Converting (Large) Applications to OSGi™

BJ Hargrave, Senior Technical Staff Member at IBM

Peter Kriens, Technical Director, OSGi

TS-5122

Java

Sun

Learn How to Simplify Application Development by Building for the OSGi Service Platform

And have fun doing it!

Agenda

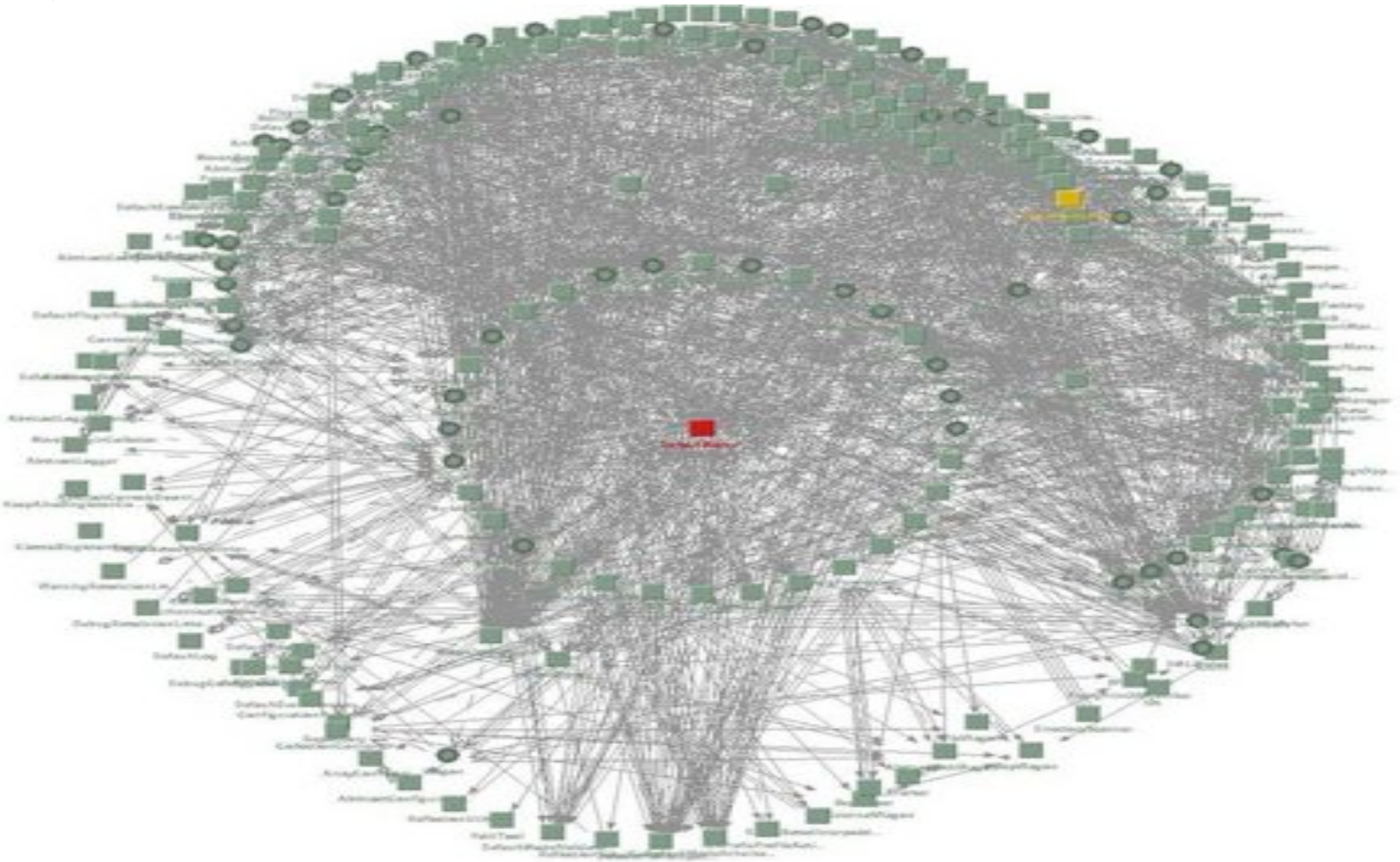
- Modularization
- Modularization in Java™ Apps
- The OSGi™ Framework
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- Building
- Pitfalls
- Conclusion

Modularization

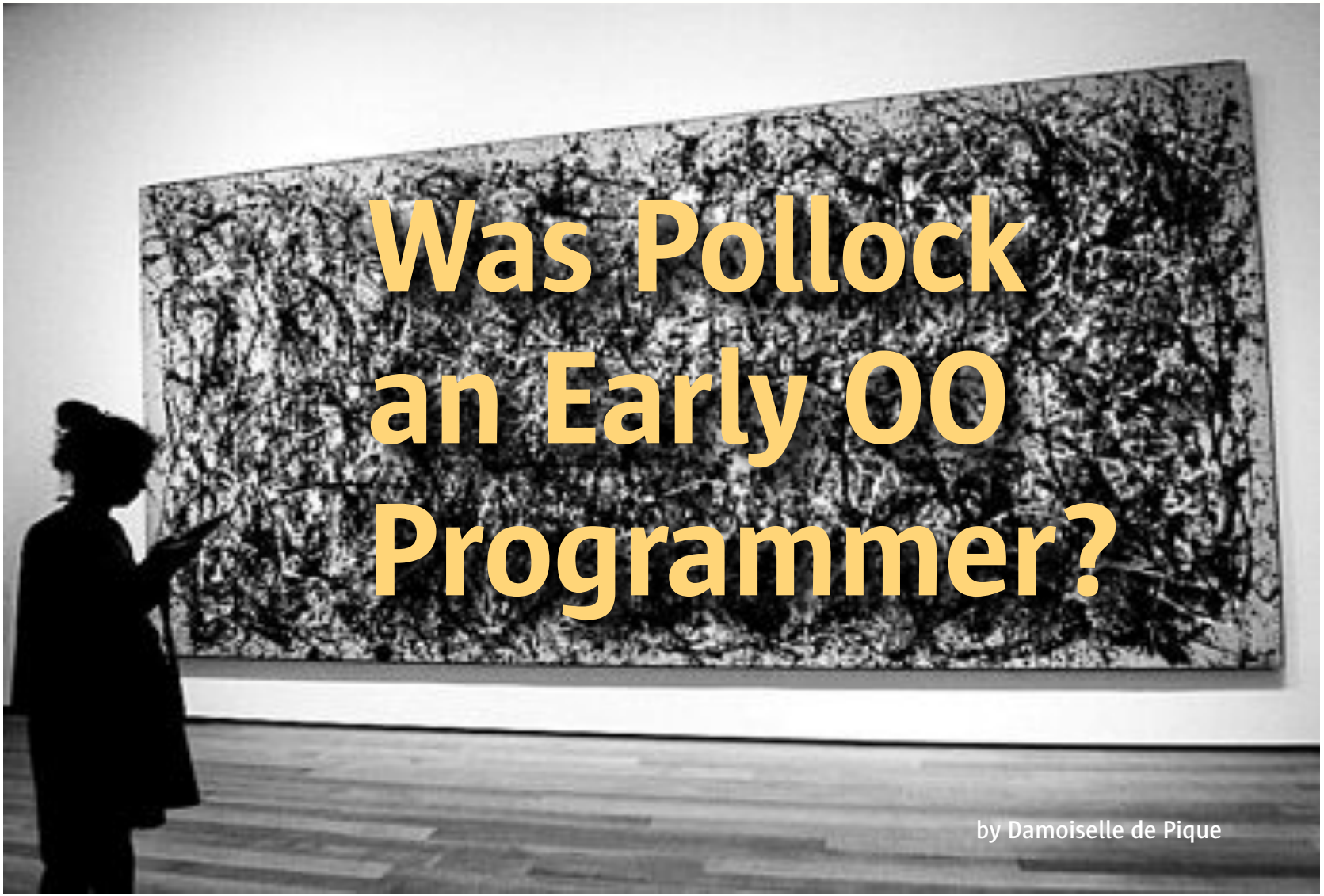
- Recycling old ideas that were raised in the early seventies by, among others, David Parnas:
  - High Cohesion
  - Low Coupling
- Modularization minimizes complexity by creating proper boundaries between the parts that compose a system
- Properly modularized systems are easier to maintain and extend
  - Changes are more localized and affect less of the overall system

What happened with Modularization and Object Oriented (OO)?

What happened with Modularization and Object Oriented (OO)?




by Damoiselle de Pique



Was Pollock an Early OO Programmer?

by Damoiselle de Pique

What happened with Modularization and OO?

- Focus in OO was on encapsulation of instance variables, which is some form of modularization, but granularity is too small
- OO systems become tangled webs quickly
  - Program knows its own structure
- Patterns like SOA, Factories, Dependency Injection, Inversion of Control are trying to minimize the consequences of OO and its lack of modularization

Agenda

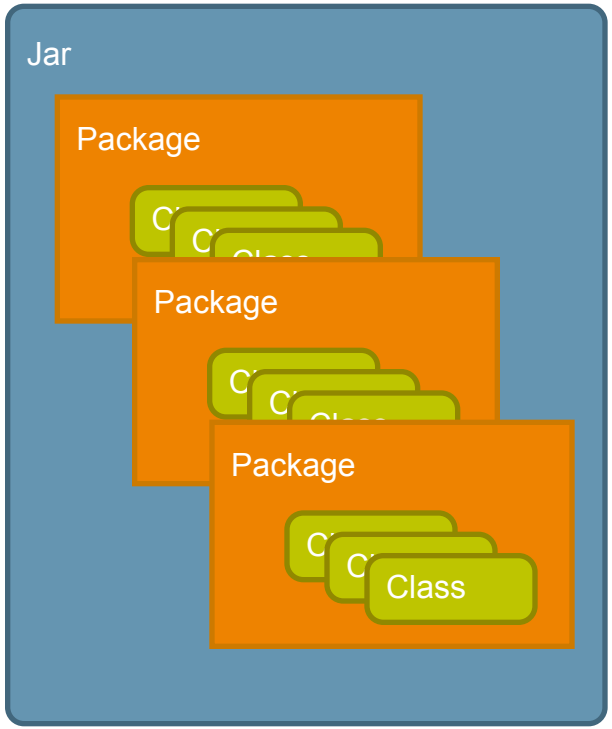
- Modularization
- **Modularization in Java™ Apps**
- The OSGi Framework
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- Building
- Pitfalls
- Conclusion

Agenda

- Modularization
- **Modularization in Java™ Apps**
- The OSGi Framework
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- Building
- Pitfalls
- Conclusion

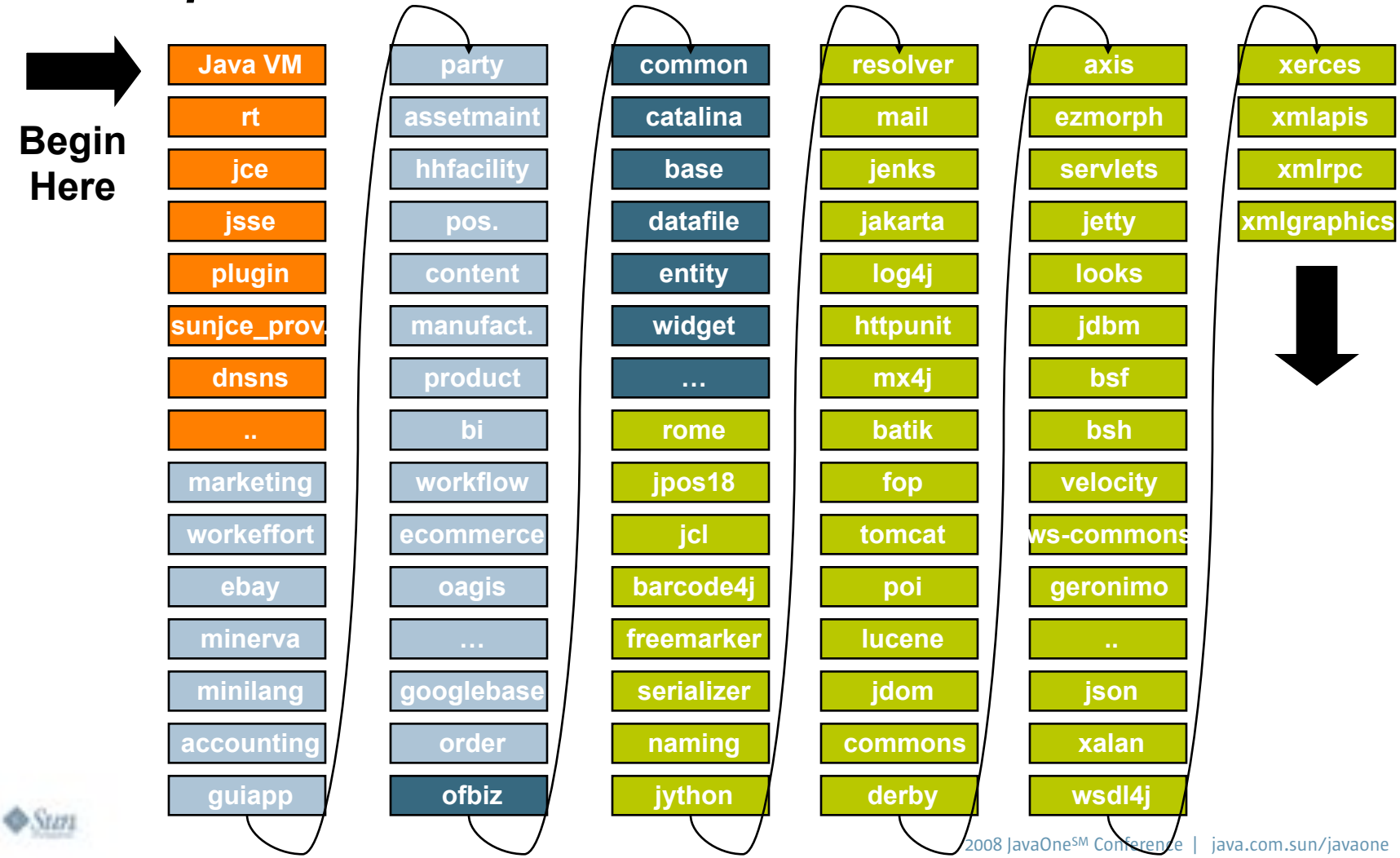
Modularization in Java Apps
Visibility

- Java Platform Modularity
  - Classes encapsulate data
  - Packages contain classes
  - Jars contain packages
- Visibility Access
  - private, package private, protected, public
- Packages look hierarchical, but are not
- Jars have no modularization characteristics



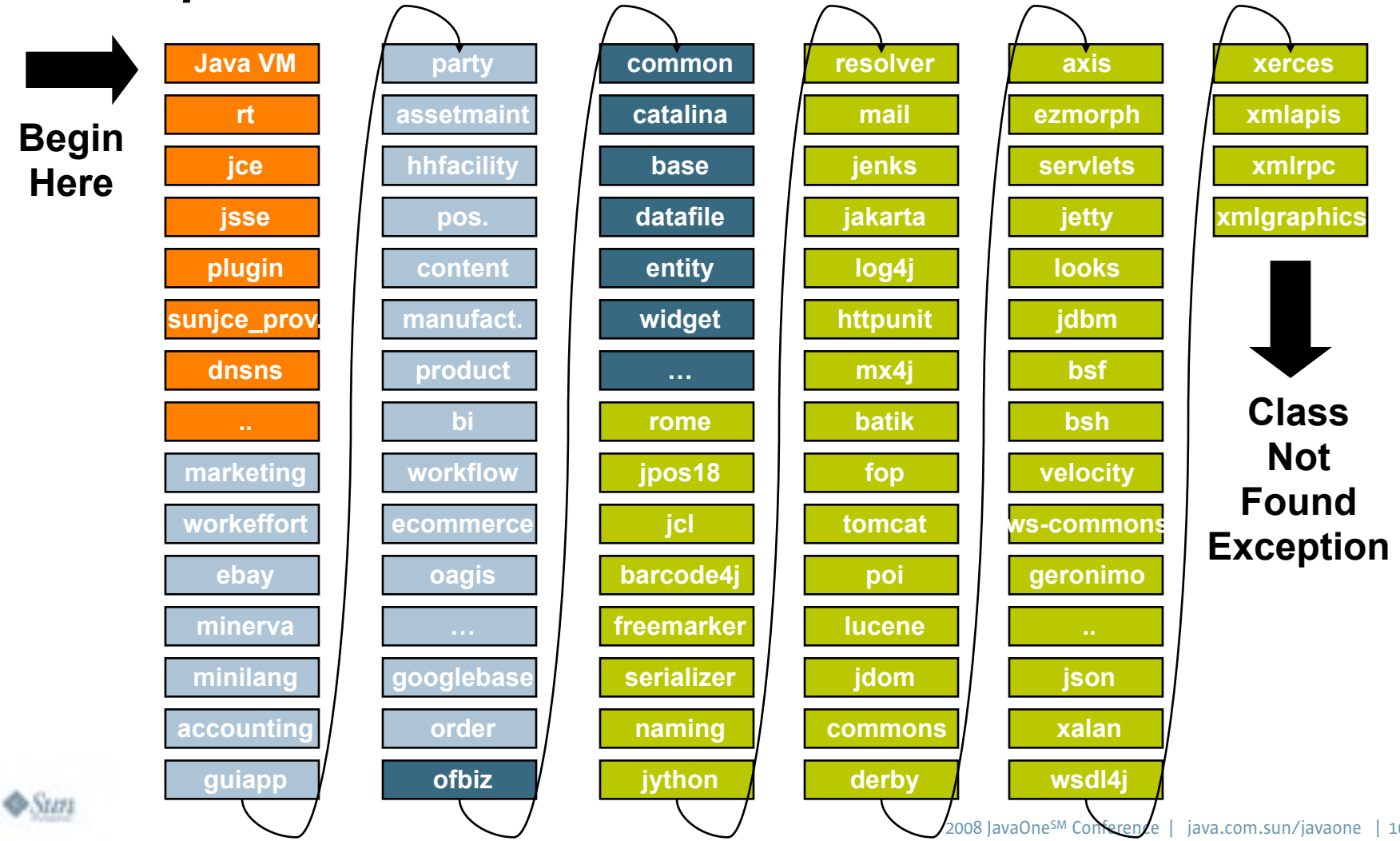
Modularization in Java Apps

Classpath



Modularization in Java Apps

Classpath

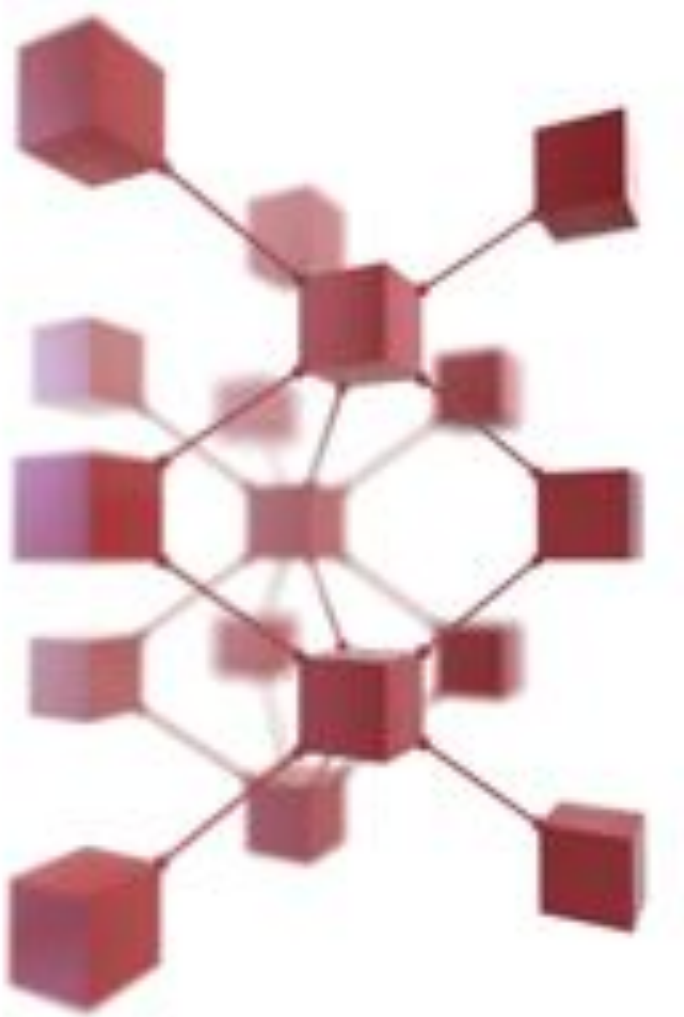


Modularization in Java Apps
Issues

- Granularity of classes and packages is too small for real world applications
- Jars provide packaging, can not be used to restrict access
  - Every public class is visible to every other class
- Severe problems like split packages
  - Multiple jars have classes in the same package
  - Often unintended
- No versioning support
  - Order on CLASSPATH define chosen version
  - Single version of a class in the VM
- Has no proper extension/collaboration model

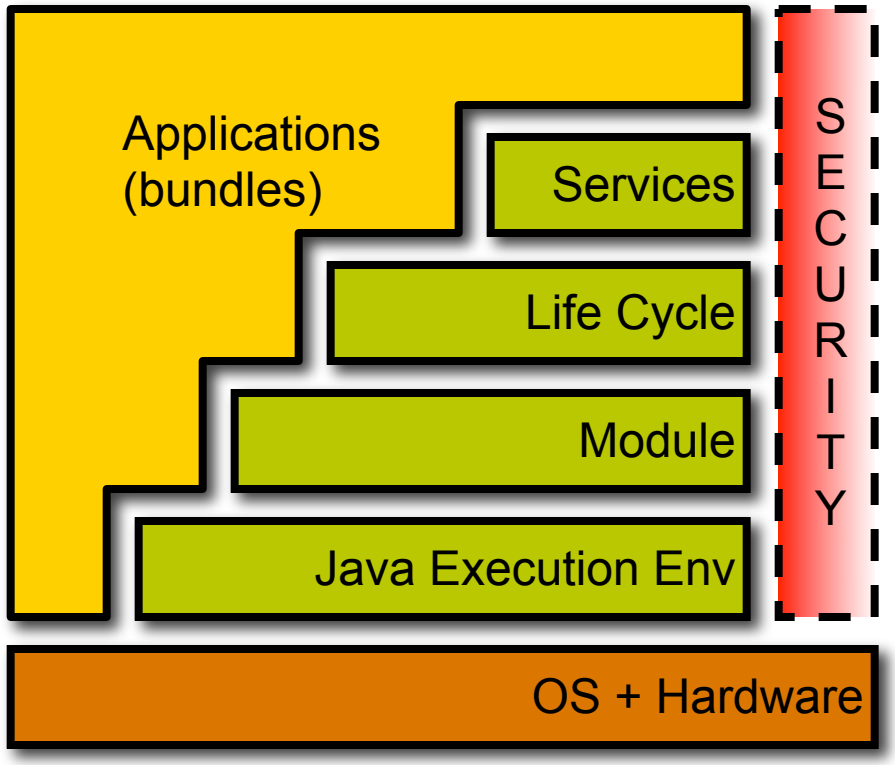
Agenda

- Modularization
- Modularization in Java™ Apps
- **The OSGi Framework**
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- Building
- Pitfalls
- Conclusion



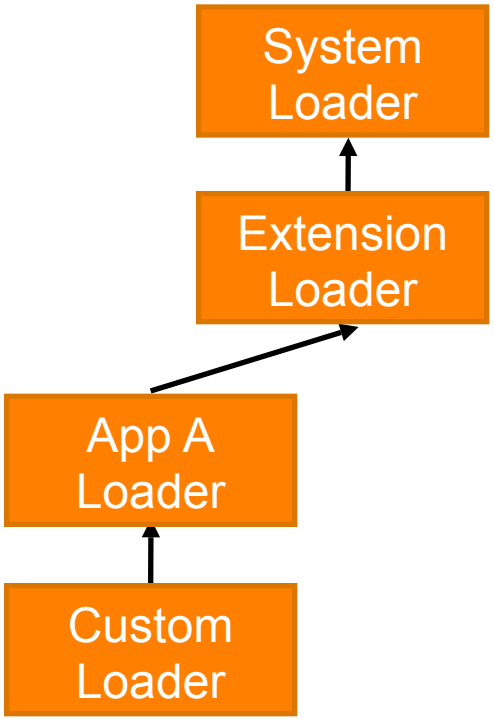
The OSGi Framework
Overview

- The OSGi Service Platform specifies a modular architecture for dynamic component based systems
  - Execution Environment
  - Module Layer
  - Life Cycle Layer
  - Service Layer
  - Security
- Introduces *Bundles* as modules



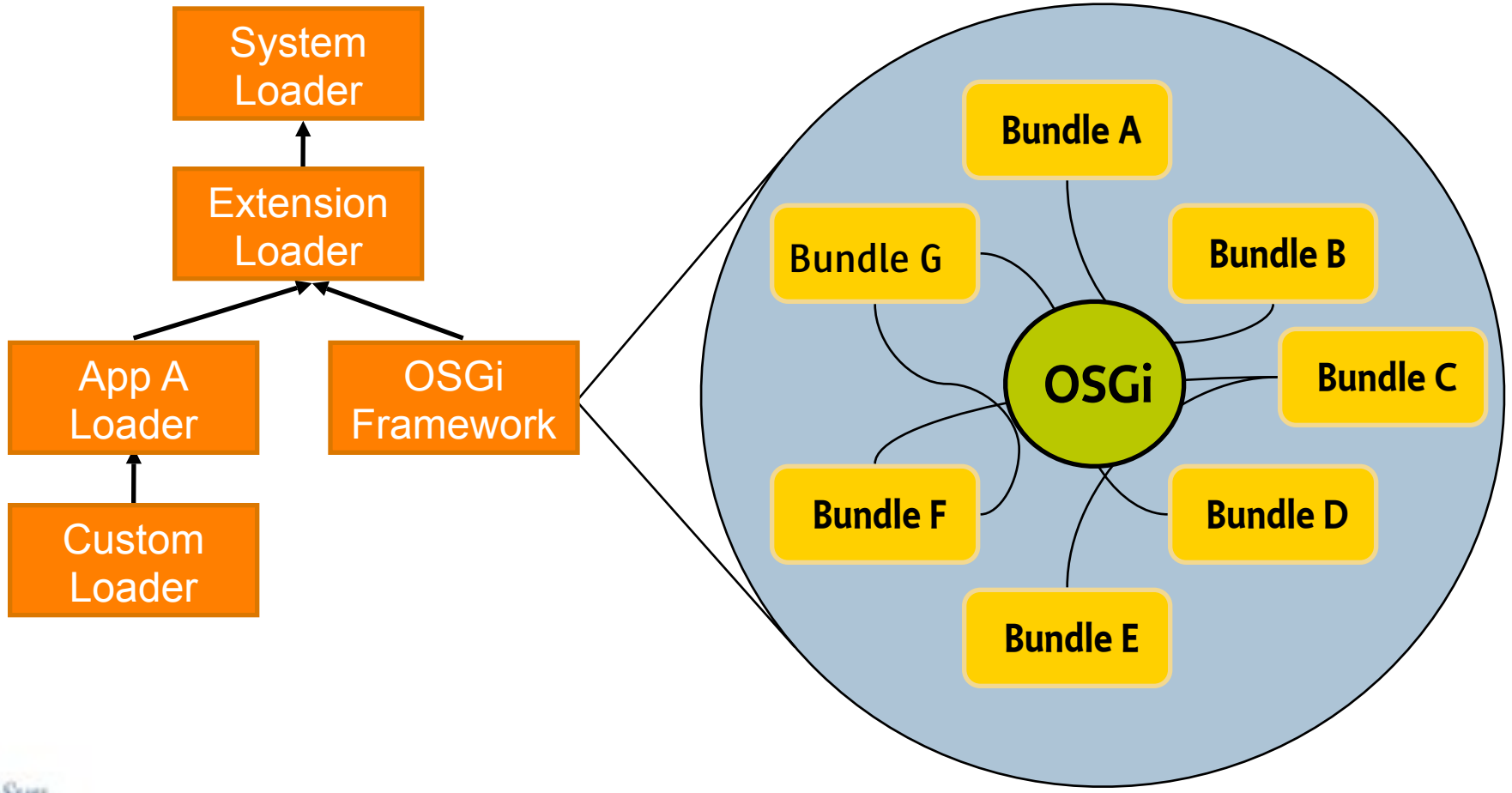
The OSGi Framework

Networked Class Loaders on "Steroids"



The OSGi Framework

Networked Class Loaders on "Steroids"



The OSGi Framework
Features

- Class loading dispatch based on package name
  - Prevents many problems with split packages
  - Faster class loading for large systems
- Allows multiple versions of the same class in one VM
  - Class spaces
- Jars can contain exported packages or Jar private packages
- Bundle == Jar
  - Manifest contains metadata
- Services provide a collaborative in-VM SOA model
  - No invocation overhead
  - Full Life Cycle Model

Agenda

- Modularization
- Java Platform Modularization
- The OSGi Framework
- **Legacy Code**
- Dynamic Class Loading
- Designing with Services
- Building
- Pitfalls
- Conclusion

Legacy Code

... legacy code is a challenge. Many developers say things like: "My code is very modular", or "My code doesn't depend on very much", or "No one uses any of my classes except from the Foo package"

Unless they are already using OSGi, they are wrong. Until modularity is enforced, ***it is not there****.*

John Wells, BEA

Legacy Code
Wrapping Libraries

- Analyzing what you have
  - What are the dependencies between the Jars that make up your applications
- OSGi bundles need manifest headers
  - Exported packages,
  - Imported Packages,
  - Optionality,
  - Versions,
  - Bundle identity
  - ...

Legacy Code
Wrapping Libraries

- Several open source projects provide OSGi metadata:
  - Apache (Derby, Struts, Felix, etc)
  - All Eclipse code
  - Codehaus Groovy
- Repositories are coming online:
  - OSGi Bundle Repository (OBR)
  - Apache Felix Commons
  - Eclipse Orbit
  - Maven is increasingly OSGi aware (see Maven Bundle plugin)
  - SpringSource Enterprise Bundle Repository
- When this fails: bnd utility
  - OSGi bundle analyze and build tool ...

Legacy Code Strategy?

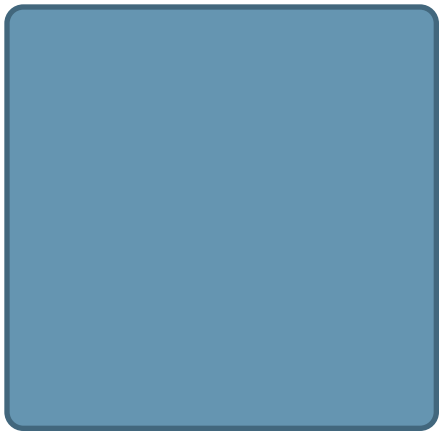

Legacy Code Strategy?
How do You Eat an Elephant?


Legacy Code Strategy?
How do You Eat an Elephant?
One Bite at a Time!

Legacy Code
Convert Existing Code

- Make a project with your application and all its dependent jars
  - Include all libraries on the classpath
- Create a bundle activator that calls main
- Include all libraries into one bundle
  - Super sized!
  - Use bnd
- Having a working bundle (whatever the size) is a good baseline and allows for gradually replacing the dependent jars with bundles
  - Keep it working!

Legacy Code
Converting Existing Code



Bundle

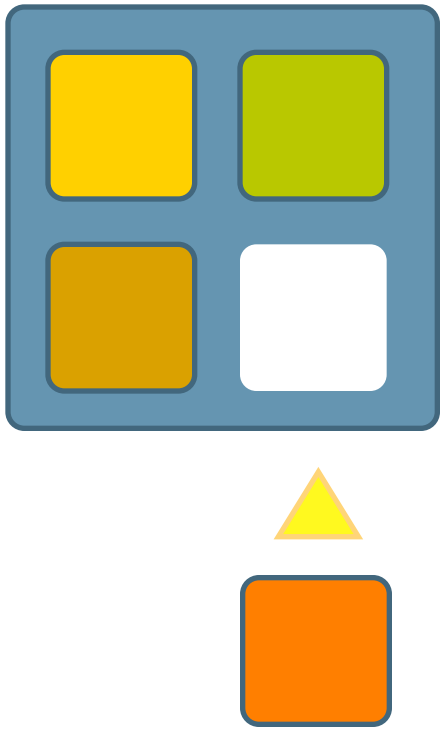
Service

Legacy Code
Converting Existing Code

Bundle

Service

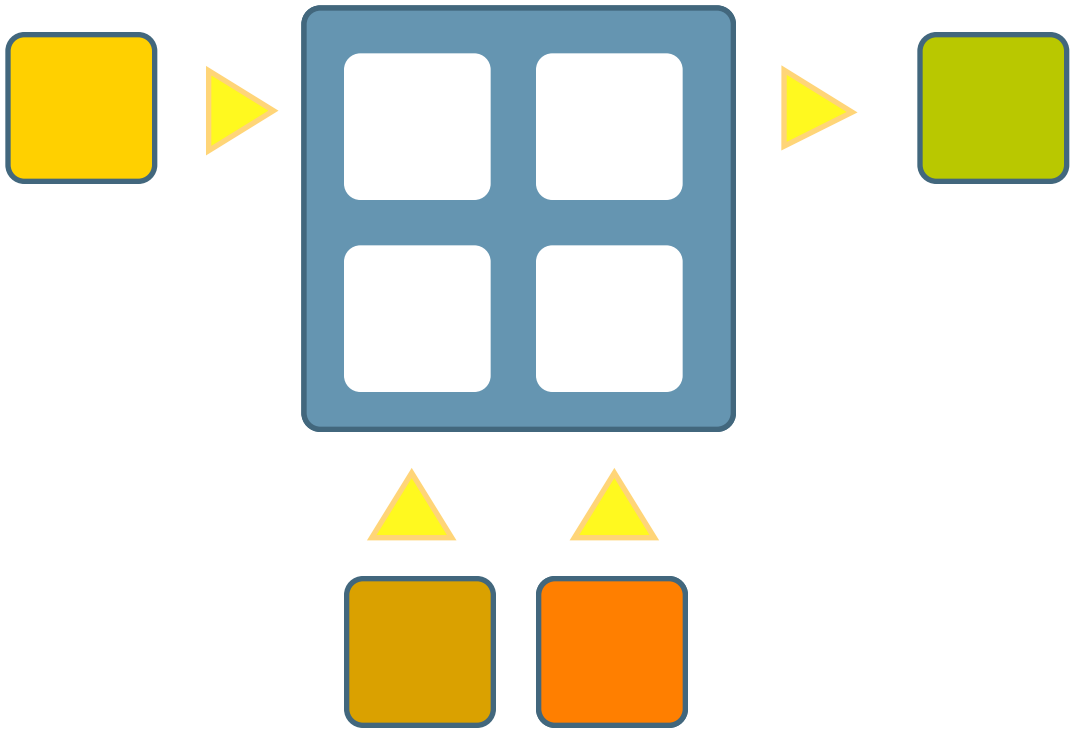
Legacy Code
Converting Existing Code



Bundle

Service

Legacy Code
Converting Existing Code



Legacy Code
Breaking out Bundles

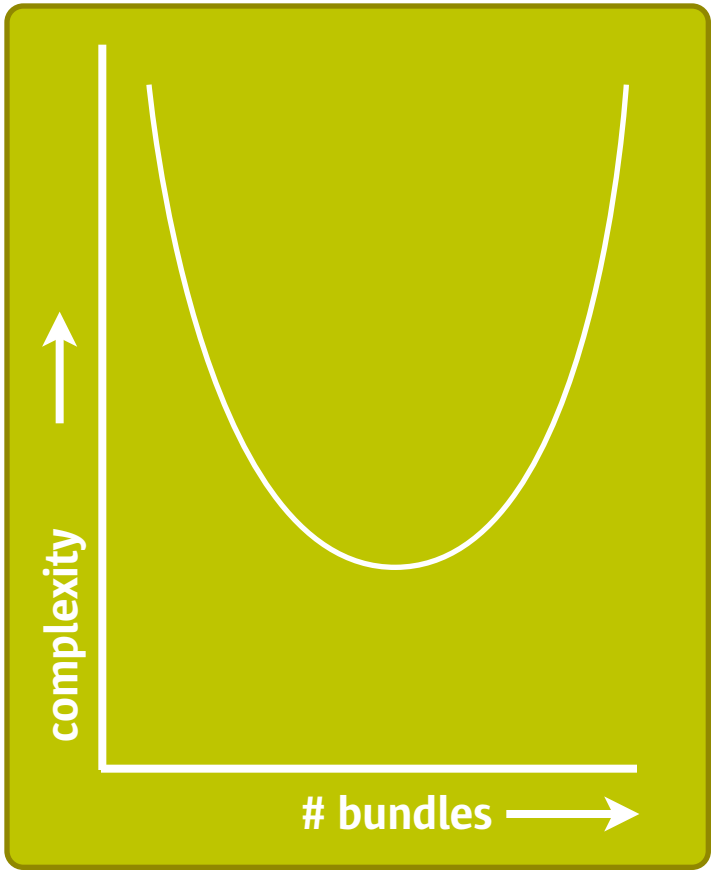
- For each dependent jar in the project, call it x.jar
  - If there is an existing bundleized version of that jar
    - use that one
  - Otherwise
    - Create a new bundle project corresponding only to x.jar file.
    - Find the Java source files for x.jar in the old projects and move them into the new source folder.
- The new application bundle project has no more jar libraries or source in it.

Legacy Code
Application Models

- Raw OSGi APIs are powerful but not always easy to use
- Best practices is to write POJOs (Plain Old Java Objects) that are not coupled to a framework
- There are many such application models for OSGi:
  - Spring-DM (formerly called Spring-OSGi)
  - Apache iPOJO
  - Service Application Toolkit
  - OSGi Declarative Services
- Pick one of these application models after you got your existing application running
- Convert the application part-by-part

Legacy Code
To Include or to Refer? That is the Question!

- When to refer to a library?
  - External API
  - Implementations can differ
    - E.g. javax.naming, javax.transaction
  - Very large library
  - Reduces size because of sharing
- When to include a library in the bundle?
  - Pure functions
  - Small
  - Reduces number of dependencies
  - High Cohesion



Legacy Code
Use case: Hibernate

```
> bnd print -impexp hibernate3.jar
```

```
antlr.…                                com.mchange.v2.c3p0
com.opensymphony.oscache.…             javassist.…
javax.naming.…                         javax.security.…
javax.sql                              javax.transaction.…
net.sf.cglib.…                         net.sf.ehcache
net.sf.swarmcache                      org.apache.commons.…
org.apache.tools.ant.…                 org.dom4j.…
org.jboss.cache.…                      org.logicalcobwebs.proxool.…
org.objectweb.asm.…                    org.w3c.dom
org.xml.sax
```

Legacy Code
Create org.hibernate.bnd File

```
-classpath: hibernate3.jar, \
    lib/antlr-2.7.6.jar, \
    lib/asm-attrs.jar, \
    lib/asm.jar, \
    lib/cglib-2.1.3.jar, \
    lib/commons-collections-2.1.1.jar, \
    lib/commons-logging-1.0.4.jar, \
    lib/dom4j-1.6.1.jar, \
    lib/log4j-1.2.11.jar, \
    lib/jta.jar
```

Legacy Code
Wrapping Hibernate

- Minimize dependencies!
  - Managing dependencies is good!
  - Not having dependencies is best!
- Let bnd do:
  - Copy all packages on the classpath
  - Import the missing packages
- Only export the hibernate packages and the javax.transaction packages from jta.jar
- Other libraries should be kept private
- This is a first guess, when we start to use hibernate, we might learn that we want to export more packages

Legacy Code
Next version of bnd file: org.hibernate.bnd

```
-classpath: hibernate3.jar, \
    lib/antlr-2.7.6.jar, \
    lib/asm-attrs.jar, \
    lib/asm.jar, \
    lib/cglib-2.1.3.jar, \
    lib/commons-collections-2.1.1.jar, \
    lib/commons-logging-1.0.4.jar, \
    lib/dom4j-1.6.1.jar, \
    lib/log4j-1.2.11.jar, \
    lib/jta.jar

Private-Package: *

Export-Package: javax.transaction.*, \
                org.hibernate.*
```

Legacy Code
View imported packages

```
> bnd org.hibernate.bnd
> bnd print -impexp org.hibernate.jar

com.mchange.v2.c3p0                com.opensymphony.oscache…
com.sun.jdmk.comm                  com.sun.msv.datatype…
javassist…                         javax.jms
javax.mail…                        javax.management
javax.naming…                      javax.security…
javax.sql                          javax.swing…
javax.xml…                         net.sf.ehcache
net.sf.swarmcache      org.apache.avalon.framework.logger
org.apache.log                     org.apache.tools.ant…
org.codehaus.aspectwerkz.hook      org.gjt.xpp
org.jaxen…                         org.jboss.cache…
org.logicalcobwebs.proxool…        org.objectweb.asm.util
org.relaxng.datatype               org.w3c.dom
org.xml.sax…
```

Legacy Code
Wrapping Hibernate

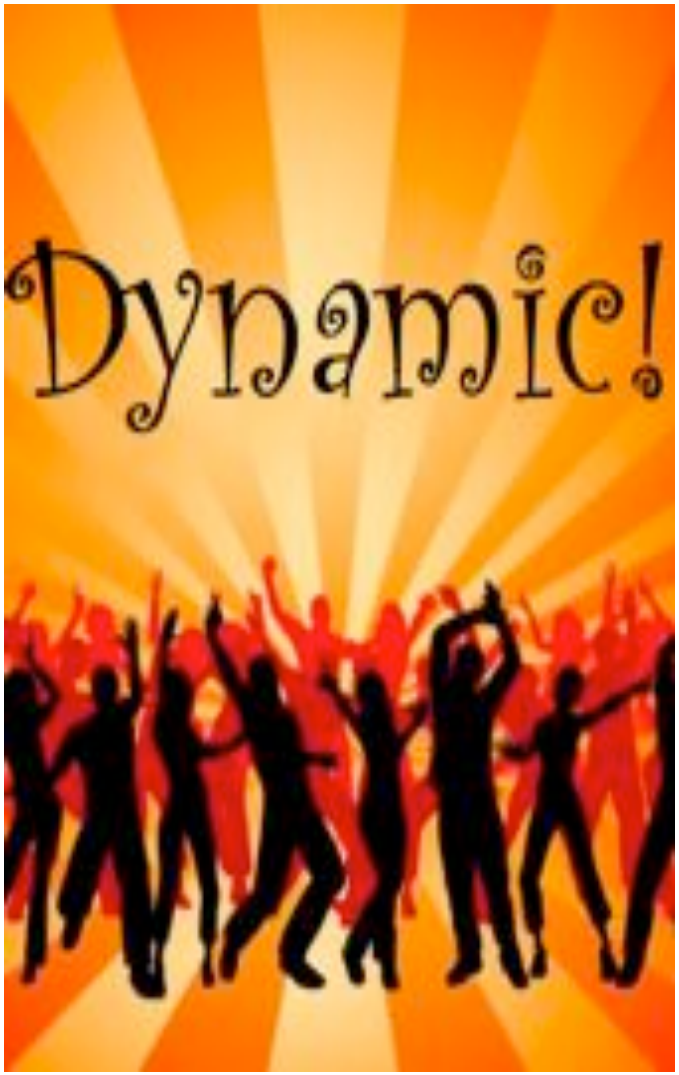
- There are many packages which are obviously not mandatory.
  - org.apache.avalon.framework.logger
  - org.apache.tools.ant
- These are *glue dependencies*. They are only needed when running inside some “framework”
- There are also optional features that should not be required by hibernate
  - for example javax.mail?
- A shortcut is to make all our imports optional
  - Detects problems in runtime
  - With more time and knowledge, it is possible to make this less coarse grained

Legacy Code
Next version bnd file org.hibernate.bnd

```
-classpath: hibernate3.jar, \
    lib/antlr-2.7.6.jar, \
    lib/asm-attrs.jar, \
    lib/asm.jar, \
    lib/cglib-2.1.3.jar, \
    lib/commons-collections-2.1.1.jar, \
    lib/commons-logging-1.0.4.jar, \
    lib/dom4j-1.6.1.jar, \
    lib/log4j-1.2.11.jar, \
    lib/jta.jar

Private-Package: *

Import-Package: javax.xml.*, javax.sql.*, \
                *;resolution:=optional

Export-Package: javax.transaction.*,org.hibernate.*
```

Agenda

- Modularization
- Modularization in Java™ Apps
- The OSGi Framework
- Legacy Code
- **Dynamic Class Loading**
- Designing with Services
- Building
- Pitfalls
- Conclusion



Dynamic Class Loading

- A surprising number of applications created their own unique plugin mechanisms using dynamic class loading
  - Configuration provided by strings
  - Class.forName during runtime
- Ignores modularity …
  - Class name strings never, ever, handle versions
- Class.forName has a bug that prevents it from handling these issues correctly
  - At least, use ClassLoader.loadClass

Dynamic Class Loading

- Context Class loader tends to revert back to “best effort” linear search
  - Eclipse supports “Buddy Class Loading” (Containerism!)
  - OSGi next specification will address the use cases for Buddy Class Loading
- Recommendations
  - Try to remove all class loading code. Is it really necessary?
  - Convert plugin mechanisms to services because it handles versions and compatibility issues
  - If not possible, use ClassLoader methods, not Class.forName
- Relax, let go of your class loaders, really OSGi does it better :-)

Agenda

- Modularization
- Modularization in Java™ Apps
- The OSGi Framework
- Legacy Code
- Dynamic Class Loading
- **Designing with Services**
- Building
- Pitfalls
- Conclusion

Designing with Services

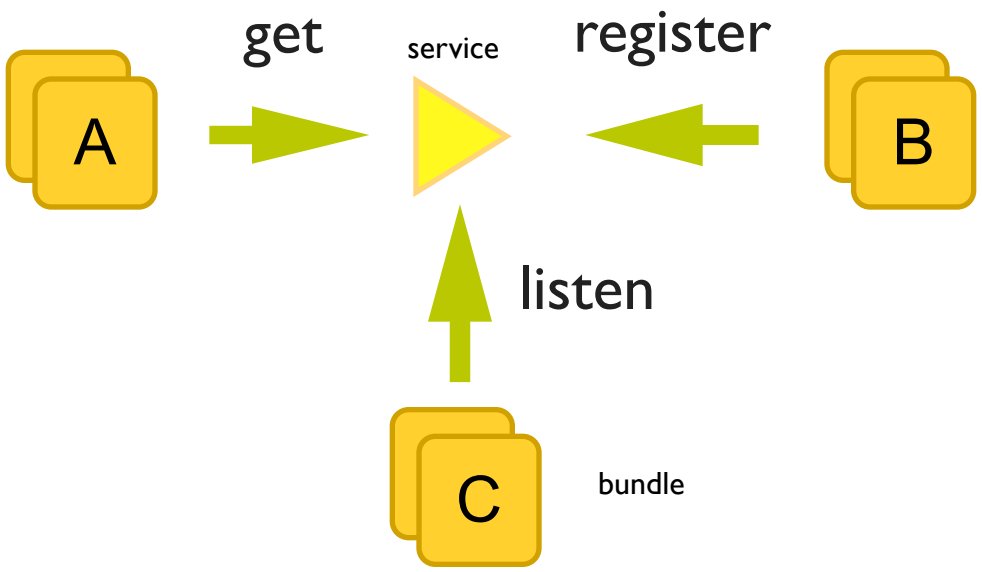
One big issue for good component and service oriented architectures is how to achieve loose coupling between components by design. As architects we are looking for minimization of dependencies and maximization of flexibility, preparing a system for future change. The OSGi framework is a great foundation for this endeavor, ...

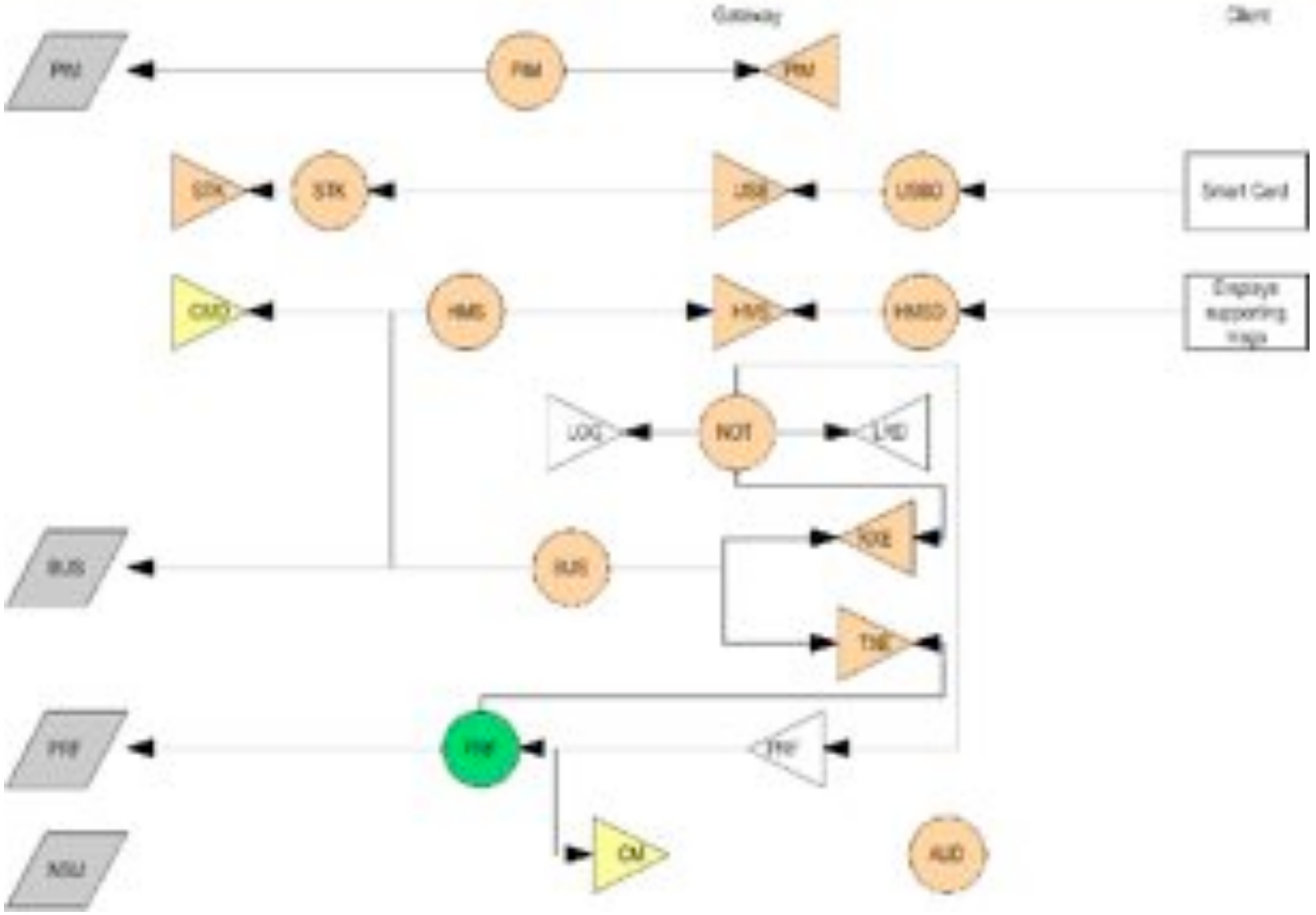
Dieter Wimberger

Designing with Services
Goals

- Services provide a very loose coupling between modules
  - Allows modules to be substituted
- Services are used as input/output ports, they *should* be the *only* links between bundles
- Key is to find connections between modules and map them to services
  - Initially, this is not always obvious so do not go overboard
- Try to use standard services (OSGi)
- Try to use standard interfaces (Java application environment, ...)
- Try to establish in-house standards

Designing with Services
Service Concepts




Gateway
Client
PIU
PIM
PIM
SDK
SIK
USB
USBD
Smart Card
CMD
HMI
HMI
HMID
Displays supporting logo
LOG
NOT
LID
BUS
BUS
SDE
TSE
PIF
PIF
PIF
CM
AUD
NSU

Agenda

- Modularization
- Modularization in Java™ Apps
- The OSGi Framework
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- **Building**
- Pitfalls
- Conclusion

Building

- Your Build system is probably the largest productivity multiplier in your organization
- Eclipse is an excellent IDE for OSGi development
  - PDE - Plugin Development Environment, Pax-Runner, JDT + Bnd
- But bundles can also be built with Netbeans™ software or other IDEs
- Offline build is different, PDE build is not very easy to run offline
- Build Systems
  - Maven + Bundle Plugin (from Apache Felix, based on bnd)
  - Ant + bnd

Agenda

- Modularization
- Modularization in Java™ Apps
- The OSGi Framework
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- Building
- **Pitfalls**
- Conclusion

Pitfalls

- Too much at once
  - Get the complete application to work on OSGi, modularize a part, test, and iterate in small increments
  - Eat the elephant one bite at a time
- Dynamic Class Loading
  - Most of the time custom class loaders are abused in applications, get rid of them
  - Watch out for the usage of Class.forName
  - Use Services

Pitfalls

- Framework Implementation Dependencies
  - “Containerisms”
  - Be careful to use OSGi only features and not become dependent on a specific OSGi implementation
  - Almost all problems can be properly solved using the OSGi capabilities
  - In rare cases, it is necessary to escape to an implementation dependent feature, e.g. Buddy loading

Agenda

- Modularization
- Java Platform Modularization
- The OSGi Framework
- Legacy Code
- Dynamic Class Loading
- Designing with Services
- Building
- Pitfalls
- **Conclusion**

Conclusion

Conclusion

To this day team members still come up to me occasionally to thank me for introducing OSGi, often after being reminded what things were like by having to go back to an old release build. I wish I could take credit but the truth is I never anticipated most of these benefits until I started the conversion.

Bill Kayser, Software Architect StepZero LLC

References

- OSGi Alliance
  - http://www.osgi.org
- bnd
  - http://www.aQute.biz/Code/Bnd
- Bill Kayser's blog about converting to OSGi
  - http://blogs.nagarro.net/kayser/osgi-from-here-to-there-part-ii
- Bundleizing Hibernate
  - http://www.aqute.biz/Code/BndHibernate
- BEA The Good, the Bad, and the Ugly
  - http://www.parleys.com/display/PARLEYS/OSGi,+the+good+the+bad+the+ugly
- Dieter Wimberger Blog
  - http://www.blogger.com/profile/01341177121570488166

THANK YOU

Join us in Berlin, June 10-11, 2008
for the 2008 OSGi Community Event!

Java

Sun