



adaptTo()

APACHE SLING & FRIENDS TECH MEETUP
BERLIN, 23-25 SEPTEMBER 2013

OSGi μ Services
Karl Pauls

Karl Pauls - karlpauls@gmail.com

Member Apache Software Foundation

Felix, ACE, Sling, Incubator: Celix

Co-author of „OSGi in Action“



Outline

- **Motivating μ Services**
 - Procedures
 - Objects
 - Interfaces
 - Factories
 - Dependency injection
- **Service orientation**
 - μ Services and OSGi Services
 - Dynamism
 - Service Dependency Injection
- **Example**
 - Declarative Services with Configuration Admin Service

Motivating μ Services

Motivating μ Services



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

```
byte[4096] canvas;
```

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

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

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

```
byte[4096] canvas;
```

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

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



Data encapsulation/abstraction

Provider/Consumer coupling

Provider/Consumer control

Provider/Consumer dynamism


```
byte[4096] canvas;
```

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

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



Data encapsulation/abstraction



Provider/Consumer coupling

Provider/Consumer control

Provider/Consumer dynamism

```
byte[4096] canvas;
```

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

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



Data encapsulation/abstraction



Provider/Consumer coupling



Provider/Consumer control

Provider/Consumer dynamism

```
byte[4096] canvas;
```

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

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



Data encapsulation/abstraction



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

	Data encapsulation/abstraction
	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 mouseClicked(int x, int y) {
        shape.draw(canvas, x, y);
    }
    public static void main() {
        new Paint(
            new Canvas(),
            new Circle(100));
    }
}
```



Data encapsulation/abstraction

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



Data encapsulation/abstraction



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

```



Data encapsulation/abstraction



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



Data encapsulation/abstraction



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



Data encapsulation/abstraction



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

```



Data encapsulation/abstraction



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 encapsulation/abstraction

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 encapsulation/abstraction



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 encapsulation/abstraction



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 encapsulation/abstraction



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 encapsulation/abstraction

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 encapsulation/abstraction
	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 encapsulation/abstraction
	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 encapsulation/abstraction
	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 encapsulation/abstraction
	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 encapsulation/abstraction
	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();
    }
}
```

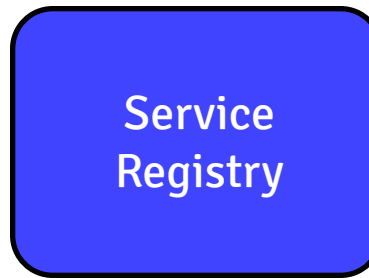
```
Paint paint =
    beanContainer.getBeanByType(
        Paint.class);
```

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

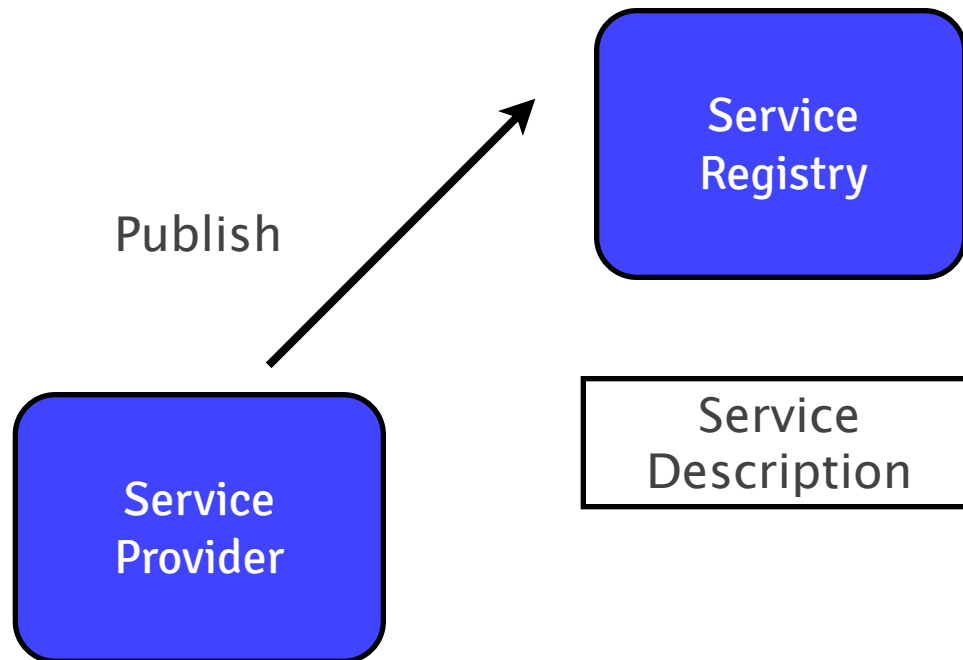
Service Orientation

- Promoting a service-oriented interaction pattern

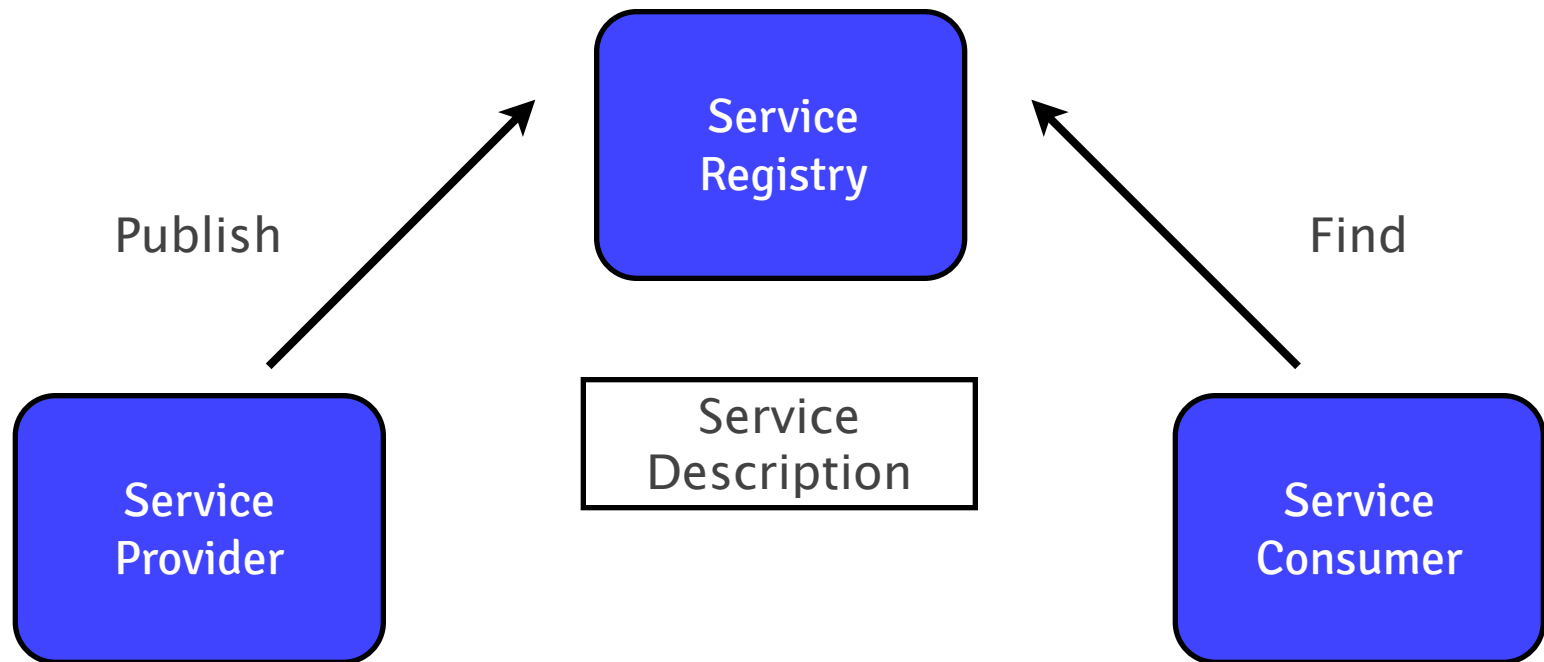
- Promoting a service-oriented interaction pattern



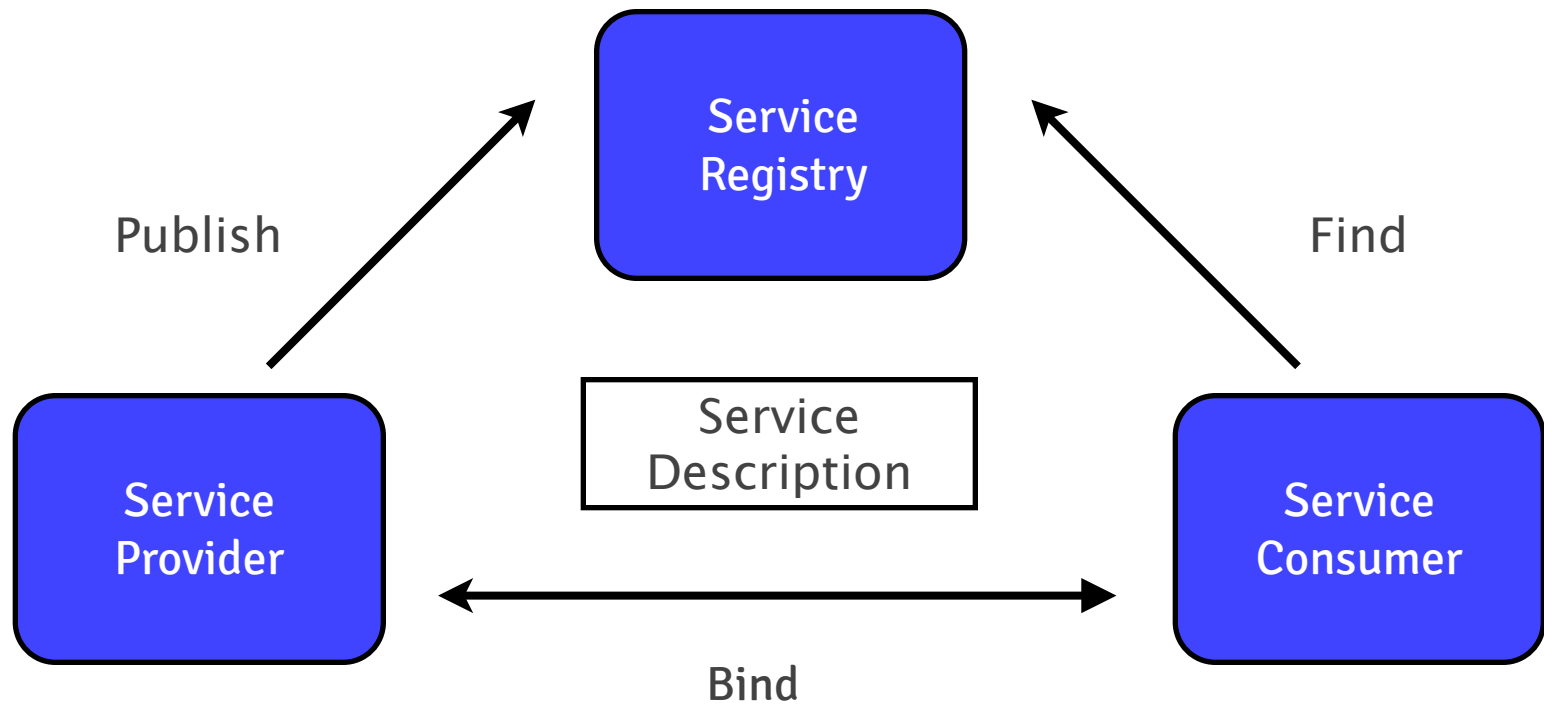
- Promoting a service-oriented interaction pattern



- Promoting a service-oriented interaction pattern



- Promoting a service-oriented interaction pattern



- Interface-based programming, but more
- Service Registry
 - Centrally accessible
 - Browsible
 - 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





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

```
List<Shape> shapes = ...
```

```
for (ShapeFactory factory : factories) {
```

```
    shapes.add(factory.next().createShape());
```

```
}
```

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

- 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

- 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 = (Shape) context.getService(ref);
            if (s != null) {
                ...
                context.ungetService(
                    ref);
            }
        }
    }
}

```

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

- BundleContext allows bundles to publish services

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

Publishing a service (1/2)

```

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 encapsulation/abstraction



Provider/Consumer coupling



Provider/Consumer control

Provider/Consumer 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;  
    public final static int MODIFIED;  
    public final static int UNREGISTERING;  
    ...  
}
```

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

- 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;
    public final static int MODIFIED;
    public final static int UNREGISTERING;
    public ServiceReference
        getServiceReference() { ... }
    public int getType() { ... }
    ...
}
```

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

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

- Here is a component providing the service

```
@Component
```

```
public class Circle implements Shape {
```

```
...
```

```
}
```

- Here is a component providing the service

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

- Implementation with service dependency

```
@Component  
public class Paint {  
    @Reference  
    public void setShape(Shape shape)  
    {...}  
    public void useShape() {... }  
}
```

- Here is a component providing the service

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

- } Implementation with service dependency

```
@Component
public class Paint {
    @Reference
    public void setShape(Shape s) {
        {...}
    }
    @Activate
    public void useShape() {... }
}
```

Bundle activator no longer necessary,
but lifecycle control still possible

- 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

Example

```
@Component(  
  configurationPolicy=ConfigurationPolicy.REQUIRE)  
public class Circle implements Shape {  
  ...  
  private volatile String color;  
  
  @Activate  
  public void init(Map config){  
    this.color = config.get(COLOR_KEY); }  
}
```

Declarative Services with Configuration Admin Service

```
@Component(immediate=true)  
public class Paint {
```

```
@Reference(  
    cardinality=ReferenceCardinality.MULTIPLE,  
    policyOption=ReferencePolicyOption.DYNAMIC  
)  
    public void setShape(Shape shape) {...}  
    public void unsetShape(Shape shape){...}
```

```
@Activate  
    public void drawFrame() { ...}
```

```
@Deactivate  
    public void dispose(){...}  
}
```

Declarative Services with Configuration Admin Service

```

@Component(configurationPolicy=ConfigurationPolicy.REQUIRE)
public class Circle implements Shape {
    private volatile String color;

    @Activate
    public void init(Map config){ this.color = config.get(COLOR_KEY);}
}

@Component(immediate=true)
public class Paint {
    @Reference(
        cardinality=ReferenceCardinality.MULTIPLE,
        policyOption=ReferencePolicyOption.DYNAMIC
    )
    public void setShape(Shape shape) {...}
    public void unsetShape(Shape shape){...}
    @Activate
    public void drawFrame() { ...}
    @Deactivate
    public void dispose() {....}
}

```



- OSGi μ Services
 - Promote separation of interface from implementation
 - Enable reuse, substitutability, loose coupling, and late binding
 - Provide (dynamic) services via a service registry
 - Loose coupling and late binding make it possible to support run-time management of module

Questions?