionStart();
  public int getSelectionEnd();
}
```



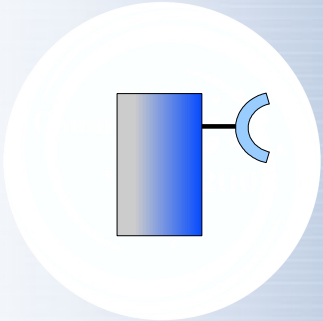
The Basics

- Assume we define a text editor like this

```
public interface TextEditor {
  public void print();
  public String getText();
  public void setText(String s);
  public int getSelectionStart();
  public int getSelectionEnd();
}
```

- Text editor implementation with service dependency

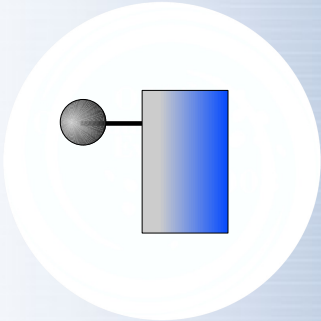
```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires
  private Printer m_printer;

  public void print() {
    m_printer.print(getText());
  }
  ...
}
```



Providing Services

Providing Services

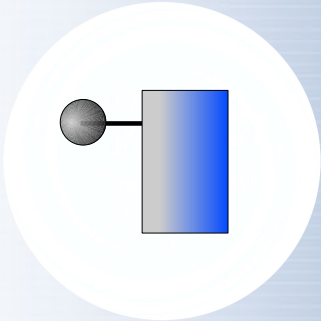
- Service providers can define dynamic service properties to reflect run-time state changes



Providing Services

- Service providers can define dynamic service properties to reflect run-time state changes
  - e.g., indicate printer status

```
@Component
@Provides
public class PrinterImpl implements Printer {
  @ServiceProperty(value=true)
  private boolean ready;
  public void deviceCallback(boolean status) {
    ready = status;
  }
  public void print(String s) {
    if (ready) {
        // print the string
    }
  }
}
```

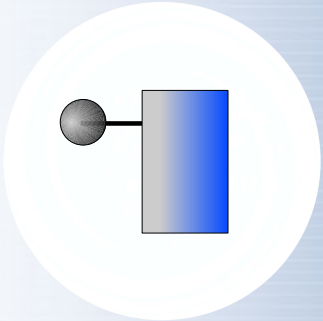


Providing Services

- Service providers can define dynamic service properties to reflect run-time state changes
  - e.g., indicate printer status

```
@Component
@Provides
public class PrinterImpl implements Printer {
  @ServiceProperty(value=true)
  private boolean ready;
  public void deviceCallback(boolean status) {
    ready = status;
  }
  public vo        int(String s) {
    if (ready
    }
  }
}
```

Changes to the associated variable are automatically reflected in the registered service's properties at run time.



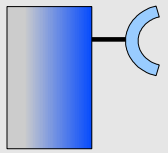
Providing Services

- Service providers can define dynamic service properties to reflect run-time state changes
  - e.g., indicate printer status

```
@Component
@Provides
public class PrinterImpl implements Printer {
  @ServiceProperty(value=true)
  private boolean ready;
  public void devic llback(boolean status) {
    ready = statu
  }
  pub
    i
    }
  }
}
```

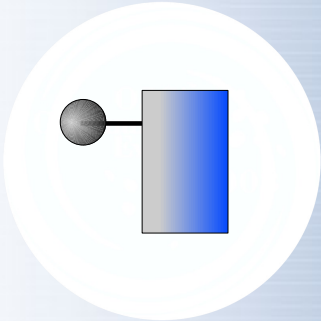
Client code can filter on this:

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(filter="(ready=true)")
  private Printer m_printer;
  ...
}
```



Providing Services

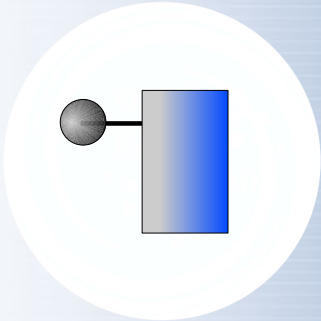
- Service providers can participate in service life cycle
  - Necessary to indicate unsatisfied requirements that cannot be managed by container



Providing Services

- Service providers can participate in service life cycle
  - Necessary to indicate unsatisfied requirements that cannot be managed by container
    - e.g., broken Bluetooth connection

```
@Component
@Provides
public class PrinterImpl implements Printer {
  @Controller
  private boolean m_valid = true;
  ...
  public void connectionCallback(boolean status) {
    m_valid = status;
  }
  ...
}
```

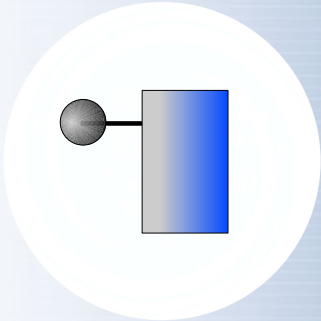


Providing Services

- Service providers can participate in service life cycle
  - Necessary to indicate unsatisfied requirements that cannot be managed by container
    - e.g., broken Bluetooth connection

```
@Component
@Provides
public class PrinterImpl implements Printer {
  @Controller
  private boolean m_valid = true;
  ...
  public void connectionCallback(boolean status) {
    m_valid = status;
  }
}
```

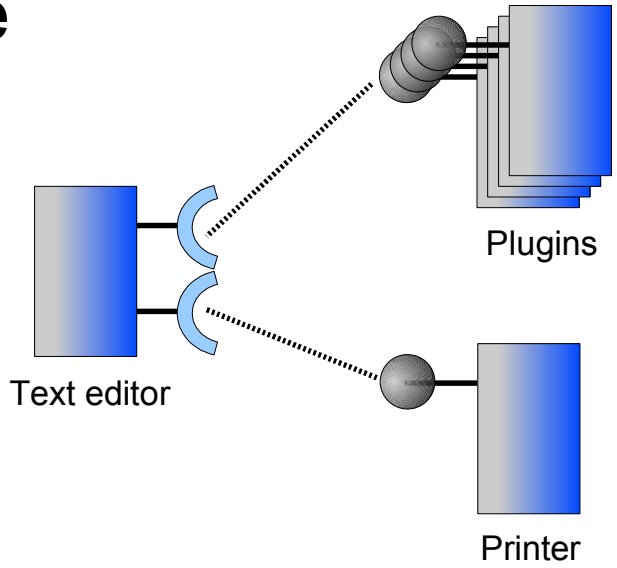
The component programmatically controls whether its service is valid, in addition to any other service dependencies already managed by the container.

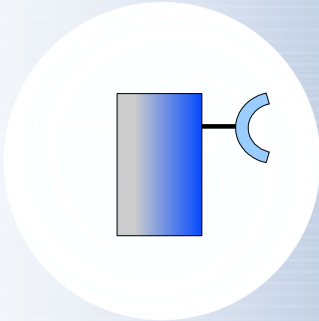


Using Services

Using Services

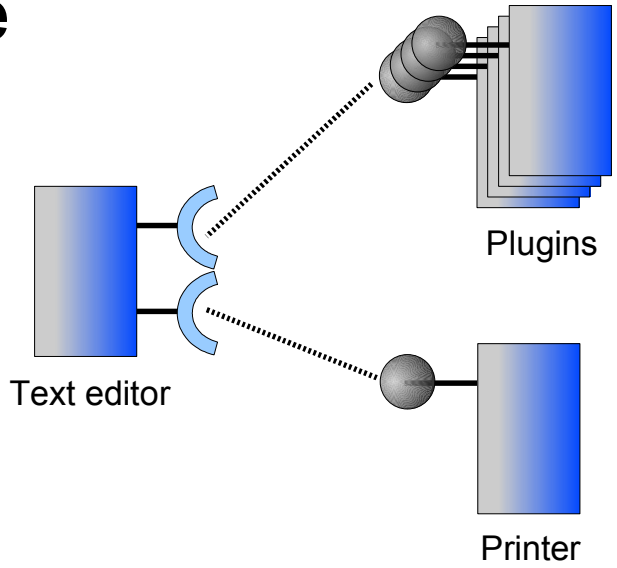
- Let's make our text editor extensible with plugins, something like





Using Services

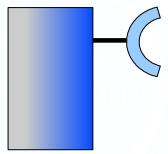
- Let's make our text editor extensible with plugins, something like



- We define a simple plugin interface, like

```
public interface Plugin {
  public String getName();
  public void execute(TextEditor te);
}
```

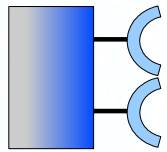
  - What about our text editor implementation?



Using Services

- We add a dependency to our text editor for plugins

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires
  private Printer m_printer;
  @Requires
  private Plugin[] m_plugins;

  public void print() {
    m_printer.print(getText());
  }

  public void listPlugins() {
    for (int i = 0; i < m_plugins.length; i++) {
      System.out.println(m_plugins[i].getName();
    }
  }
  ...
}
```

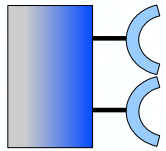


Using Services

- We add a dependency to our text editor for plugins

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public vo
    m_print
  }

  public void listPlugins() {
    for (int i = 0; i < m_plugins.length; i++) {
      System.out.println(m_plugins[i].getName();
    }
  }
  ...
}
```

Although ***cardinality*** is assumed depending on the field type, we probably also want to specify that the printer and plugins are ***optional***.

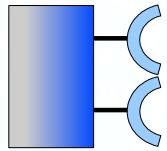


Using Services

- We add a dependency to our text editor for plugins

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    m_printer.print(getText());
  }

  public v
    for (i
      Syst
    }
  }
  ...
}
```

By default, the container uses the ***Null Object*** pattern, so components do not need to check for null on optional services.

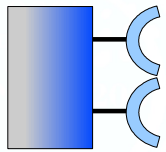


Using Services

- We add a dependency to our text editor for plugins

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    m_printer.print(getText());
  }

  public void listPlugins() {
    for (int i = 0; i < m_plugins.length; i++) {
      System.out.println(m_plugins[i].getName();
    }
  }
  ...
}
```

However, it is also possible to actually get a null.

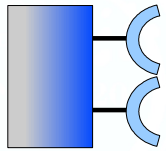


Using Services

- We add a dependency to our text editor for plugins

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optic
  private Plugin[]
  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  public void listPlugins() {
    for (int i = 0; i < m_plugins.length; i++) {
      System.out.println(m_plugins[i].getName());
    }
  }
  ...
}
```

But now we will have to check for null...

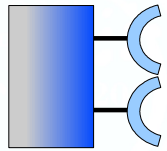


Using Services

- We add a dependency to our text editor for plugins

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  public void listPlugins() {
    for (int i = 0; i < m_plugins.length; i++) {
      System.out.println(m_plugins[i].getName();
    }
  }
  ...
}
```

This raises some concurrency issues, but keep those in mind for later...

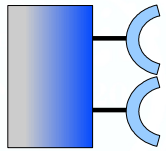


Using Services

- We add a dependency to our text editor for plugins

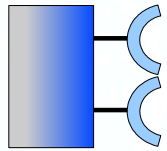
```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true, \
            default-implementation=FilePrinter.class)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[

  public void print() {
    m_printer.print(getText());
  }

  public void listPlugins() {
    for (int i = 0; i < m_plugins.length; i++) {
      System.out.println(m_plugins[i].getName();
    }
  }
  ...
}
```

Instead, we could specify a default implementation.



Using Services

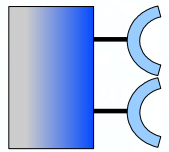
- Service dependencies have a binding policy
  - ***Static*** – snapshot, any change impacts life cycle
  - ***Dynamic*** – default, tracks run-time changes
  - ***Dynamic priority*** – tracks priority run-time changes



Using Services

- Service dependencies have a binding policy
  - ***Static*** – snapshot, any change impacts life cycle
  - ***Dynamic*** – default, tracks run-time changes
  - ***Dynamic priority*** – tracks priority run-time changes

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,policy="dynamic-priority")
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    m_printer.print(getText());
  }
  ...
}
```

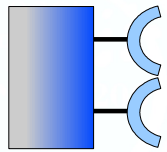


Using Services

- Service dependencies have a binding policy
  - ***Static*** – snapshot, any change impacts life cycle
  - ***Dynamic*** – default, tracks run-time changes
  - ***Dynamic priority*** – tracks priority run-time changes

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,policy="dynamic-priority")
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print()
    m_printer.print(getText());
  }
  ...
}
```

OSGi service ranking is default priority...

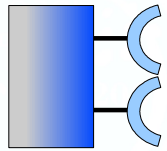


Using Services

- Service dependencies have a binding policy
  - ***Static*** – snapshot, any change impacts life cycle
  - ***Dynamic*** – default, tracks run-time changes
  - ***Dynamic priority*** – tracks priority run-time changes

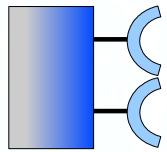
```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,policy="dynamic-priority", \
            comparator=NearestPrinter.class)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    m_printer.print(getText());
  }
  ...
}
```

But a custom sort order can be specified.



Using Services

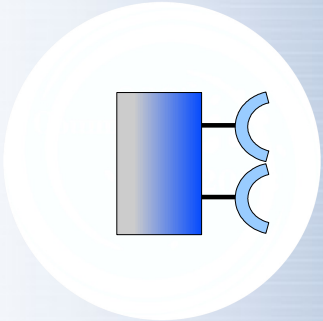
- Temporal dependencies associate service usage with method-level access
  - For a dependency that is only required at a given time
    - e.g., services that are only used at initialization



Using Services

- Temporal dependencies associate service usage with method-level access
  - For a dependency that is only required at a given time
    - e.g., services that are only used at initialization

```
@Component
public class TextEditorImpl implements TextEditor {
  @Temporal
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    m_printer.print(getText());
  }
  ...
}
```



Using Services

- Temporal dependencies associate service usage with method-level access
  - For a dependency that is only required at a given time
    - e.g., services that are only used at initialization

```
@Component
public class TextEditorImpl implements TextEditor {
  @Temporal
  private Printer m_printer;
  @Requires(optional=t
  private Plugin[] m_p

  public void p
    m_printer.p
  }
  ...
}
```

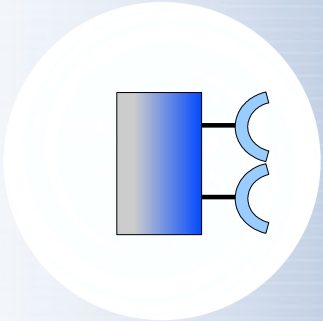
If the service is not available, it will ***wait*** and eventually ***timeout*** and ***throw an exception***. By default, all methods accessing the service are managed, but they can be specified.

Using Services

- Temporal dependencies associate service usage with method-level access
  - For a dependency that is only required at a given time
    - e.g., services that are only used at initialization

```
@Component
public class TextEditorImpl implements TextEditor {
  @Temporal(timeout=5000)
  private Printer  printer;
  @Requires(optio      ue)
  private P

  public vo
    m_printer.print(getText());
  }
  ...
}
```

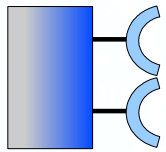
Can specify the timeout value or disable the timeout altogether.



Using Services

- Callback methods can be associated with managed service dependencies

```
@Component
public class TextEditorImpl implements TextEditor {
  ...
  @Requires(id="plugins",optional=true)
  private Plugin[] m_plugins;

  @Bind(id="plugins")
  private void addPlugin() {
    // Update menus
  }

  @Unbind(id="plugins")
  private void removePlugin() {
    // Update menus
  }
  ...
}
```

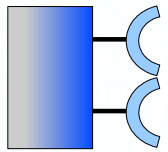


Using Services

- Callback methods can be associated with managed service dependencies

```
@Component
public class TextEditorImpl implements TextEditor {
  ...
  @Requires(id="plugins",optional=true)
  private Plugin[] m_plugins;

  @Bind(id="plugins")
  private void addPlugin() {
    // Update menus
  }

  @Unbind(id="plugins")
  private void removePlugin() {
    // Update menus
  }
  ...
}
```

Binding methods will be called whenever the managed array of plugins changes.

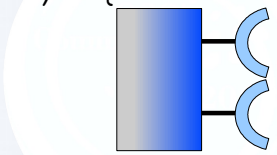


Using Services

- Callback methods can be associated with managed service dependencies

```
@Component
public class TextEditorImpl implements TextEditor {
  ...
  @Requires(id="plugins",optional=true)
  private Plugin[] m_plugins;

  @Bind(id="plugins")
  private void addPlugin(ServiceReference ref) {
    // Update menus
  }

  @Unbind(id="plugins")
  private void removePlugin(ServiceReference ref) {
    // Update menus
  }
  ...
}
```

Binding methods can also receive service reference, if necessary.

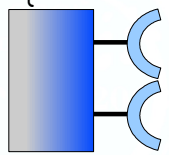


Using Services

- Callback methods can be associated with managed service dependencies

```
@Component
public class TextEditorImpl implements TextEditor {
  ...
  // Manage plugins ourselves
  private Plugin[] m_plugins;

  @Bind(id="plugins",aggregate=true,optional=true)
  private synchronized void addPlugin(Plugin p) {
    // Update menus
  }

  @Unbind(id="plugins")
  private synchronized void removePlugin(Plugin p) {
    // Update menus
  }
  ...
}
```

Also possible to handle service management ourselves, but this definitely requires ***synchronization***.

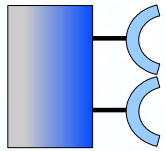


Using Services

- Callback methods can be associated with managed service dependencies

```
@Component
public class TextEditorImpl implements TextEditor {
  ...
  // Manage plugins ourselves
  private Plugin[] m_plugins;

  @Bind(id="plugins",aggregate=true,optional=true)
  private synchronized void addPlugin(
    Plugin p, ServiceReference ref) {
    // Update menus
  }

  @Unbind(id="plugins")
  private synchronized void removePlugin(
    Plugin p, ServiceReference ref) {
    // Update menus
  }
  ...
}
```

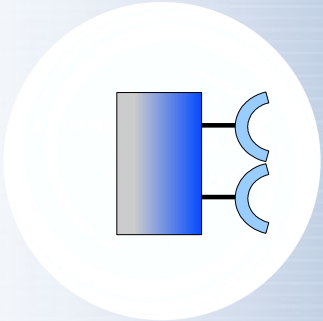
Still possible to get the service reference in this case.



Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

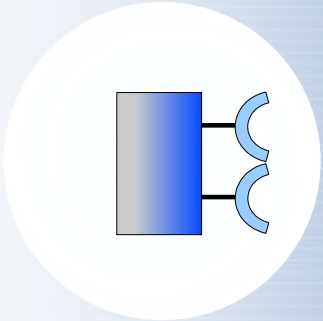


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

Is this check-then-act action valid?

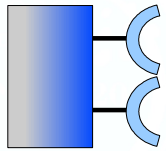


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

Thread local service reference copies are cached upon access until the thread exits to ensure a consistent view of services...

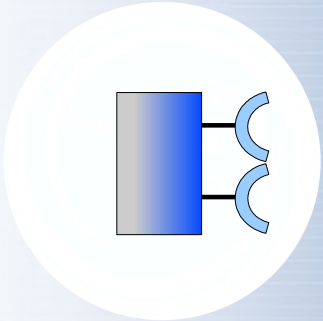


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

Thread enters here.

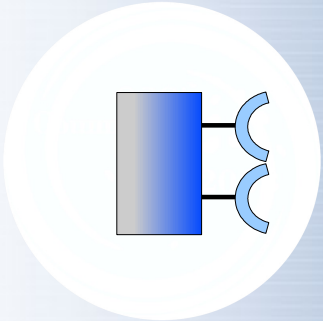


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

First use caches reference.

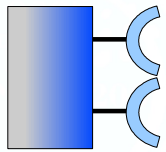


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

Cached reference reused here.

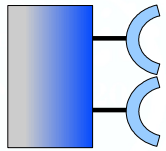


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

Cached reference released here.

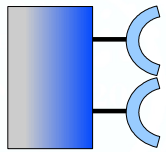


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

Cached service references work for aggregate dependencies and across invocations if thread calls out from a method and re-enters.

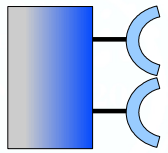


Using Services

- Recall previous concurrency issues...

```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires(optional=true,nullable=false)
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  public void print() {
    if (m_printer != null) {
      m_printer.print(getText());
    }
  }
  ...
}
```

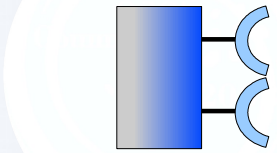
Components still need to guard their own shared state to be thread safe.



Using Services

- Components can receive life-cycle callbacks
  - i.e., be notified when their dependencies are valid

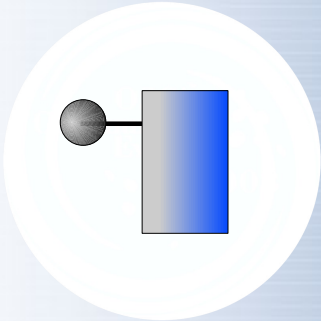
```
@Component
public class TextEditorImpl implements TextEditor {
  @Requires
  private Printer m_printer;
  @Requires(optional=true)
  private Plugin[] m_plugins;

  @Validate
  private void valid() { /* do something */ }

  @Invalidate
  private void invalid() { /* do something */ }

  ...
}
```



Configuring Instances

Configuring Components

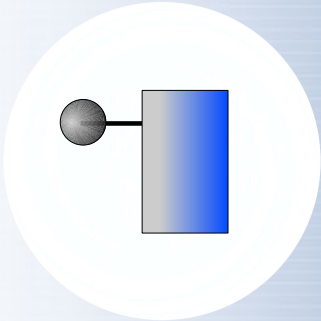
- Component instances can be configured...
  - ...but first, how are instances created?



Configuring Components

- Component instances can be configured...
  - ...but first, how are instances created?
    - Recall our simple printer service

```
@Component
@Provides
public class PrinterImpl implements Printer {
  public void print(String s) {
    // print the string
  }
}
```

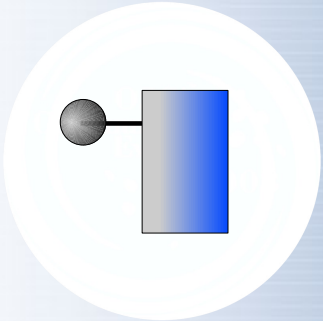


Configuring Components

- Component instances can be configured...
  - ...but first, how are instances created?
    - Recall our simple printer service

```
@Component
@Provides
public class PrinterImpl implements Printer {
  public void print(String s) {
    // print the string
  }
}
```

This actually defines a component type, for which iPOJO automatically registers a factory service.



Configuring Components

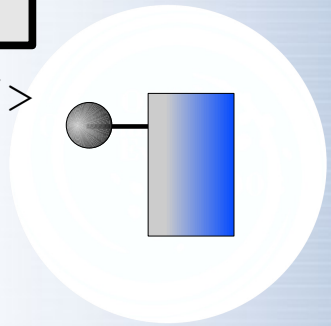
- Component instances can be configured...
  - ...but first, how are instances created?
    - Recall our simple printer service

```
@Component
@Provides
public class PrinterImpl implements Printer {
  public void print(String s) {
    // print the string
  }
}
```

Instances are created by declaring them in a metadata file...

metadata.xml

```
<ipojo>
  <instance component="org.foo.PrinterImpl"/>
</ipojo>
```



Configuring Components

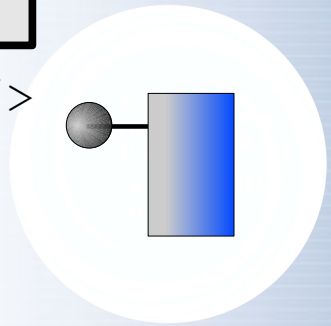
- Component instances can be configured...
  - ...but first, how are instances created?
    - Recall our simple printer service

```
@Component
@Provides
public class PrinterImpl implements Printer {
  public void print(String s) {
    // print the string
  }
}
```

Need not be in the same bundle; thus, you can deploy your types and deploy your instances separately.

metadata.xml

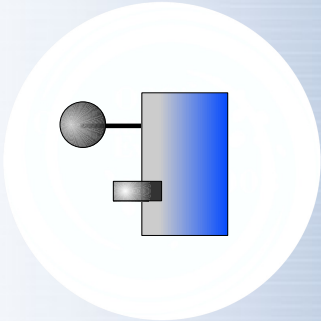
```
<ipojo>
  <instance component="org.foo.PrinterImpl"/>
</ipojo>
```



Configuring Components

- Created instances can be configured directly
  - First, modify our service to have configuration properties

```
public interface Printer {
  void setProperties(Dictionary config);
  void print(String s);
}

@Component
@Provides
public class PrinterImpl implements Printer {
  @Property(name="printer.config")
  private Dictionary m_config;
  public void setProperties(Map config) {
    m_config = config;
  }
  public void print(String s) {
    // Print the string somehow...
  }
}
```

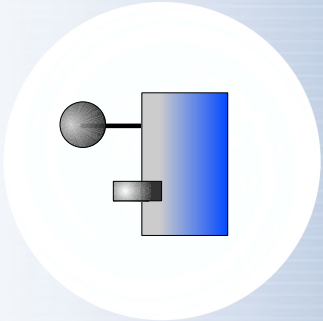


Configuring Components

- Created instances can be configured directly
  - First, modify our service to have configuration properties

```
public interface Printer {
  void setProperties(Dictionary config);
  void print(String s);
}

@Component
@Provides
public class PrinterImpl implements Printer {
  @Property(name="printer.config")
  private Dictionary m_config;
  public void setProperties(Map config) {
    m_config = config;
  }
  public void print(String s) {
    // Print the string somehow...
  }
}
```

We've modified our interface to accept a property dictionary, although this is not necessary...

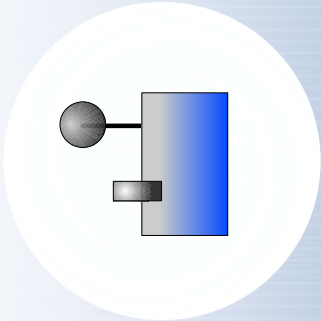


Configuring Components

- Created instances can be configured directly
  - First, modify our service to have configuration properties

```
public interface Printer {
  void setProperties(Dictionary config);
  void print(String s);
}

@Component
@Provides
public class PrinterImpl implements Printer {
  @Property(name="printer.config")
  private Dictionary m_config;
  public void setProperties(Map config) {
    m_config = config;
  }
  public void print(String s) {
    // Print the string somehow...
  }
}
```

And we've specified a field in our implementation to be injected with the configuration properties.

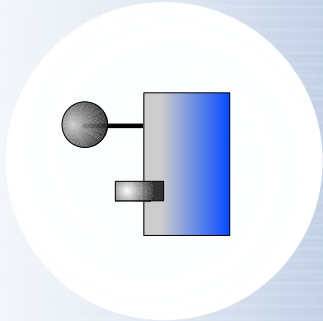


Configuring Components

- Now instances can be declared and directly configured in the metadata file

metadata.xml

```
<ipojo>
  <instance component="org.foo.PrinterImpl">
    <property name="printer.config">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```



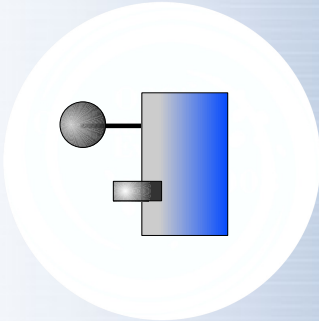
Configuring Components

- Now instances can be declared and directly configured in the metadata file

metadata.xml

```
<ipojo>
  <instance component="org.foo.PrinterImpl">
    <property name="printer.config">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```

While scalar properties are supported, this example assembles properties into a dictionary for injection into the associated field.

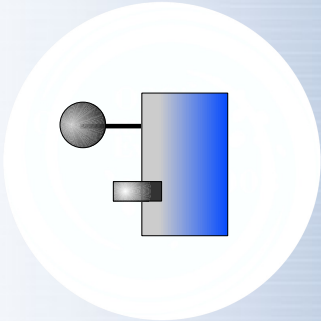


Configuring Components

- Managed services from Configuration Admin are supported...

metadata.xml

```
<ipojo>
  <instance component="org.foo.PrinterImpl">
    <property name="managed.service.pid" value="foo"/>
    <property name="printer.config">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```



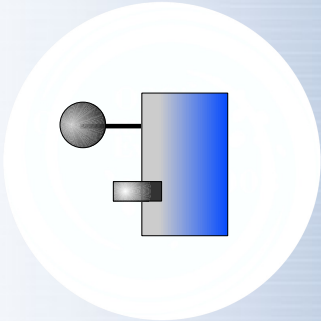
Configuring Components

- Managed services from Configuration Admin are supported...

Component instance will be dynamically injected with named configuration from Configuration Admin.

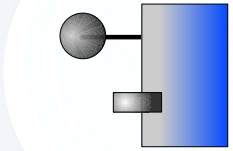
metadata.xml

```
<ipojo>
  <instance component="org.foo.PrinterImpl">
    <property name="managed.service.pid" value="foo"/>
    <property name="printer.config">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```



Configuring Components

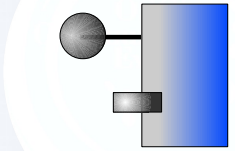
- Managed service factories from Configuration Admin are also supported...
  - In this case, component factory services are assigned a managed service factory PID
    - Default PID is the component class name
  - Simply add configurations to Configuration Admin associated with the appropriate factory PID



Configuring Components

- Managed service factories from Configuration Admin are also supported...
  - In this case, component factory services are assigned a managed service factory PID
    - Default PID is the component class name
  - Simply add configurations to Configuration Admin associated with the appropriate factory PID

For both managed services and managed factories, configuration changes are properly tracked and injected at run time.



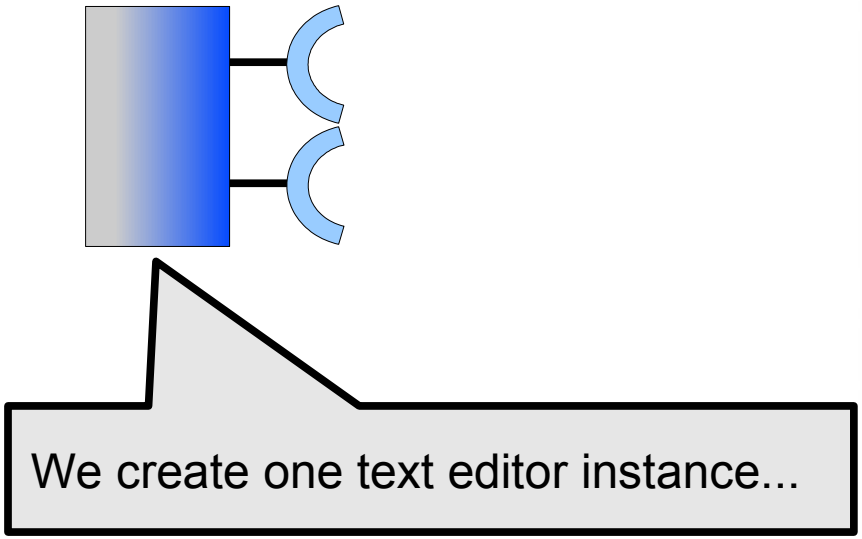
Creating Composites

Creating Composites

- Now we have all the pieces in place, let's create a text editor

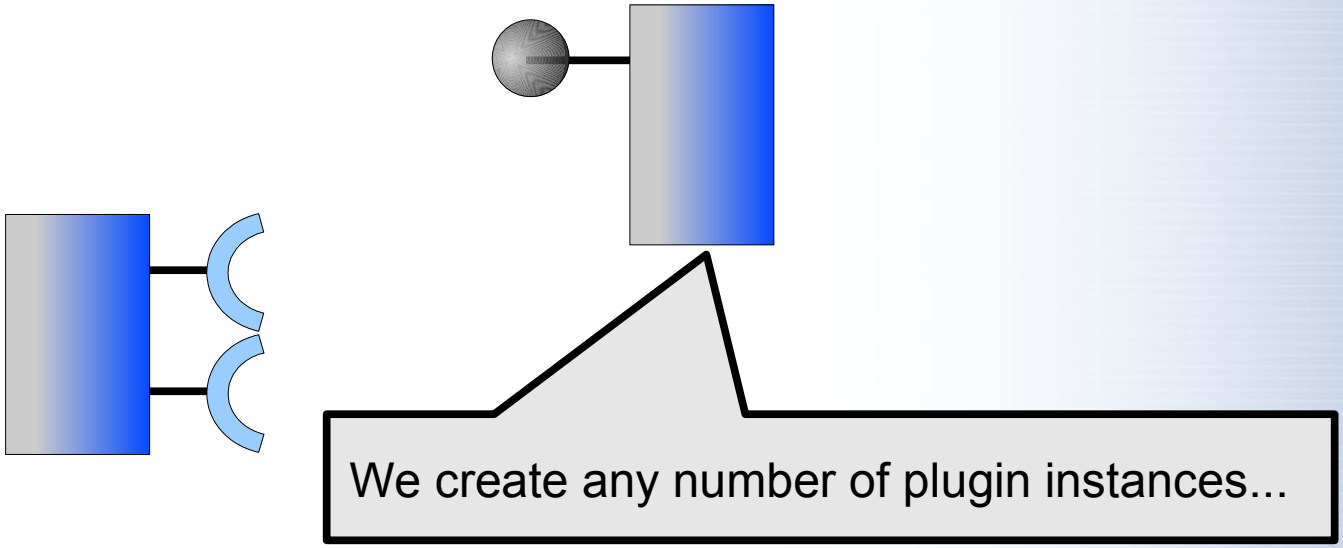
Creating Composites

- Now we have all the pieces in place, let's create a text editor



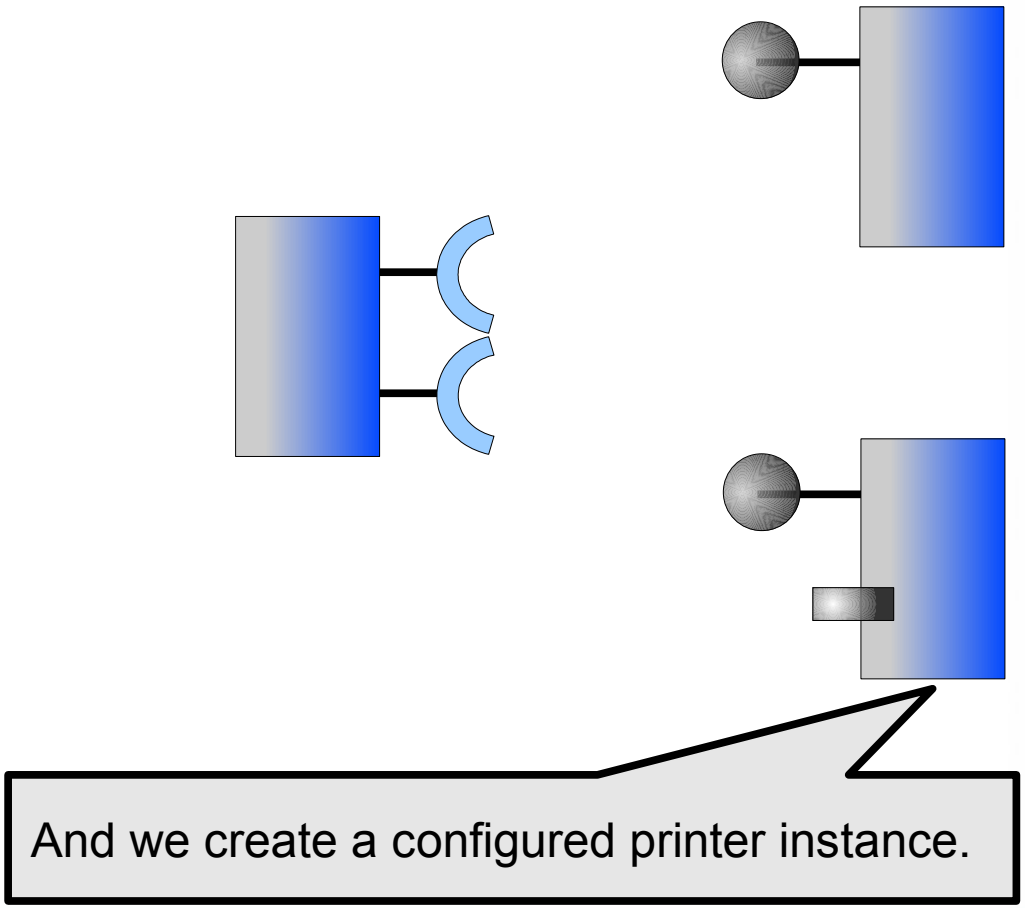
Creating Composites

- Now we have all the pieces in place, let's create a text editor



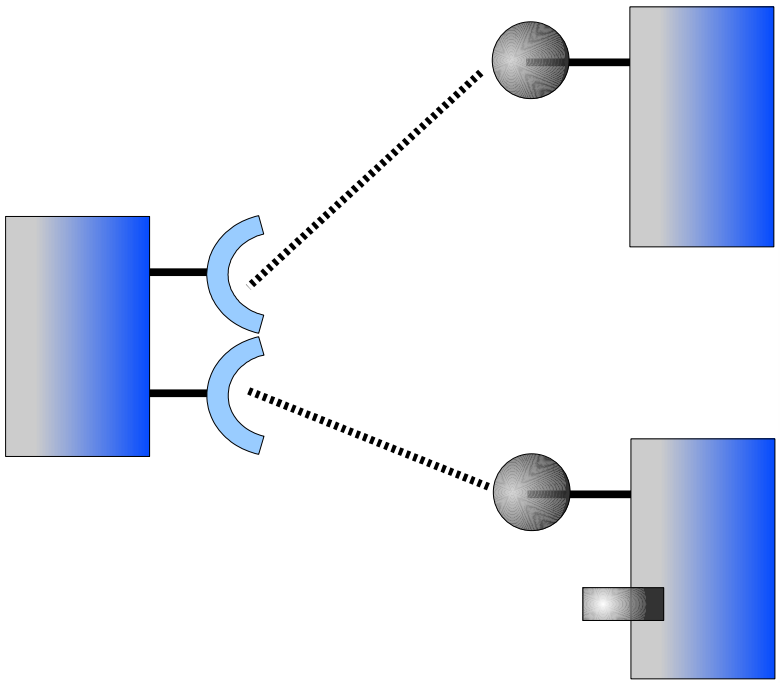
Creating Composites

- Now we have all the pieces in place, let's create a text editor



Creating Composites

- Now we have all the pieces in place, let's create a text editor



iPOJO correctly binds them and manages their dynamic availability at run time...

Creating Composites

- This configuration might look something like this...

metadata.xml

```
<ipojo>
  <instance component="org.bar.TextEditorImpl"/>
  <instance component="org.woz.SpellCheckPlugin"/>
  <instance component="org.foo.PrinterImpl">
    <property name="printer.configuration">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```

Creating Composites

- This configuration might look something like this...

metadata.xml

```
<ipojo>
  <instance component="org.bar.TextEditorImpl"/>
  <instance component="org.woz.SpellCheckPlugin"/>
  <instance component="org.foo.PrinterImpl">
    <property name="printer.configuration">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```

All resulting services are published into the OSGi service registry...*hmm*...

Creating Composites

- This configuration might look something like this...

metadata.xml

```
<ipojo>
  <instance component="org.bar.TextEditorImpl"/>
  <instance component="org.woz.SpellCheckPlugin"/>
  <instance component="org.foo.PrinterImpl">
    <property name="printer.configuration">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```

Recall our configurable printer service definition:

```
public interface Printer {
  void setProperties(Map config);
  void print(String s);
}
```

What happens if someone else decides to change our printer configuration?

Creating Composites

- This configuration might look something like this...

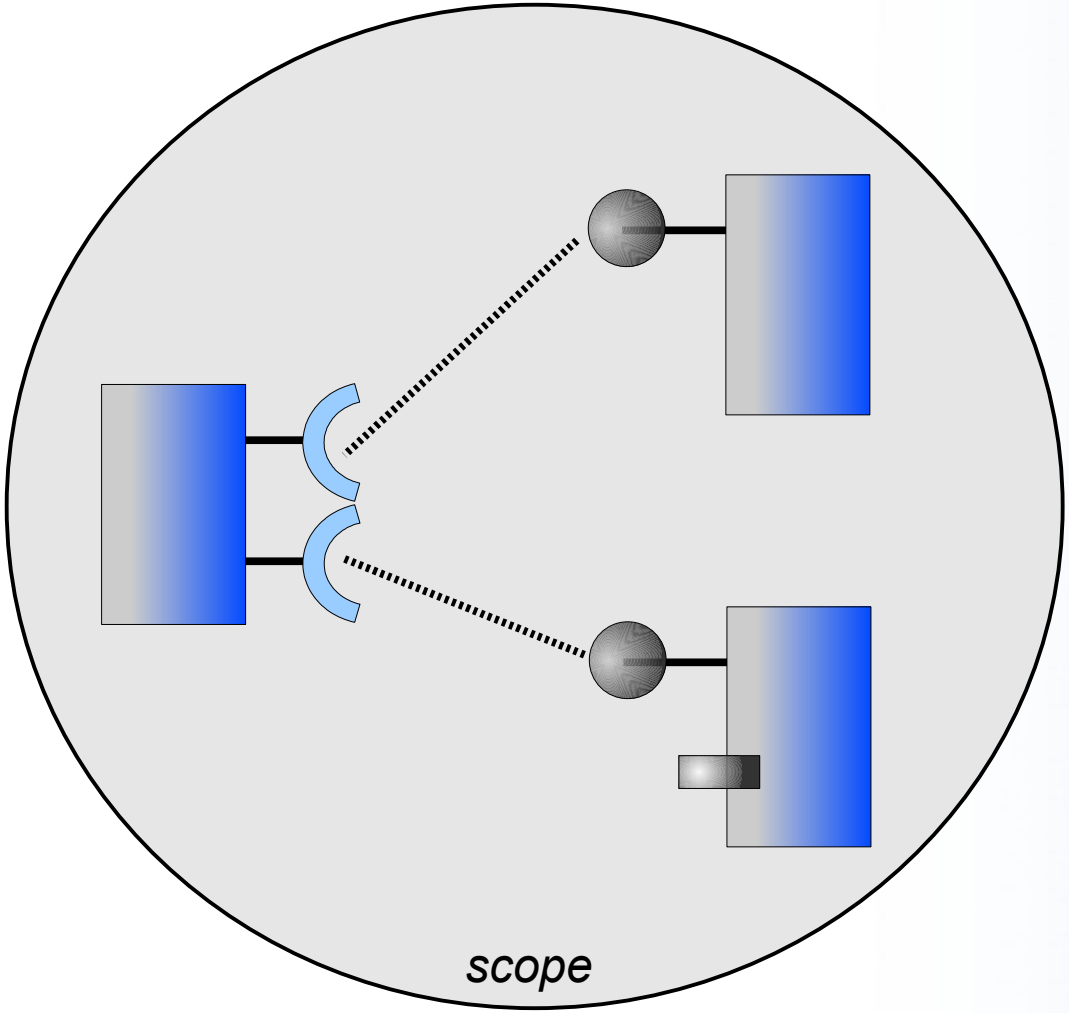
metadata.xml

```
<ipojo>
  <instance component="org.bar.TextEditorImpl"/>
  <instance component="org.woz.SpellCheckPlugin"/>
  <instance component="org.foo.PrinterImpl">
    <property name="printer.configuration">
      <property name="duplex" value="true"/>
      <property name="orientation" value="landscape"/>
    </property>
  </instance>
</ipojo>
```

Services are global, so anyone can use them and/or change them. ***Bummer!***

Creating Composites

- We really want something like...

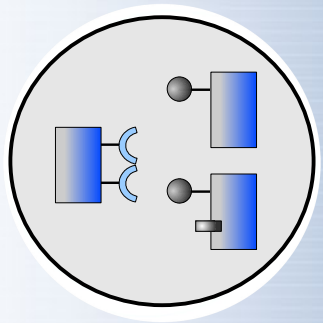


Creating Composites

- iPOJO composites do just that...

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <instance component="org.woz.SpellCheckPlugin"/>
    <instance component="org.foo.PrinterImpl">
      <property name="printer.configuration">
        <property name="duplex" value="true"/>
        <property name="orientation" value="landscape"/>
      </property>
    </instance>
  </composite>
</ipojo>
```



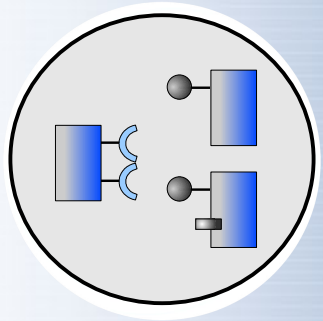
Creating Composites

- iPOJO composites do just that...

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <instance component="org.woz.SpellCheckPlugin"/>
    <instance component="org.foo.PrinterImpl">
      <property name="printer.configuration">
        <property name="duplex" value="true"/>
        <property name="orientation" value="landscape"/>
      </property>
    </instance>
  </composite>
</ipojo>
```

Composites declare contained instances and create a scope for them, without any changes to the components themselves.

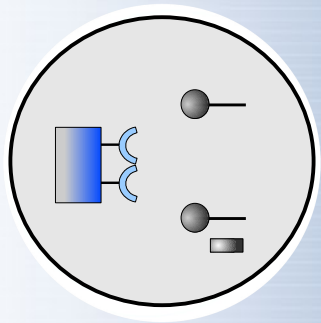


Creating Composites

- Perhaps we don't want specific implementations...

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"/>
    <sub-service action="instantiate"
      specification="org.foo.Printer">
      <property name="printer.configuration">
        <property name="duplex" value="true"/>
        <property name="orientation" value="landscape"/>
      </property>
    </sub-service>
  </composite>
</ipojo>
```



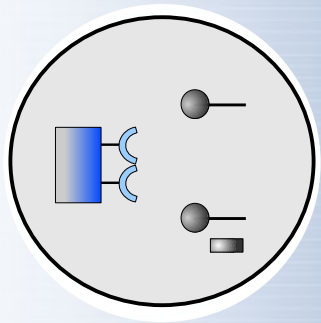
Creating Composites

- Perhaps we don't want specific implementations...

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"/>
    <sub-service action="instantiate"
      specification="org.foo.Printer">
      <property name="prin      configuration">
        <property name
        <property name                    cape"/>
      </property>
    </sub-service>
  </composite>
</ipojo>
```

Composite sub-services specify only the desired service interface, leaving iPOJO to inject an available implementation at run time.

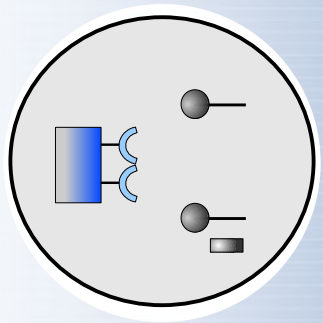


Creating Composites

- Composites also support context-bound properties
  - Context sources are a dictionary of run-time properties

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="global:user-context-source"
      filter="(location=${user.location})">
      <property name="printer.configuration">
        <property name="duplex" value="true"/>
        <property name="orientation" value="landscape"/>
      </property>
    </sub-service>
    <sub-service action="instantiate"
      specification="org.bar.Plugin"
      aggregate="true" optional="true"/>
  </composite>
</ipojo>
```



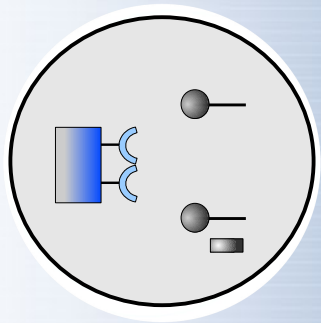
Creating Composites

- Composites also support context-bound properties
  - Context sources are a dictionary of run-time properties

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="global:user-context-source"
      filter="(location=${user.loc                    })">
      <property name="printe
        <property name="dupl
        <property name="orie
      </property>
    </sub-service>
    <sub-service action="instantiate"
      specification="org.bar.Plugin"
      aggregate="true" optional="true"/>
  </composite>
</ipojo>
```

A sub-service can specify the identifier of a global or local context source, such as in this example for user location context...



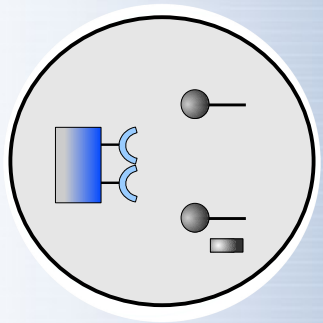
Creating Composites

- Composites also support context-bound properties
  - Context sources are a dictionary of run-time properties

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="global:user-context-source"
      filter="(location=${user.location})">
      <property name="printer.co          uration">
        <property name="dup
        <property name="ori
      </property>
    </sub-service>
    <sub-service action="instantiate"
      specification="org.bar.Plugin"
      aggregate="true" optional="true"/>
  </composite>
</ipojo>
```

The sub-service filter can reference contextual properties that will be resolved to values at run time...



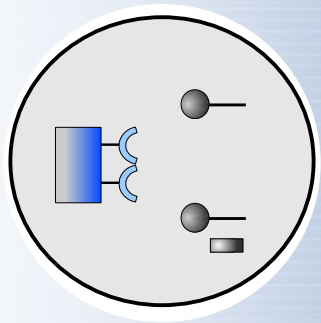
Creating Composites

- Composites also support context-bound properties
  - Context sources are a dictionary of run-time properties

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="global:user-context-source"
      filter="(location=${user.location})">
      <property name="printer.co[illegible]uration">
        <property name="dup[illegible]
        <property name="ori[illegible]
      </property>
    </sub-service>
    <sub-service action="instantiate"
      specification="org.bar.Plugin"
      aggregate="true" optional="true"/>
  </composite>
</ipojo>
```

Such as in this example, where the printer service is filtered by the current location of the user.

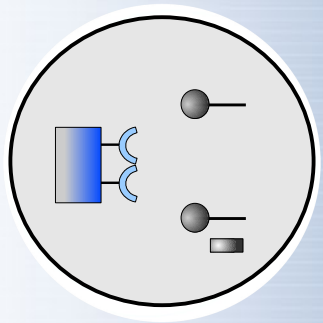


Creating Composites

- Components can also be context sources

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="global:user-context-source"
      filter="(location=${user.location})">
      <property name="printer.configuration">
        <property name="duplex" value="true"/>
        <property name="orientation" value="landscape"/>
      </property>
    </sub-service>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="(mime.type=${mime.type})"/>
  </composite>
</ipojo>
```



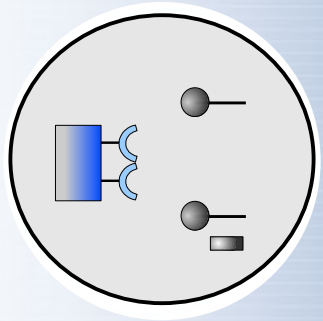
Creating Composites

- Components can also be context sources

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="globa
      filter="(location=${u
      <property name="print
        <property name="dup
        <property name="orientation" value="landscape"/>
      </property>
    </sub-service>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="(mime.type=${mime.type})"/>
  </composite>
</ipojo>
```

For example, the text editor component implements a context source that publishes its current document MIME type...



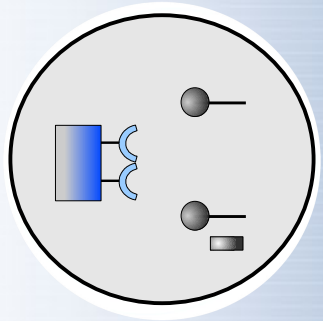
Creating Composites

- Components can also be context sources

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="instantiate" specification="org.foo.Printer"
      context-source="global:user-context-source"
      filter="(location=${user.location})">
      <property name="printer.configuration">
        <property name="duplex" value="true"/>
        <property name="orienta
      </property>
    </sub-service>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="(mime.type=${mime.type})"/>
  </composite>
</ipojo>
```

Thus, we can dynamically track plugins that are relevant to the current MIME type.

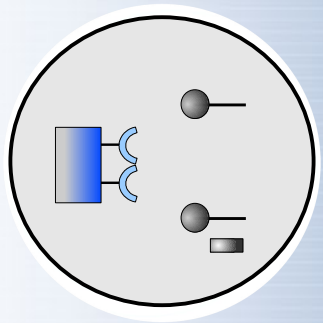


Creating Composites

- Finally, composites are actually normal components
  - i.e., they have factories and can be instantiated

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <provides action="implement"
     specification="org.foo.TextEditor"/>
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="import"
     specification="org.foo.Printer"/>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="(mime.type=${mime.type})"/>
  </composite>
</ipojo>
```



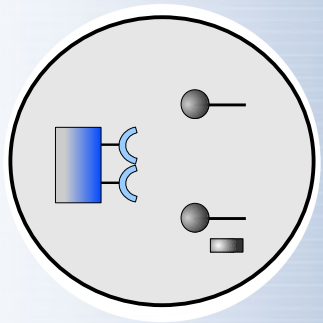
Creating Composites

- Finally, composites are actually normal components
  - i.e., they have factories and can be instantiated

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <provides action="implement"
     specification="org.bar.TextEditor"/>
    <instance component="o  bar.TextEditorImpl"/>
    <sub-service actio
     specification="org
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="(mime.type=${mime.type})"/>
  </composite>
</ipojo>
```

They can provide services.



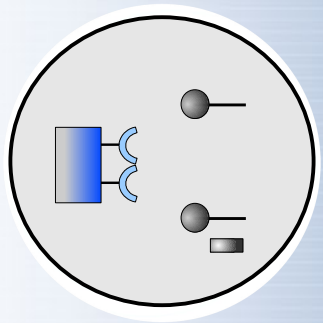
Creating Composites

- Finally, composites are actually normal components
  - i.e., they have factories and can be instantiated

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <provides action="implement"
     specification="org.bar.TextEditor"/>
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="import"
     specification="org.foo.Printer"/>
    <sub-service action="instantiate"
     specification="org
      aggregate="true"
      filter="(mime.type=${mime.type})"/>
  </composite>
</ipojo>
```

They can require services.



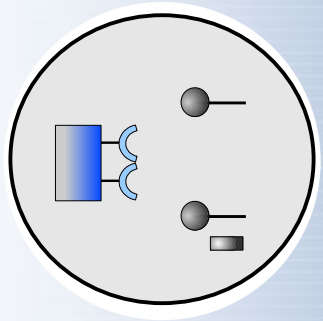
Creating Composites

- Finally, composites are actually normal components
  - i.e., they have factories and can be instantiated

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <provides action="implement"
     specification="org.bar.TextEditor"/>
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="import"
     specification="org.foo.Printer"/>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="(mime.t
  </composite>
</ipojo>
```

They can contain instances of components providing services.



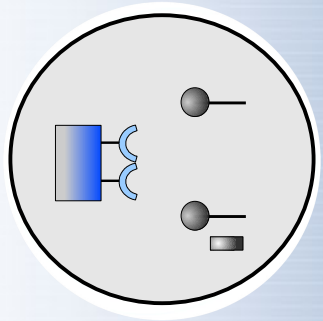
Creating Composites

- Finally, composites are actually normal components
  - i.e., they have factories and can be instantiated

metadata.xml

```
<ipojo>
  <composite name="ExtensibleTextEditor">
    <provides action="implement"
     specification="org.bar.TextEditor"/>
    <instance component="org.bar.TextEditorImpl"/>
    <sub-service action="import"
     specification="org.foo.Printer"/>
    <sub-service action="instantiate"
     specification="org.bar.Plugin"
      aggregate="true" optional="true"
      filter="
  </composite>
</ipojo>
```

They can be instantiated into other composites, creating a fully hierarchical component model.



Conclusion

Conclusion

- The OSGi framework is a great platform for dynamically extensible applications
  - However, there is a price to pay in added complexity
- iPOJO aims to mitigate complexity
  - Simplify the programming model without sacrificing the underlying power
  - Provide an advanced, fully hierarchical component model for dynamically extensible applications

Questions?