OSGi Application Best Practices

Emily Jiang, emijiang@uk.ibm.com

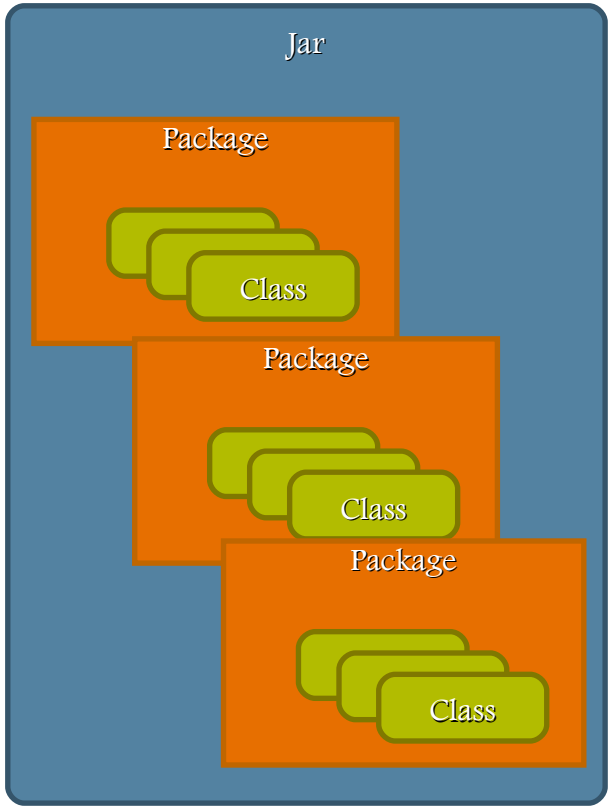
WebSphere Application Server OSGi Developer, Apache Aries Committer

AGENDA

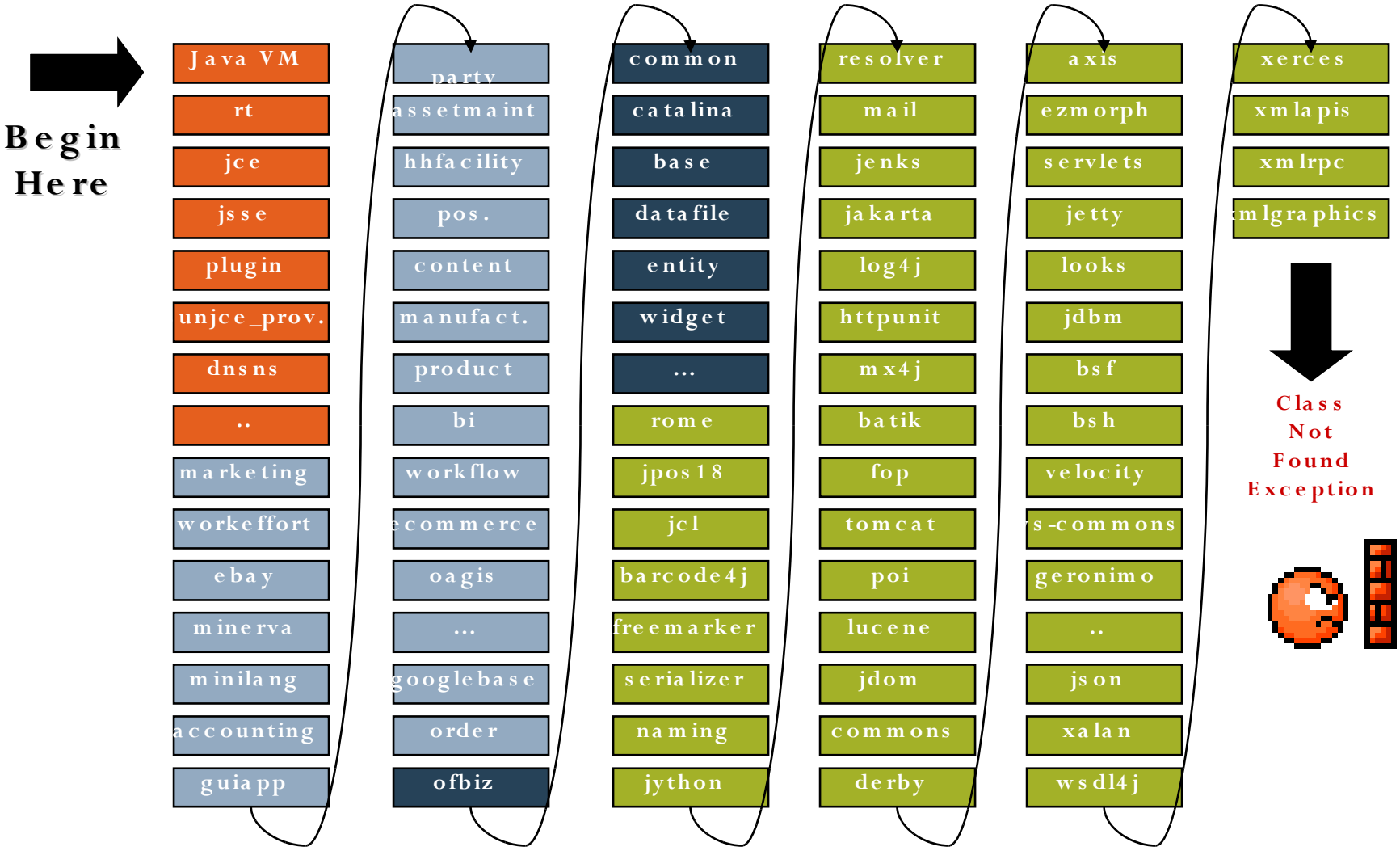
- Why OSGi?
- What is OSGi?
- How to best use OSGi?

Modularization in Java

- Jars have no modularization characteristics
  - No “jar scoped” access modifiers.
  - No means for a jar to declare its dependencies.
  - No versioning.



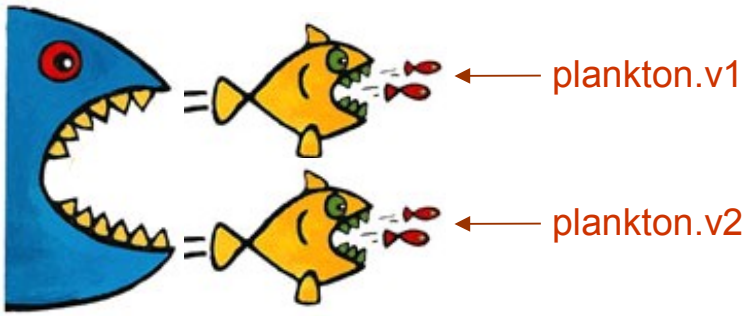
Problems with Global Java ClassPath



Problems with EARs/WARs

Enterprise Applications have isolated classpaths but…

- No Sharing
  - Common libraries/frameworks in apps and memory
- Version conflicts



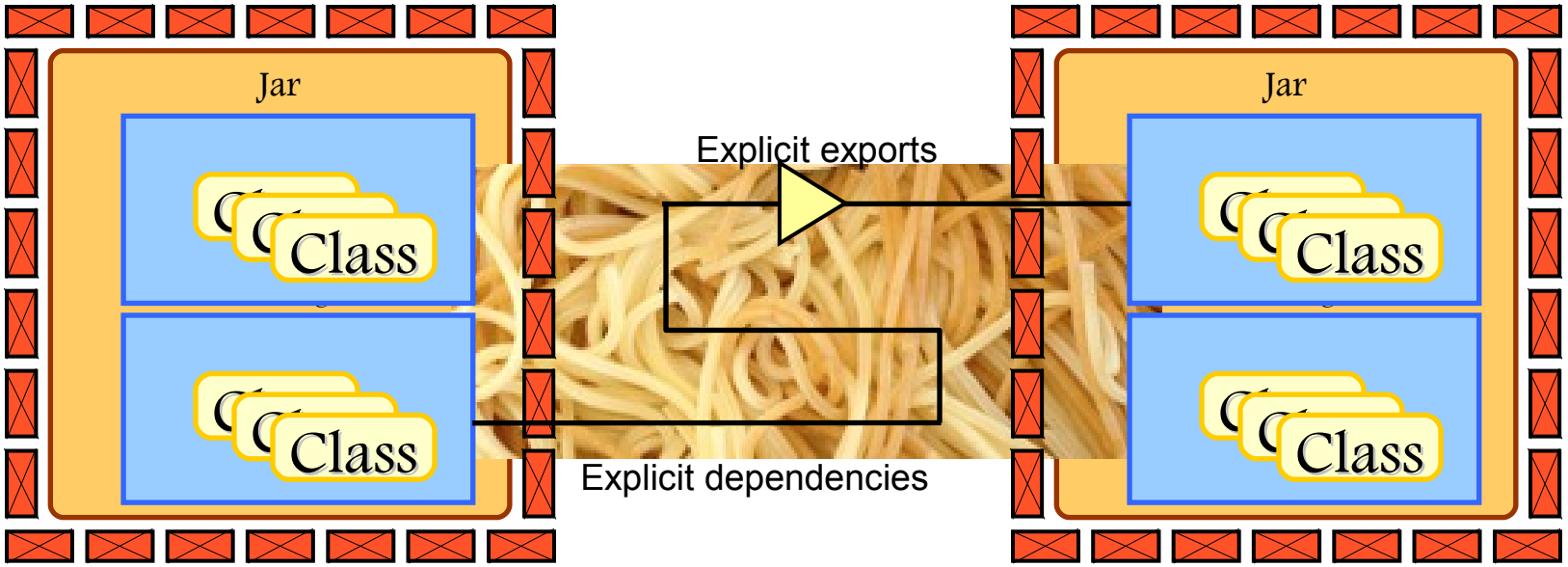
AGENDA

- Why OSGi?
- What is OSGi?
- How to best use OSGi?

What is OSGi?

"The dynamic module system for Java"

- Mature 10-year old technology
- Governed by OSGi Alliance: http://www.osgi.org
- Used *inside* just about *all* Java-based middleware
  - IBM WebSphere, Oracle WebLogic, Red Hat JBoss, Sun GlassFish, Paremus Service Fabric, Eclipse Platform, Apache Geronimo, (non-exhaustive list)
    http://www.osgi.org/wiki/uploads/News/2008_09_16_worldwide_market.pdf



OSGi Bundles and Class Loading

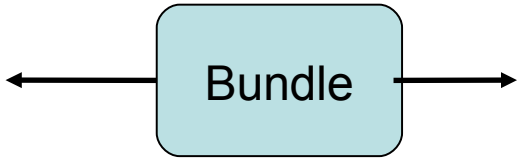
OSGi Bundle – A jar containing:

- Classes and resources.
- OSGi Bundle manifest.

What's in the manifest:

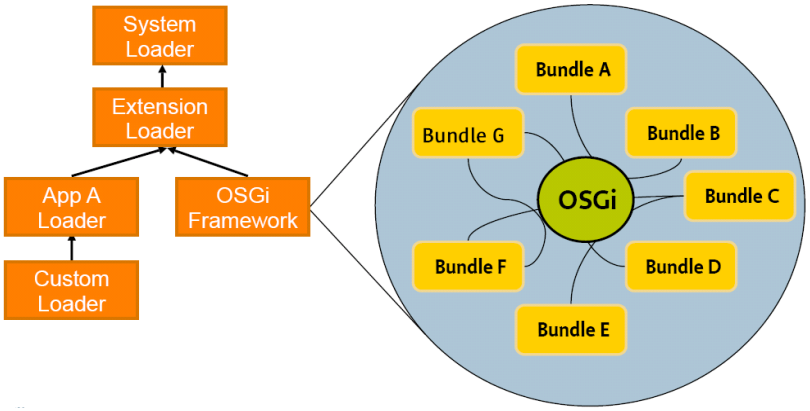
- Bundle-Version: Multiple versions of bundles can live concurrently.
- Import-Package: What packages from other bundles does this bundle depend upon?
- Export-Package: What packages from this bundle are visible and reusable outside of the bundle?

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyService bundle
Bundle-SymbolicName: com.sample.myservice
Bundle-Version: 1.0.0
Bundle-Activator: com.sample.myservice.Activator
Import-Package: com.something.i.need;version="1.1.2"
Export-Package: com.myservice.api;version="1.0.0"
```



Class Loading

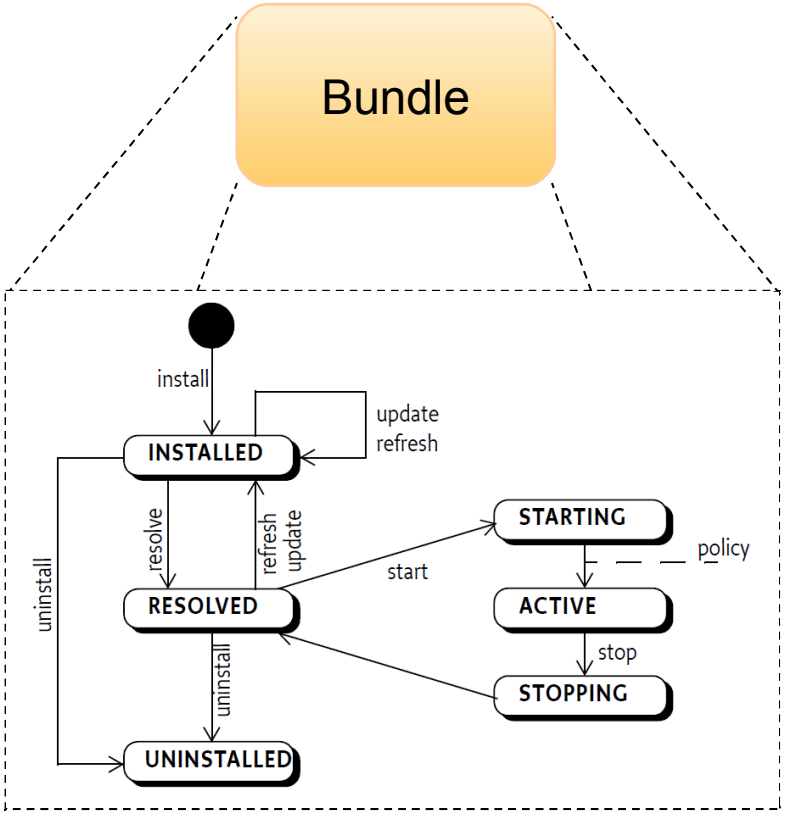
- Each bundle has its own loader.
- No flat or monolithic classpath.
- Class sharing and visibility decided by declarative dependencies, not by class loader hierarchies.
- OSGi framework works out the dependencies including versions.



OSGi Bundle

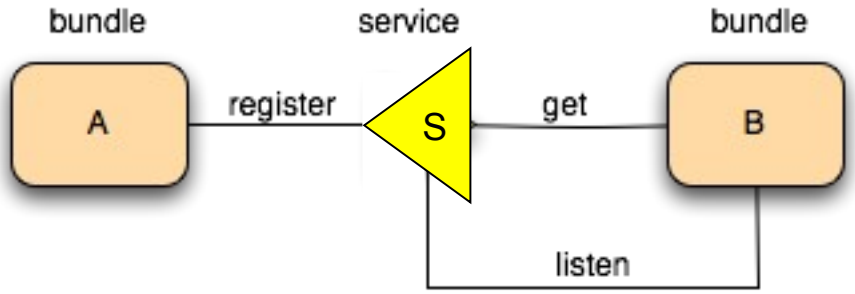
Bundles have a dynamic lifecycle

Can come and go independently



OSGi Service Registry

- Service
  - An object associated with a list of classes (usually interfaces) it provides
  - Dynamic (can come and go), framed by bundle lifecycle
  - Services are the primary means of collaboration between bundles.



AGENDA

- Why OSGi?
- What is OSGi?
- How to best use OSGi?

BP1 - Use Import-Package not Require-Bundle

Require-Bundle

- Tightly coupled with a particular bundle with the specified symbolic name and version
- High coupling between bundles
- Import all packages
- Bundle version management

MANIFEST.MF

...

Require-Bundle: com.ibm.ws.service;bundle-version=2.0.0

- Import-Package
  - Can wire to any bundles exporting the specified package
  - Loose coupling between bundles
  - Only import the package you need
  - Package version management

MANIFEST.MF

...

Import-Package: com.ibm.ws.service.api;version=2.0.0

BP2 - Avoid split packages

- Split package
  - A package is exported by two bundles at the same version and the set of classes provided by each bundle differs.
- Why?
  - Leads to the use of Require-Bundle, compromising the extent to which systems using the bundles can be extended and maintained.
- How?
  - Keep all of the classes from any one package in a single bundle

BP2 - Split Package examples

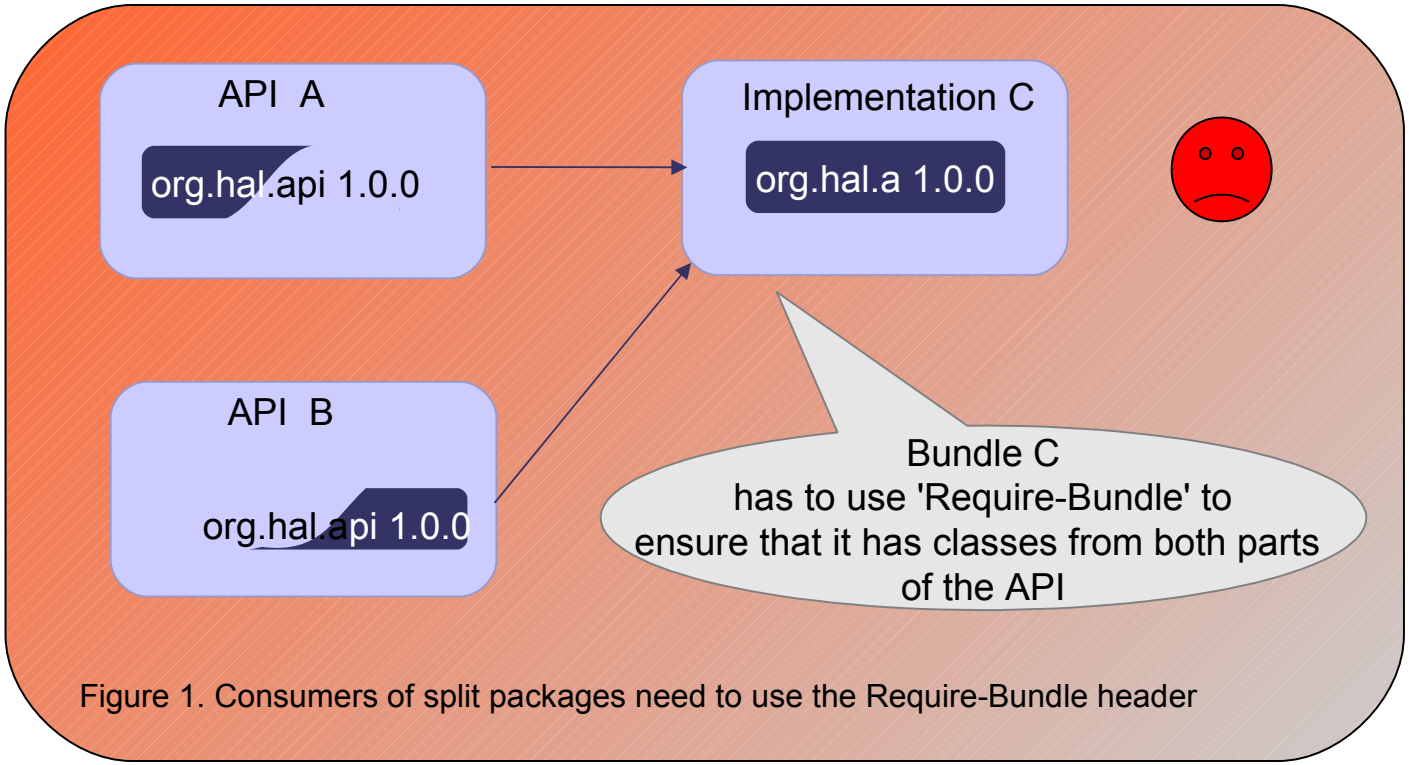


Figure 1. Consumers of split packages need to use the Require-Bundle header

BP2 - Split Package examples

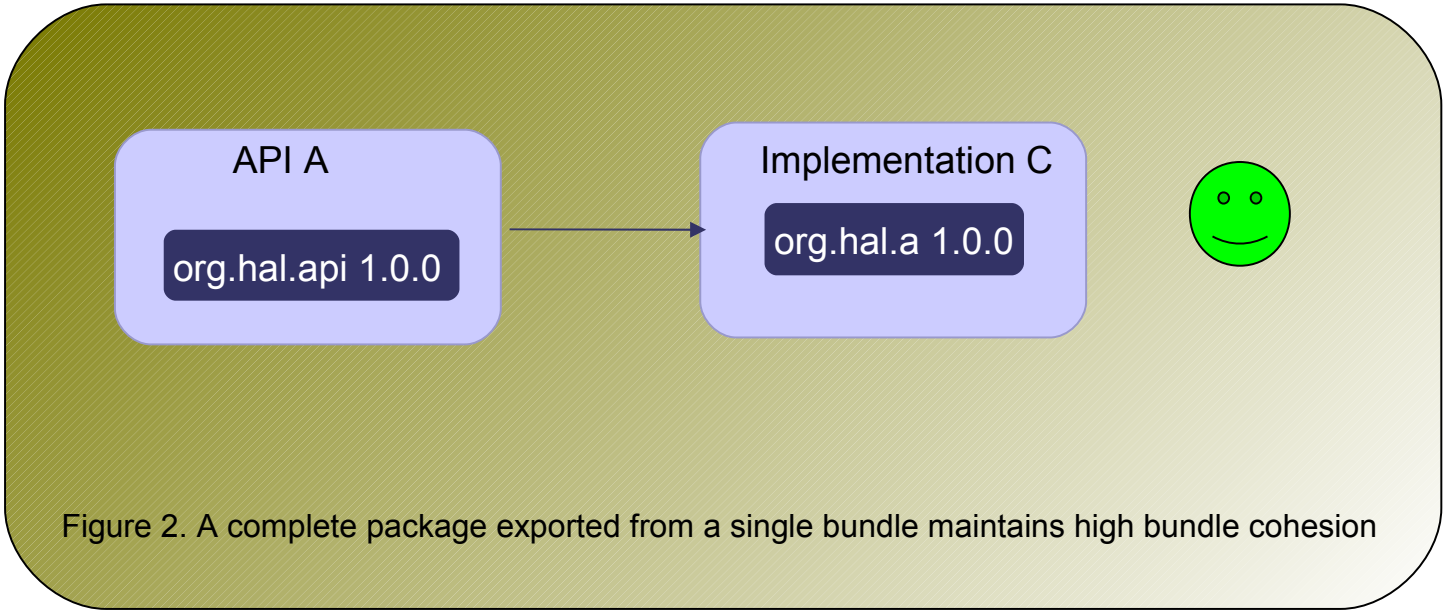


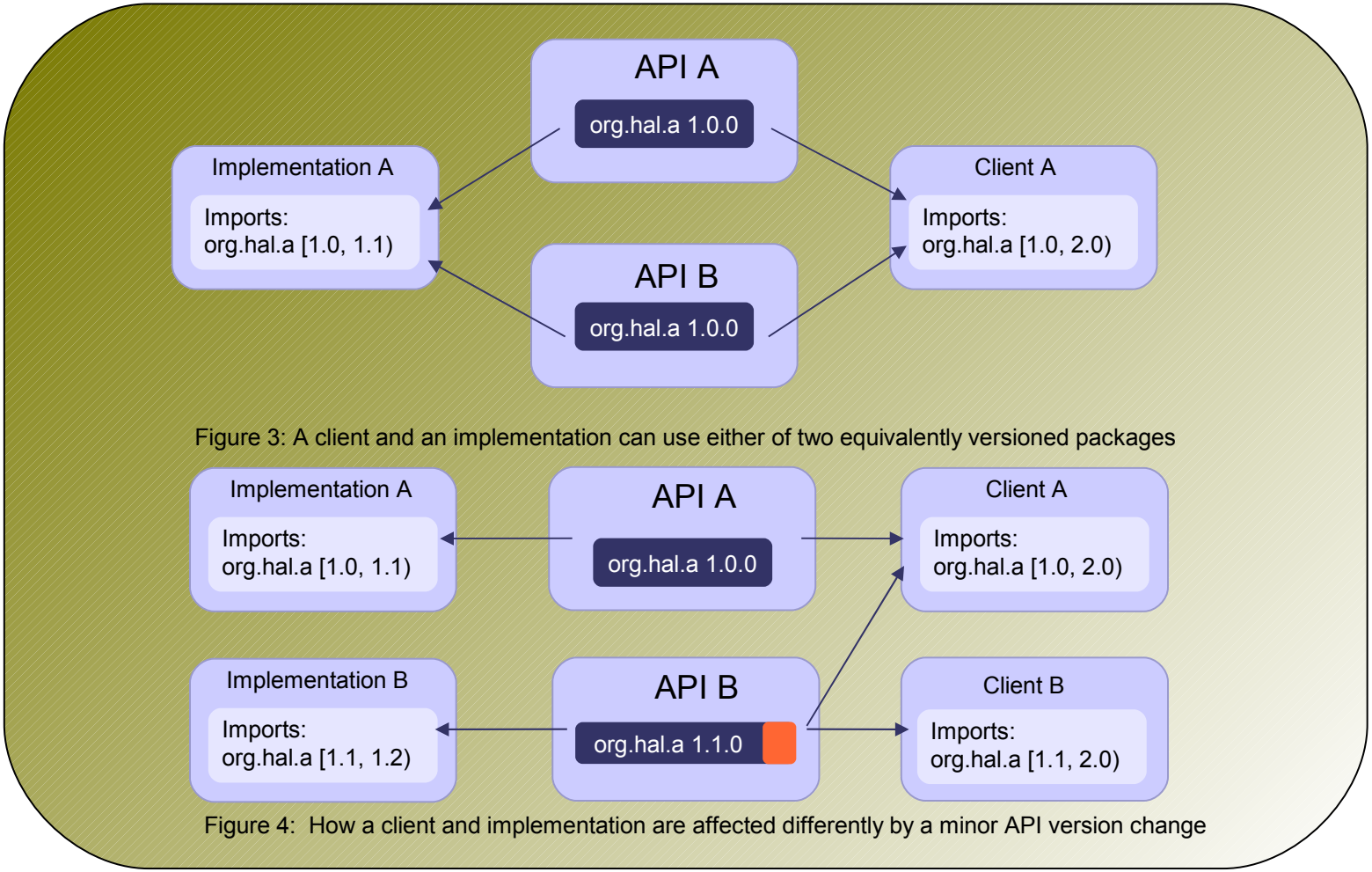
Figure 2. A complete package exported from a single bundle maintains high bundle cohesion

BP3 - Version bundles and packages

- What is semantic versioning?
  - Uses a major.minor.micro.qualifier numbering scheme
    - Major - Packages with versions that have different major parts are not compatible both for providers as well as consumers.
    - Minor – API enhancement, e.g. adding methods to an API
    - Micro – bug fixing
    - Qualifier – identifier such as timestamp
  - Changes in major: a binary incompatible, minor: enhanced API, micro: no API changes

- Why?
  - Clients can protect themselves against API changes that might break them.

BP3 - Version bundles and packages examples



BP3 - Version bundles and packages examples

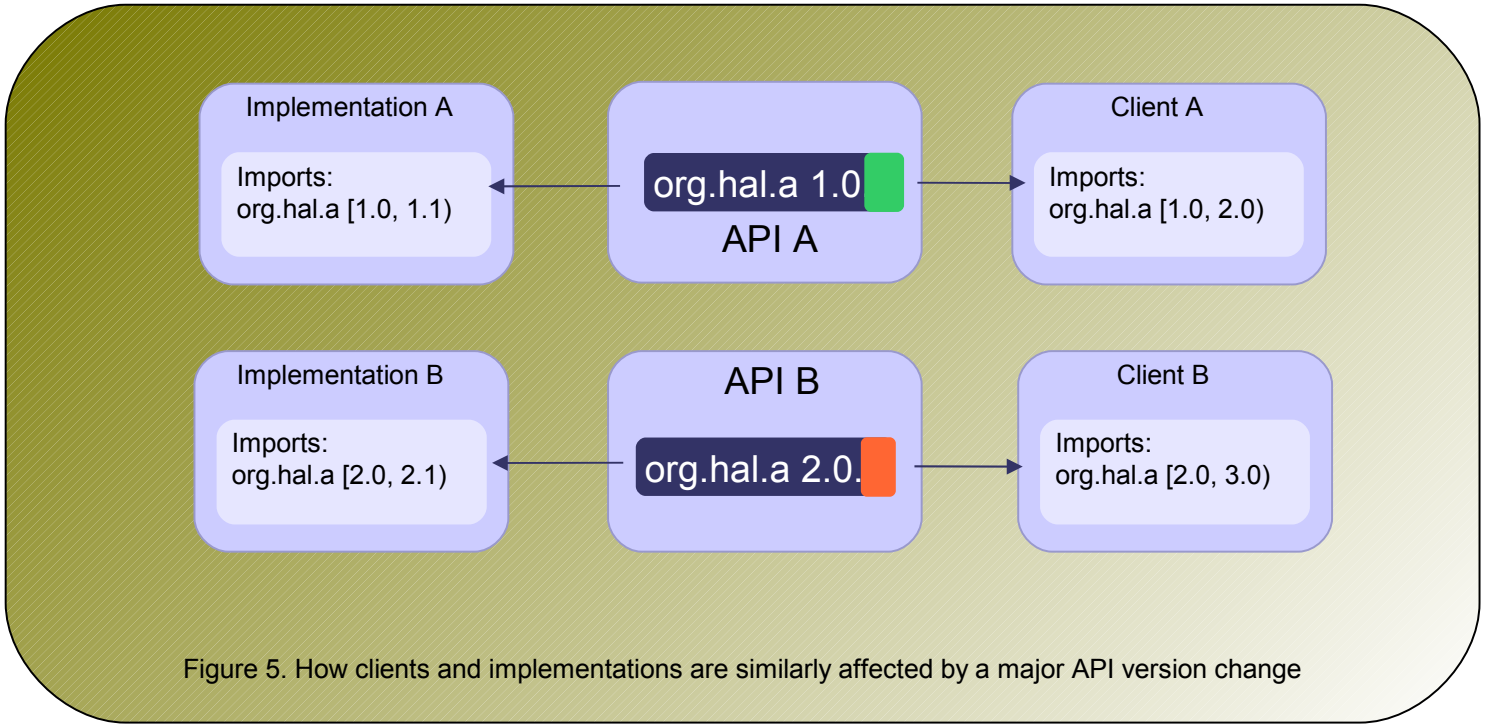


Figure 5. How clients and implementations are similarly affected by a major API version change

BP4 - Separate API from Implementations

- Why?
  - Great flexibility
  - Many implementation bundles → enable more services provided
  - Reduce package dependencies → reduce circular dependencies
- How?
  - Put API classes in one bundle
  - Put implementation classes in a separate bundle

BP4 - Separate API and implementation examples

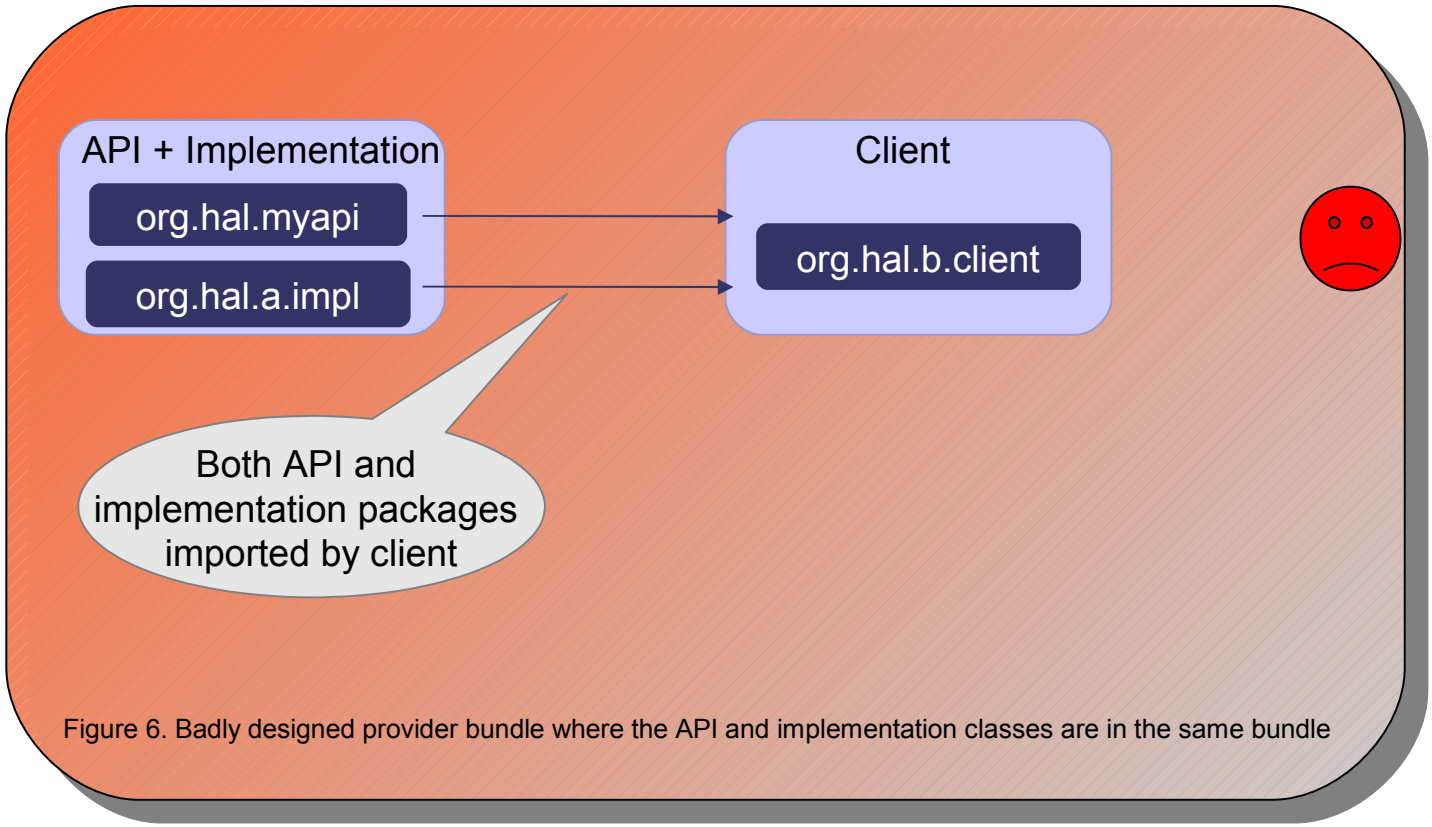
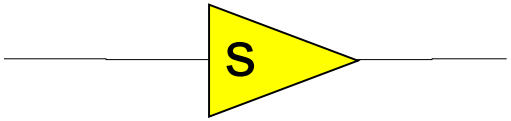


Figure 6. Badly designed provider bundle where the API and implementation classes are in the same bundle

BP5 - Share services not implementations

- Use the OSGi service registry to construct instances
- Why?
  - Able to obtain an instance of an implementation without knowing which one
  - Achieve a loosely coupling of client, API and implementation
- How?



  - Register an instance of the API interface in the OSGi service registry
  - Register an implementation of the OSGi ServiceFactory interface in the OSGi service registry.

BP4 & 5 - examples

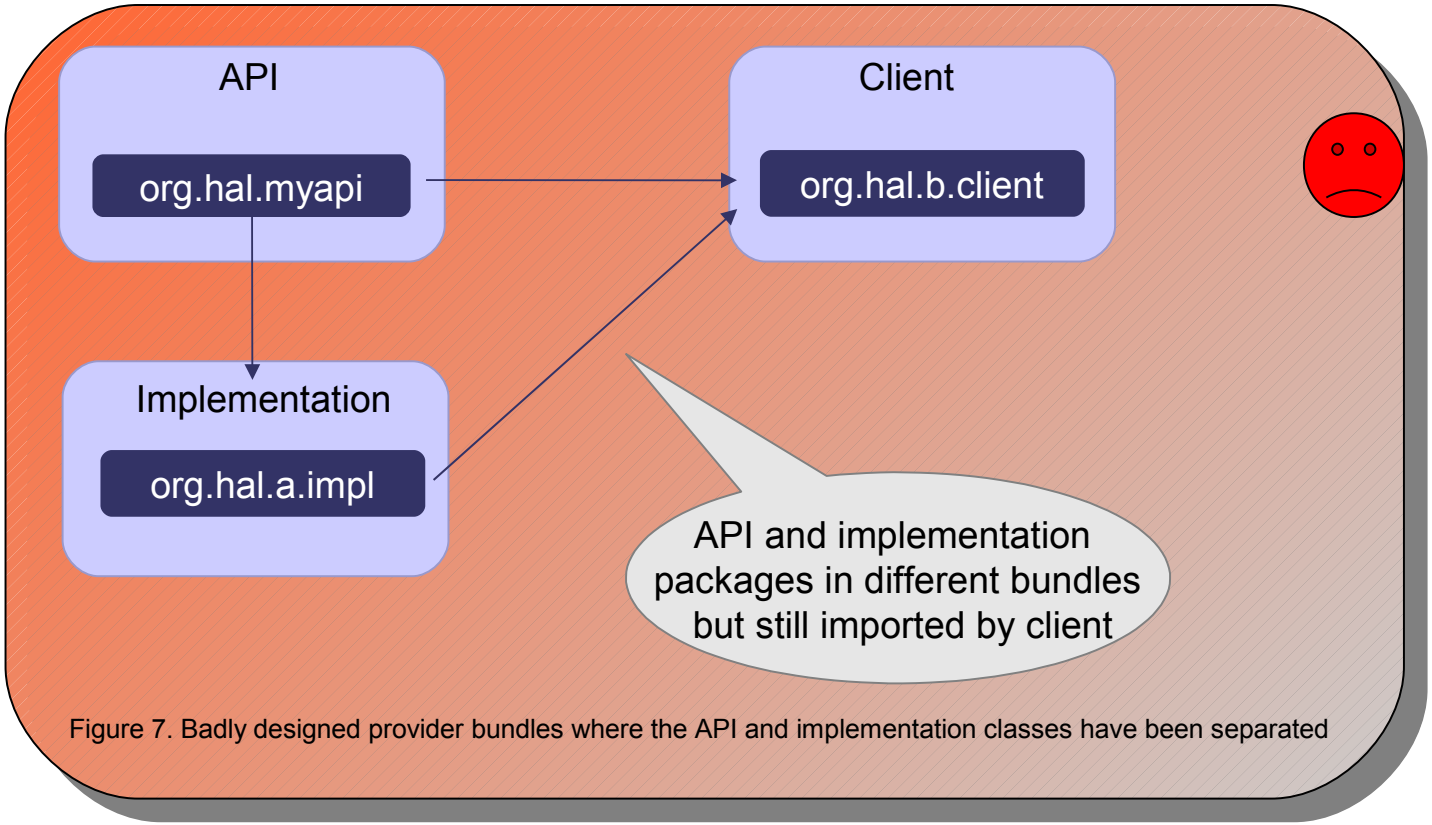


Figure 7. Badly designed provider bundles where the API and implementation classes have been separated

BP4 & 5 - examples

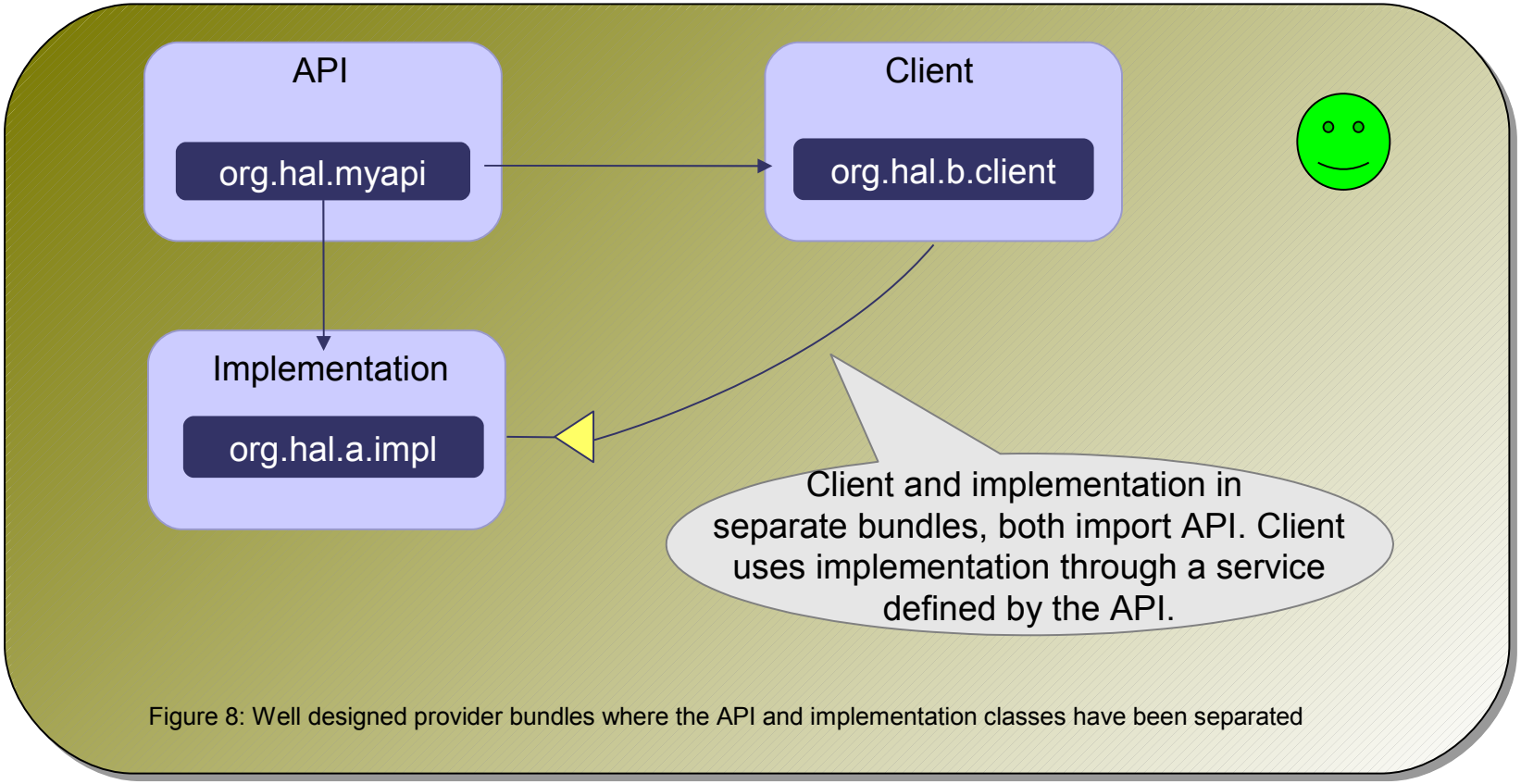


Figure 8: Well designed provider bundles where the API and implementation classes have been separated

BP6 – Make Bundles Loosely Coupled & Highly Cohesive

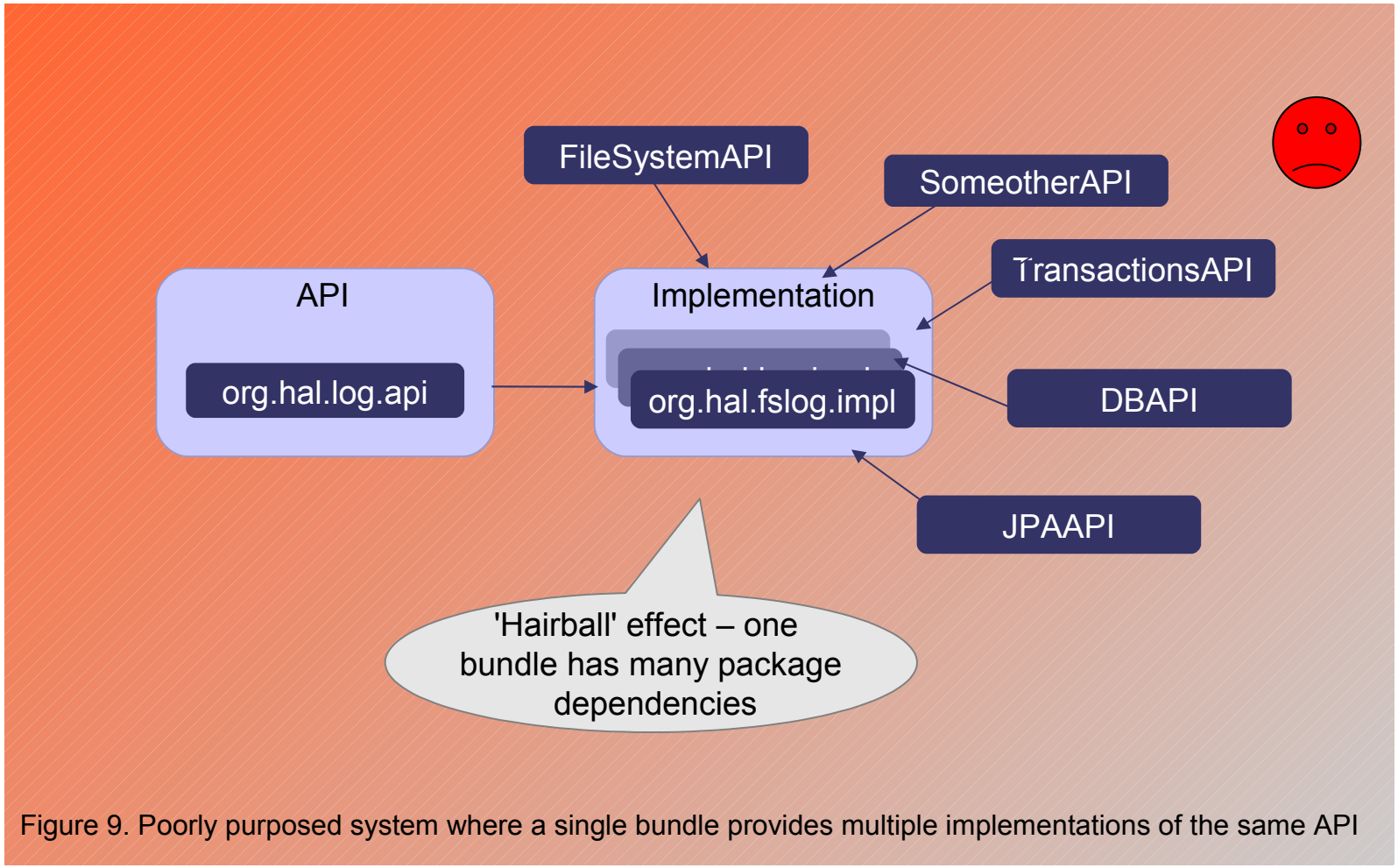


Figure 9. Poorly purposed system where a single bundle provides multiple implementations of the same API

BP6 – Make Bundles Loosely Coupled & Highly Cohesive

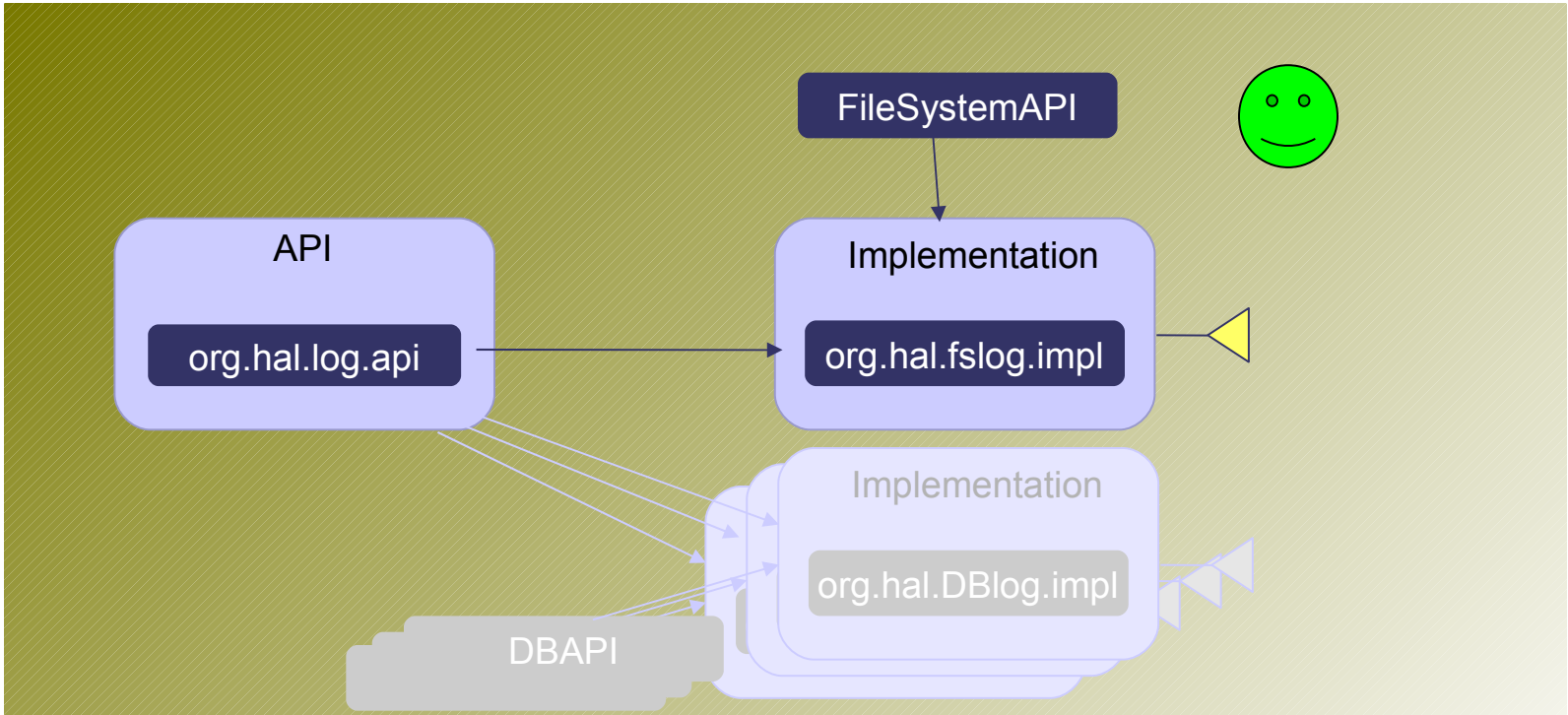


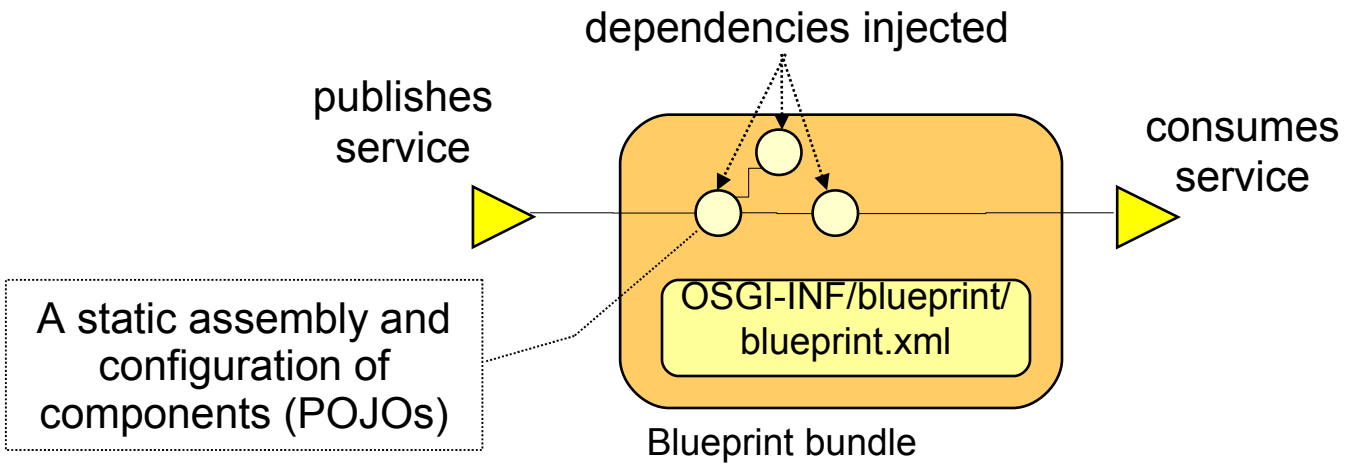
Figure 10. A well purposed system where each implementation of an API is provided by a separate bundle

Using services can be hard!

```
private BundleContext ctx;


private AtomicReference<LogService> ls = new
AtomicReference<LogService>();
private AtomicReference<ServiceReference> lr = new
AtomicReference<ServiceReference>();


public void start(BundleContext ctx) throws
InvalidSyntaxException
{
this.ctx = ctx;
ctx.addServiceListener(this,
"(objectClass=org.osgi.service.log.LogService)");
ServiceReference ref =
ctx.getServiceReference(LogService.class.getName());
if (ref != null) {
ls.set((LogService) ctx.getService(ref));
lr.set(ref);
}
}
```

```
@Override
public void serviceChanged(ServiceEvent event)
{
ServiceReference ref = event.getServiceReference();


if (ls.get() == null && event.getType() ==
     ServiceEvent.REGISTERED) {
ls.set((LogService) ctx.getService(ref));
} else if (ls.get() != null && event.getType() ==
     ServiceEvent.UNREGISTERING &&
ref == lr.get()) {
ref = ctx.getServiceReference(LogService.class.getName());
if (ref != null) {
ls.set((LogService) ctx.getService(ref));
lr.set(ref);
}
}
}
```

BP7 - Use Blueprint

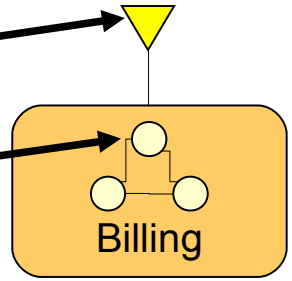


- Specifies a Dependency Injection container, standardizing established Spring conventions
- Configuration and dependencies declared in XML “module blueprint”, which is a standardization of Spring “application context” XML.
  - Extended for OSGi: publishes and consumes components as OSGi services
- Simplifies unit test outside either Java EE or OSGi r/t.
- **The Blueprint DI container is a part of the server runtime (compared to the Spring container which is part of the application.)**

BP7 - Blueprint service-bundle examples

Billing service bundle

```
<blueprint>
  <service ref="service" interface =
        "org.example.bill.BillingService" />
  <bean id="service" scope="prototype"
    class="org.example.bill.impl.BillingServiceImpl" />
</blueprint>
```



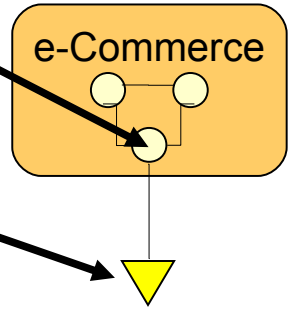
-"prototype" scope indicates a new instance is created by the container for each use.
-"singleton" scope is the default.

```
public interface BillingService {
    void bill(Order o);
}
```

BP7- Blueprint client-bundle examples

e-Commerce bundle

```
<blueprint>
  <bean id="shop" class="org.example.ecomm.ShopImpl">
    <property name="billingService" ref="billingService" />
  </bean>
  <reference id="billingService"
    interface="org.example.bill.BillingService" />
</blueprint>
```




```
public class ShopImpl {

private BillingService billingService;
void setBillingService(BillingService srv) {
    billingService = srv;
    }

void process(Order o) {
    billingService.bill(o);
    }
}
```

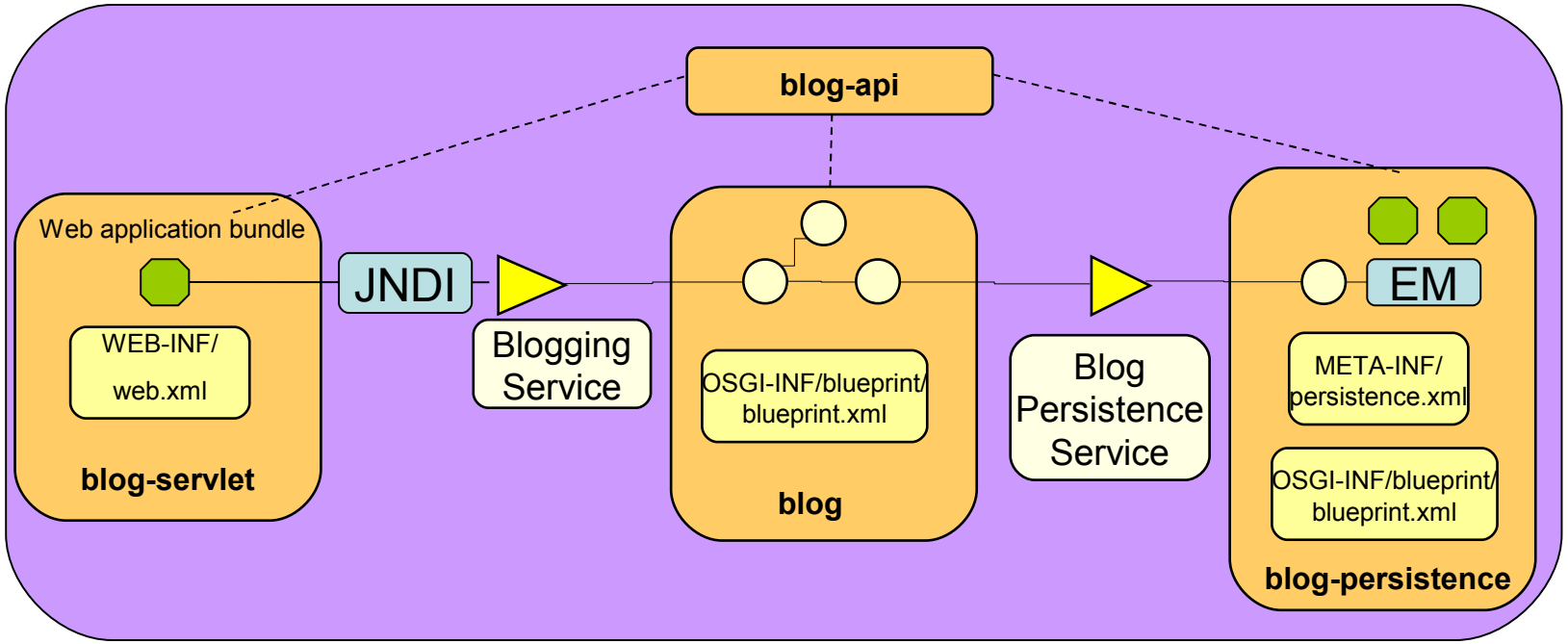
-injected service reference
-service can change over time
-can be temporarily absent
  without the bundle caring
-managed by Blueprint container

Summary

OSGi Application Best Practices

- BP1 - Use Import-Package instead of Require-Bundle
- BP2 - Avoid split packages
- BP3 - Version bundles and packages
- BP4 - Separate API from implementations
- BP5 - Share services not implementations
- BP6 – Make bundles loosely coupled & highly cohesive
- BP7 - Use Blueprint

Quiz – What best practices are used?



Questions?