Felix Connect

Karl Pauls

ApacheCon Europe 2012

Slides together with Richard S. Hall

$ whoami

- Karl Pauls
  - Member Apache Software Foundation
    - Felix, ACE, Incubator: Celix
  - Co-author of „OSGi in Action“

karlpauls@gmail.com

Outline

- Motivating µServices
  - Procedures
  - Objects
  - Interfaces
  - Factories
  - Dependency injection
- Service orientation
- PojoSR
- Felix Connect

Motivating µServices

Motivating µServices

	Data encapsulation/abstraction
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Procedures

```
byte[4096] canvas;

void mouseClickCallback(int x, int y) {
  drawCircle(x, y, 100);
}

void drawCircle(int x, int y, int r) {
  ...
  // draws into canvas
  ...
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Procedures

```
byte[4096] canvas;

void mouseClickCallback(int x, int y) {
  drawCircle(x, y, 100);
}

void drawCircle(int x, int y, int r) {
  ...
  // draws into canvas
  ...
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Procedures

```
byte[4096] canvas;

void mouseClickCallback(int x, int y) {
  drawCircle(x, y, 100);
}

void drawCircle(int x, int y, int r) {
  ...
  // draws into canvas
  ...
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Procedures

```
byte[4096] canvas;

void mouseClickCallback(int x, int y) {
  drawCircle(x, y, 100);
}

void drawCircle(int x, int y, int r) {
  ...
  // draws into canvas
  ...
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Procedures

```
byte[4096] canvas;

void mouseClickCallback(int x, int y) {
  drawCircle(x, y, 100);
}

void drawCircle(int x, int y, int r) {
  ...
  // draws into canvas
  ...
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Objects

```
public abstract class Shape {
  public void draw(Canvas c);
}
public class Paint {
  private Canvas;
  private Shape;
  public Paint(Canvas canvas, Shape shape) { … }

  public void mouseClick(int x, int y) {
    shape.draw(canvas, x, y);
  }
  public static void main() {
    new Paint(
      new Canvas(),
      new Circle(100));
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Objects

```
public abstract class Shape {
  public void draw(Canvas c);
}
public class Paint {
  private Canvas;
  private Shape;
  public Paint(Canvas canvas, Shape shape) { … }

  public void mouseClick(int x, int y) {
    shape.draw(canvas, x, y);
  }
  public static void main() {
    new Paint(
      new Canvas(),
      new Circle(100));
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Objects

```
public abstract class Shape {
  public void draw(Canvas c);
}
public class Paint {
  private Canvas;
  private Shape;
  public Paint(Canvas canvas, Shape shape) { … }

  public void mouseClick(int x, int y) {
    shape.draw(canvas, x, y);
  }
  public static void main() {
    new Paint(
      new Canvas(),
      new Circle(100));
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Objects

```
public abstract class Shape {
  public void draw(Canvas c);
}
public class Paint {
  private Canvas;
  private Shape;
  public Paint(Canvas canvas, Shape shape) { … }

  public void mouseClick(int x, int y) {
    shape.draw(canvas, x, y);
  }
  public static void main() {
    new Paint(
      new Canvas(),
      new Circle(100));
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Objects

```
public abstract class Shape {
  public void draw(Canvas c);
}
public class Paint {
  private Canvas;
  private Shape;
  public Paint(Canvas canvas, Shape shape) { … }

  public void mouseClick(int x, int y) {
    shape.draw(canvas, x, y);
  }
  public static void main() {
    new Paint(
      new Canvas(),
      new Circle(100));
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Interfaces

```
public interface Shape {
  public void draw(Canvas c);
}
public class Paint {
  private Canvas;
  private Shape;
  public Paint(Canvas canvas, Shape shape) { … }

  public void mouseClick(int x, int y) {
    shape.draw(canvas, x, y);
  }
  public static void main() {
    new Paint(
      new Canvas(),
      new Circle(100));
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Interfaces

```
public interface Shape {
   public void draw(Canvas c);
}
public class Paint {
   private Canvas;
   private Shape;
   public Paint(Canvas canvas, Shape shape) { … }

   public void mouseClick(int x, int y) {
      shape.draw(canvas, x, y);
   }
   public static void main() {
      new Paint(
         new Canvas(),
         new Circle(100));
   }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Interfaces

```
public interface Shape {
   public void draw(Canvas c);
}
public class Paint {
   private Canvas;
   private Shape;
   public Paint(Canvas canvas, Shape shape) { … }

   public void mouseClick(int x, int y) {
      shape.draw(canvas, x, y);
   }
   public static void main() {
      new Paint(
         new Canvas(),
         new Circle(100));
   }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Factories

```
public class Paint {
   …
   public static void main() {
      new Paint(
         new Canvas(),
         ShapeFactory.createShape());
   }
}
public class ShapeFactory {
   public static Shape createShape() {
      return Class.forName(
           System.getProperty("shapefactory.shapeimpl"));
   }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Factories

```
public class Paint {
   …
   public static void main() {
      new Paint(
         new Canvas(),
         ShapeFactory.createShape());
   }
}
public class ShapeFactory {
   public static Shape createShape() {
      return Class.forName(
           System.getProperty("shapefactory.shapeimpl"));
   }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Factories

```
public class Paint {
   …
   public static void main() {
      new Paint(
         new Canvas(),
         ShapeFactory.createShape());
   }
}
public class ShapeFactory {
   public static Shape createShape() {
      return Class.forName(
           System.getProperty("shapefactory.shapeimpl"));
   }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Factories

```
public class Paint {
   …
   public static void main() {
      new Paint(
         new Canvas(),
         ShapeFactory.createShape());
   }
}
public class ShapeFactory {
   public static Shape createShape() {
      return Class.forName(
           System.getProperty("shapefactory.shapeimpl"));
   }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Dependency injection

```
public class Paint implements ShapeConsumer {
  @Inject
  public Paint(Shape shape) { … }
}
public class ShapeModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Shape.class).to(Circle.class);
    bind(ShapeConsumer.class).to(Paint.class);
  }
}

Injector injector = Guice.createInjector(new ShapeModule());
ShapeConsumer consumer = injector.getInstance(ShapeConsumer.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Dependency injection

```
public class Paint implements ShapeConsumer {
  @Inject
  public Paint(Shape shape) { … }
}
public class ShapeModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Shape.class).to(Circle.class);
    bind(ShapeConsumer.class).to(Paint.class);
  }
}

Injector injector = Guice.createInjector(new ShapeModule());
ShapeConsumer consumer = injector.getInstance(ShapeConsumer.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Dependency injection

```
public class Paint implements ShapeConsumer {
  @Inject
  public Paint(Shape shape) { … }
}
public class ShapeModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Shape.class).to(Circle.class);
    bind(ShapeConsumer.class).to(Paint.class);
  }
}

Injector injector = Guice.createInjector(new ShapeModule());
ShapeConsumer consumer = injector.getInstance(ShapeConsumer.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Dependency injection

```
public class Paint implements ShapeConsumer {
  @Inject
  public Paint(Shape shape) { … }
}
public class ShapeModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Shape.class).to(Circle.class);
    bind(ShapeConsumer.class).to(Paint.class);
  }
}

Injector injector = Guice.createInjector(new ShapeModule());
ShapeConsumer consumer = injector.getInstance(ShapeConsumer.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Context and Dependency Injection (CDI)

```
public class Paint implements ShapeConsumer {
  @Inject
  public Shape shape;
}
@Default
public class CircleProducer {
  @Produces
  protected Shape createShape() {
    return new CircleImpl();
  }
}

Paint paint = beanContainer.getBeanByType(Paint.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Context and Dependency Injection (CDI)

```
public class Paint implements ShapeConsumer {
  @Inject
  public Shape shape;
}
@Default
public class CircleProducer {
  @Produces
  protected Shape createShape() {
    return new CircleImpl();
  }
}

Paint paint = beanContainer.getBeanByType(Paint.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Context and Dependency Injection (CDI)

```
public class Paint implements ShapeConsumer {
  @Inject
  public Shape shape;
}
@Default
public class CircleProducer {
  @Produces
  protected Shape createShape() {
    return new CircleImpl();
  }
}

Paint paint = beanContainer.getBeanByType(Paint.class);
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

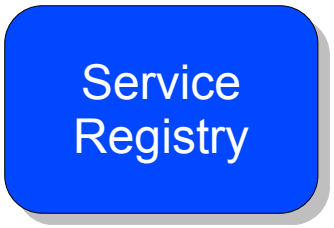
Service Orientation

Service orientation

- Promoting a service-oriented interaction pattern

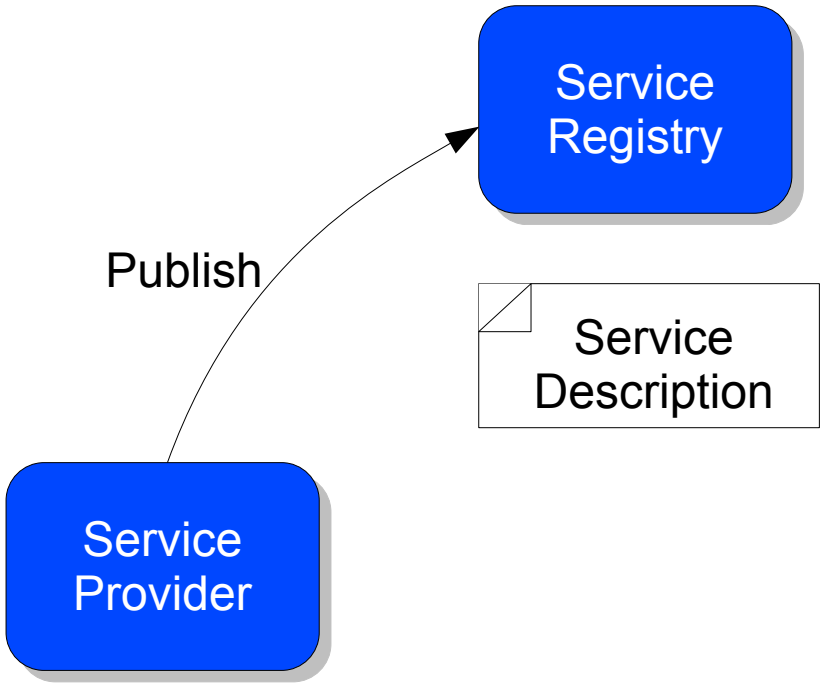
Service orientation

- Promoting a service-oriented interaction pattern



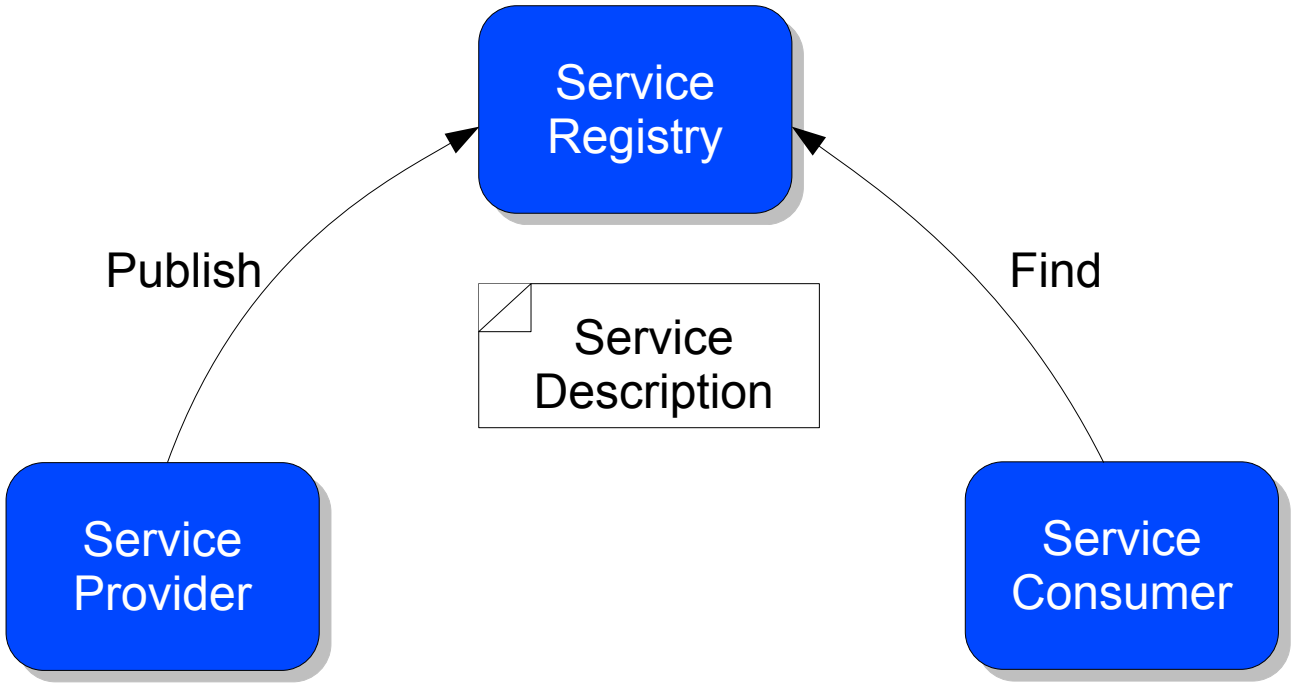
Service orientation

- Promoting a service-oriented interaction pattern



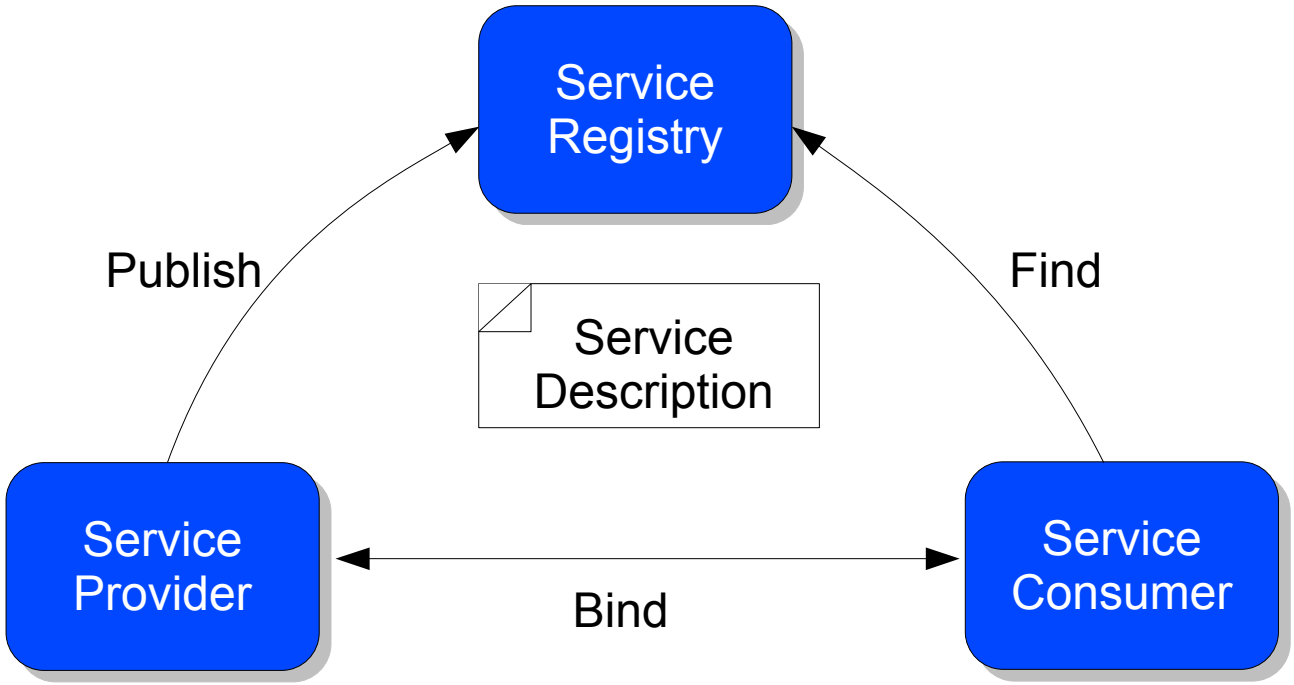
Service orientation

- Promoting a service-oriented interaction pattern



Service orientation

- Promoting a service-oriented interaction pattern



µServices

- Interface-based programming, but more
- Service Registry
  - Centrally accessible
  - Browsable
  - Notifications

- Service Registry Benefits
  - Consuming code is in control of provider selection
    - But not provider instantiation and configuration
  - Provider code is in control of when to provide
  - Promotes very loose coupling and late binding

META-INF/services

```
ServiceLoader<ShapeFactory> factories =
   ServiceLoader.load(ShapeFactory.class);

List<Shape> shapes = …

for (ShapeFactory factory : factories) {
   shapes.add(factory.next().createShape());
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

OSGi services

- OSGi framework provides the concepts we need
  - Centralized service registry
  - Consumer has control over selection
  - Provider has control over when to provide
  - Plus full-blown deployment and packaging modularity with run-time dynamism

OSGi service advantages

- Lightweight services
  - Direct method invocation
- Structured code
  - Promotes separation of interface from implementation
  - Enables reuse, substitutability, loose coupling, and late binding
- Dynamics
  - Loose coupling and late binding make it possible to support run-time management of module

Using a service (1/2)

- `BundleContext` allows bundles to find services

```
public interface BundleContext {
  …
  ServiceReference[] getServiceReferences(...);
  ServiceReference getServiceReference(...);
  Object getService(...);
  boolean ungetService(...);
}
```

Using a service (2/2)

```
public class Paint implements BundleActivator {
  public void start(BundleContext context) {
    ServiceReference ref = context.getServiceReference(
      com.foo.Shape.class.getName());
    if (ref != null) {
      Shape s = (Hello) context.getService(ref);
      if (s != null) {
        ...
        context.ungetService(h);
      }
    }
  }
  public void stop(BundleContext context) {
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Publishing a service (1/2)

- `BundleContext` allows bundles to publish services

```
public interface BundleContext {
   ...
   ServiceRegistration registerService(...);
   ...
}
```

Publishing a service (2/2)

```
public class Activator implements BundleActivator {
  private ServiceRegistration reg = null;

  public void start(BundleContext context) {
    reg = context.registerService(
      com.foo.Shape.class.getName(),
      new Circle(100), null);
  }
  public void stop(BundleContext context) {
    reg.unregister();
  }
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Service dynamism

- Services can be monitored

```
BundleContext.addServiceListener()

public interface ServiceListener extends EventListener {
  public void serviceChanged(ServiceEvent event);
}

public class ServiceEvent extends EventObject {
  public final static int REGISTERED    = 0x00000001;
  public final static int MODIFIED      = 0x00000002;
  public final static int UNREGISTERING = 0x00000004;
  public ServiceReference getServiceReference() { … }
  public int getType() { … }
  ...
}
```

	Data abstraction/encapsulation
	Provider/consumer coupling
	Provider/consumer control
	Provider/consumer dynamism

Services and dependency injection

- Services and dependency injection
  - Complementary
- Use POJOs
  - Avoid dependencies on OSGi API

Apache Felix iPOJO example

- Here is an iPOJO component providing the service

```
@Component
@Provides
public class Circle implements Shape {
...
}
```

Apache Felix iPOJO example

- Here is an iPOJO component providing the service

```
@Component
@Provides
public class Circle implements Shape {
...
}
```

- Implementation with service dependency

```
@Component
public class Paint {
  @Requires
  private Shape shape;

  public void useShape() {
    ...
  }
}
```

Apache Felix iPOJO example

- Here is an iPOJO component providing the service

```
@Component
@Provides
public class Circle implements Shape {
...
}
```

- Implementation with service dependency

```
@Component
public class Paint {
   @Requires
   private Shape shape;

   public void useShape() {
      ...
   }
}
```

Bundle activator no longer necessary, but lifecycle control still possible

Services and dependency Injection

- Advantages when combined with service orientation
  - Dependency injection no longer needs global view
    - Information localized to just the provider/consumer
  - No longer restricted to a single DI framework
    - Different DI frameworks can play together via the service registry

OSGi service disadvantages

- The downside to OSGi is that it requires a bottom-up commitment
  - You need to convert all of your code into proper modules to take advantage of services
  - A top-down approach of adopting services can help ease migration to more modular code

PojoSR

and (OSGi) µServices for the rest of us

What is PojoSR?

- It largely removes the modularity layer from the OSGi framework

- Provides
  - A centralized service registry based on OSGi API
  - Lifecycle hooks for JAR files
  - A “light” OSGi framework for the class path

- Available at http://pojosr.googlecode.com

Why this approach?

- OSGi API is a standard with years of experience behind it
- Can re-use OSGi modules (a.k.a. bundles) and/or technology
- Can leverage services without having to completely modularize first (i.e., top-down)
- Provides a path to full-blown modularity
  - Go see BJ Hargrave and Peter Kriens slides on "Service Migration First"

What you keep

- Bundle activator
  - JAR file lifecycle hook
    - Gives JAR file a lifecycle state (started or stopped)
  - Two simple methods (start()/stop())
    - Give you a bundle context
- Bundle context
  - Allows you to
    - Lookup services
    - Provide services
    - Listen for services
      - *or you can still use iPOJO instead*

What you lose

- Module-private encapsulation
- Side-by-side versions
- Dependency consistency checking
- Dynamic module deployment
- Full OSGi compatibility

What you don't lose

- Surprisingly, some of the seemingly more advanced OSGi dynamism features
  - Bundle-based dynamism
  - Service-based dynamism
- How is this possible?

Bundle-based dynamism

- Bundle lifecycle state provides a hook for bundle-based dynamic extensibility
- The extender pattern
  - An application component, called the extender, listens for bundles to be started, and stopped
  - On startup, the extender probes bundles to see if they are extensions
    - Typically, extension contain special metadata or resources to indicate they provide an extension
  - When an extension is started, the extender integrates the extension into the application
  - When an extension is stopped, the extender removes the extension from the application

Service-based dynamism

- Service lifecycle state provides a hook for service-based dynamic extensibility
  - Still overall controlled by bundle state, but more fine grained
- Treats the service registry as a whiteboard
  - A reverse way to create a service
- An application component listens for services of a particular type to be added and removed
- On addition, the service is integrated into the application
- On removal, the service is removed from the application

Wrap up

- (Dynamic) Services and a service registry are needed in Java
- The OSGi framework provides concepts we need
  - But requires bottom-up commitment
- PojoSR allows you to use OSGi concepts in standard Java
  - Makes it possible to leverage existing OSGi technology, like iPOJO
  - A top-down approach of adopting services can help ease migration to more modular code
- Works so well that OSGi is thinking of standardizing this approach

Felix Connect

Questions?