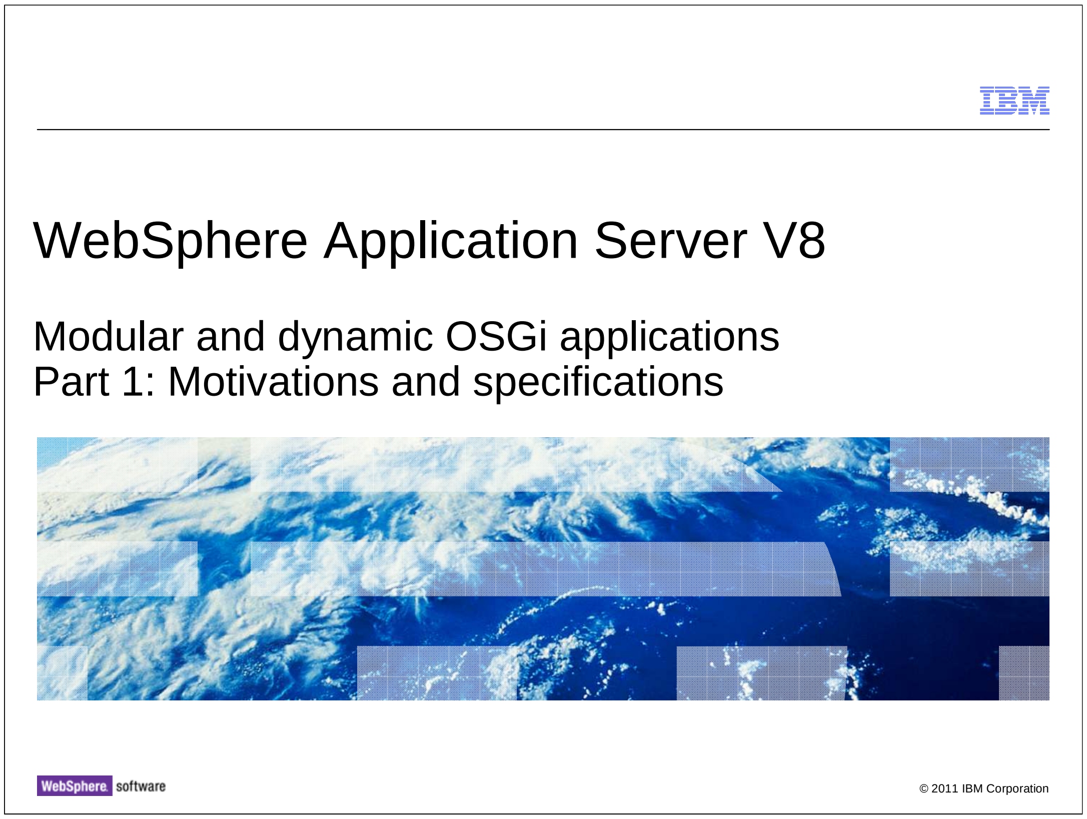
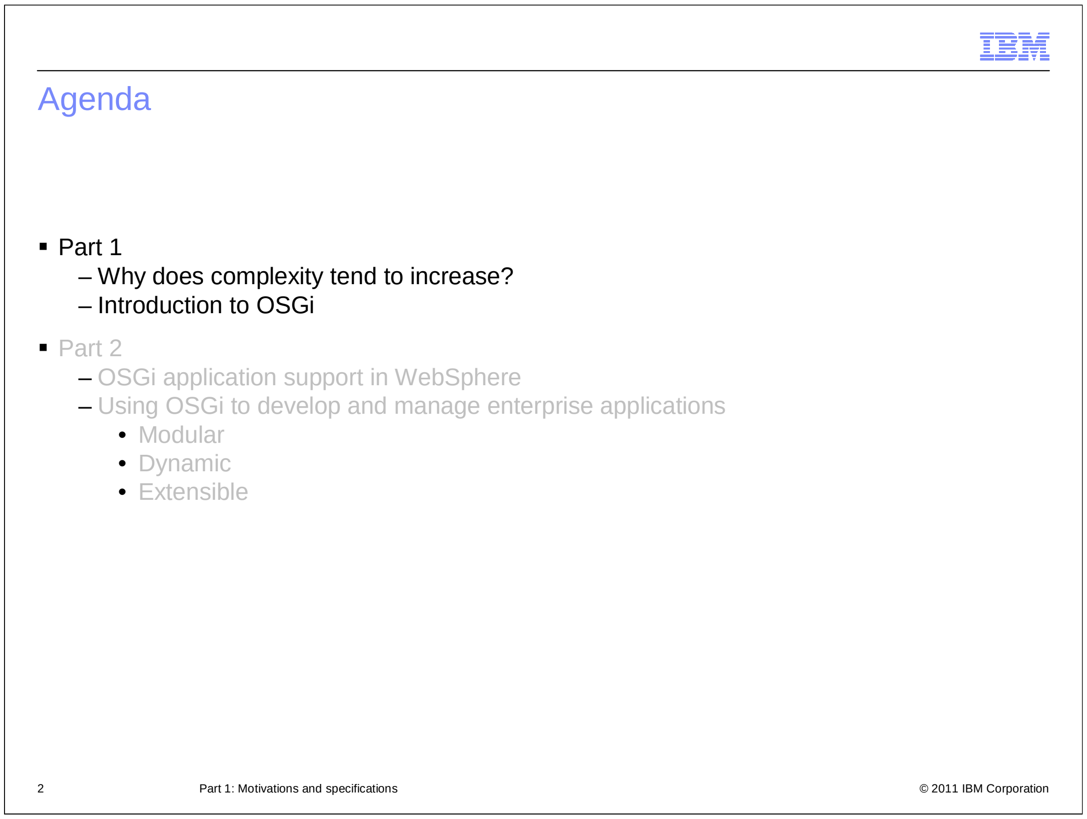


WebSphere Application Server V8

Modular and dynamic OSGi applications Part 1: Motivations and specifications

WebSphere software

This presentation covers modular and dynamic OSGi applications in WebSphere® Application Server. This is Part 1 - motivations and specifications.



Agenda

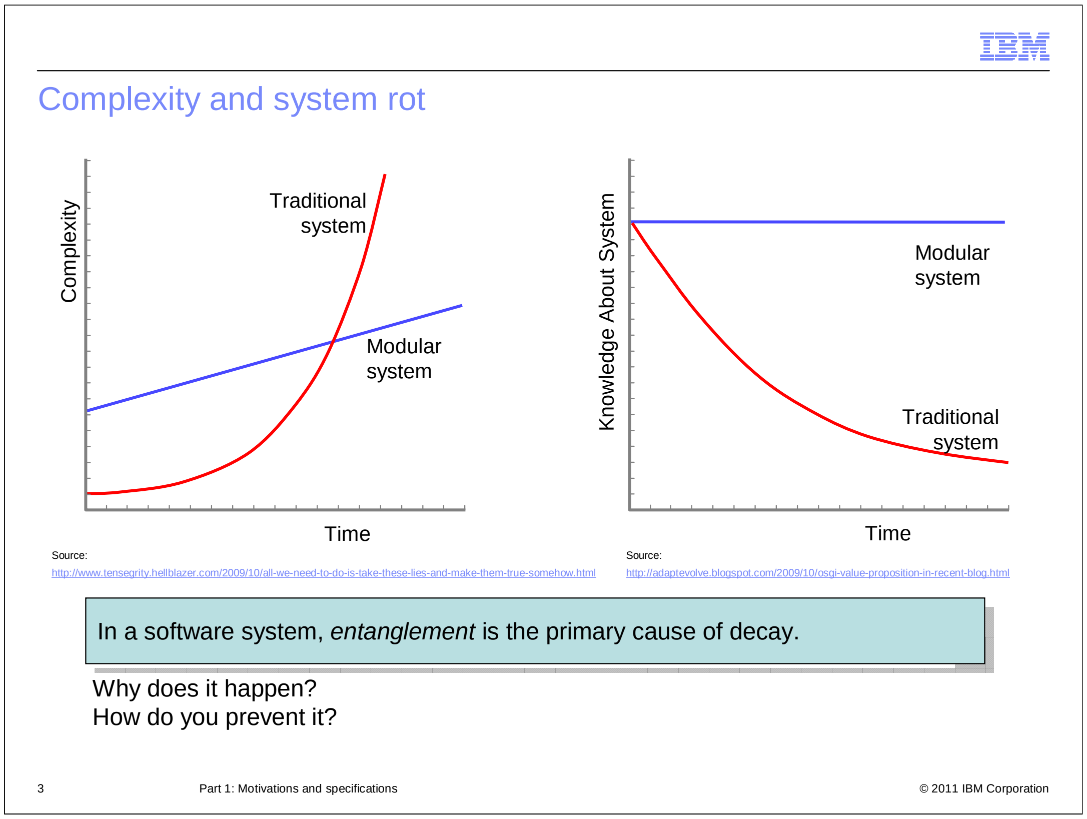
- Part 1
  - – Why does complexity tend to increase?
  - – Introduction to OSGi
- Part 2
  - – OSGi application support in WebSphere
  - – Using OSGi to develop and manage enterprise applications
    - Modular
    - Dynamic
    - Extensible

2 Part 1: Motivations and specifications © 2011 IBM Corporation

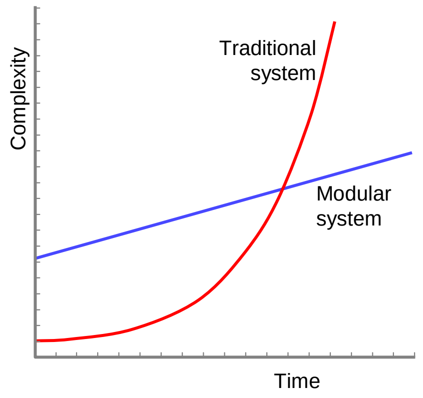
This is the first part of a two part series that covers the WebSphere Application Server OSGi Applications feature, introduced in version 7 and extended in version 8.

Part 1 describes some common problems experienced with Java™ Enterprise Edition (or Java EE for short) application deployments to understand what problems this feature solves.

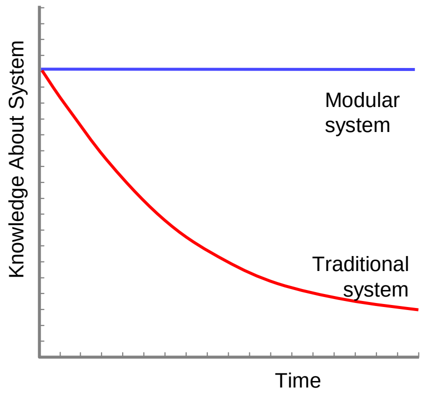
Then it describes how OSGi and Java Enterprise Edition have come together over the last two years in standards and in open-source. The second part looks at how this is implemented in WebSphere Application Server.



Complexity and system rot



Source:
http://www.tensegrity.hellblazer.com/2009/10/all-we-need-to-do-is-take-these-lies-and-make-them-true-somehow.html



Source:
http://adaptevolve.blogspot.com/2009/10/osgi-value-proposition-in-recent-blog.html

In a software system, *entanglement* is the primary cause of decay.

Why does it happen?
How do you prevent it?

3 Part 1: Motivations and specifications © 2011 IBM Corporation

Systems evolve over time to fix problems and meet new and changing requirements. As a result, systems become complex. If the system was well-architected to start with then the complexity can be better managed and is more likely to grow in a linear fashion because first, changes to module internals are isolated only to that module, and second, the impact on the system of a change to one module is well understand and so testing can be more easily targeted.

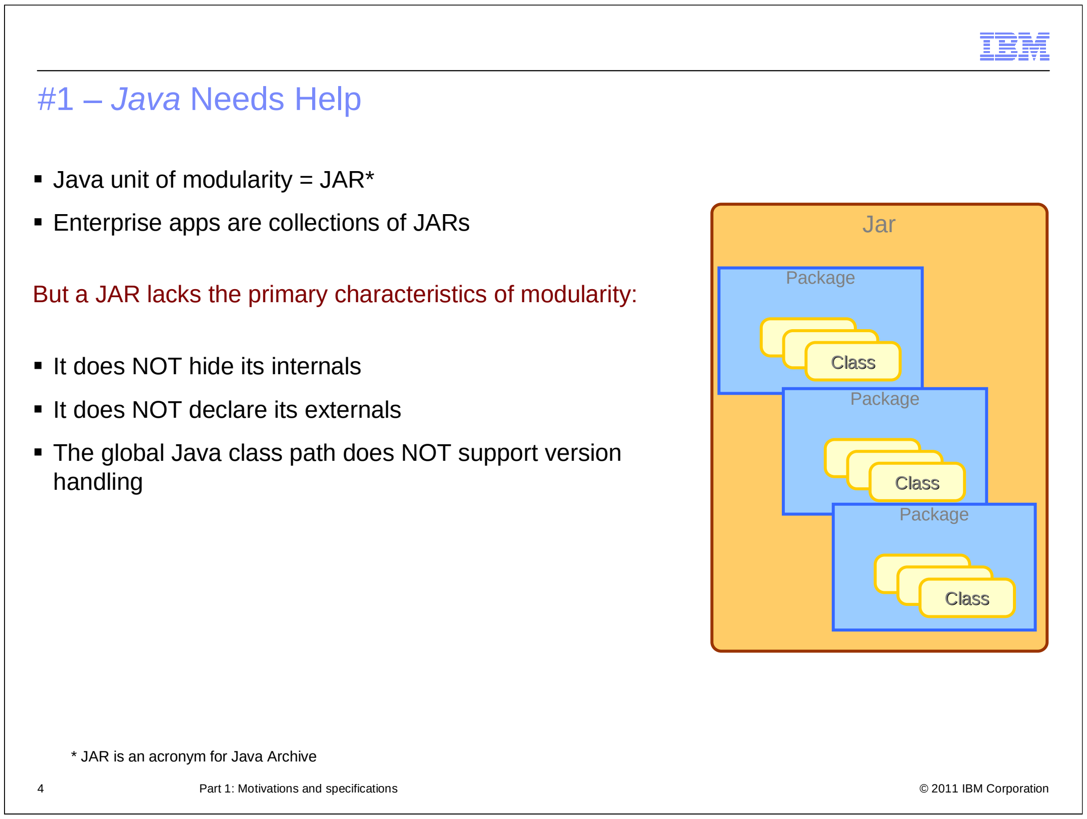
“Entangled” systems that lack structure evolve in a more uncontrolled fashion such that with each release, changes of a particular scale become increasingly expensive to perform. This is due to the “compound interest” of the entangled growth of previous releases. The impact of any change subsequently becomes difficult to estimate and targeting the testing of that change becomes more of an art than a science.

There is an up-front cost to designing and implementing a modular system. There are some concepts and processes that support the strictly modular behavior that developers have to understand; in contrast, the initial costs at the beginning of the life cycle of a traditional system are lower and initial progress will likely be faster. But as time passes and additional subsystems are introduced, the initial investment pays off and keeps on giving.

Another way at looking at the benefit of a well modularized system is in terms of how well structural knowledge about the system is retained over time. The problem faced with many complex systems is that they become subject to “system rot”. Over time, structural knowledge is lost as key developers and architects leave the organization. These key people were relied upon when documentation was inadequate or missing. It then becomes increasingly difficult to effectively re-factor the system in response to changing business requirements and accidental complexity is introduced over time as non-optimal changes are made.

Structural knowledge is not lost from a modular system even as the systems grows over time because the relationships between the system modules are well-defined and module internals are fully encapsulated and do not leak out.

The common theme here is that software systems have a tendency to decay. The primary cause of that decay is the increasing entanglement of the components that make up that system.

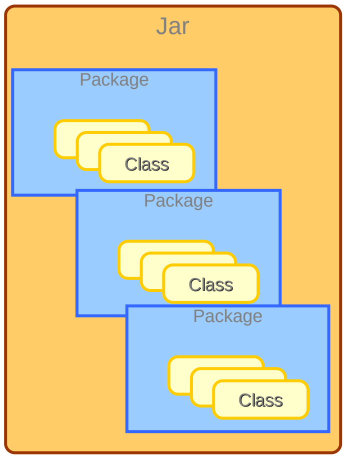


#1 – *Java* Needs Help

- Java unit of modularity = JAR*
- Enterprise apps are collections of JARs

But a JAR lacks the primary characteristics of modularity:

- It does NOT hide its internals
- It does NOT declare its externals
- The global Java class path does NOT support version handling



* JAR is an acronym for Java Archive

4 Part 1: Motivations and specifications © 2011 IBM Corporation

In the Java programming language there are three primary areas to reduce the occurrence of software entanglement and to increase the ability of developers and operations staff to deliver and maintain agile, dynamic systems.

The first is Java itself.

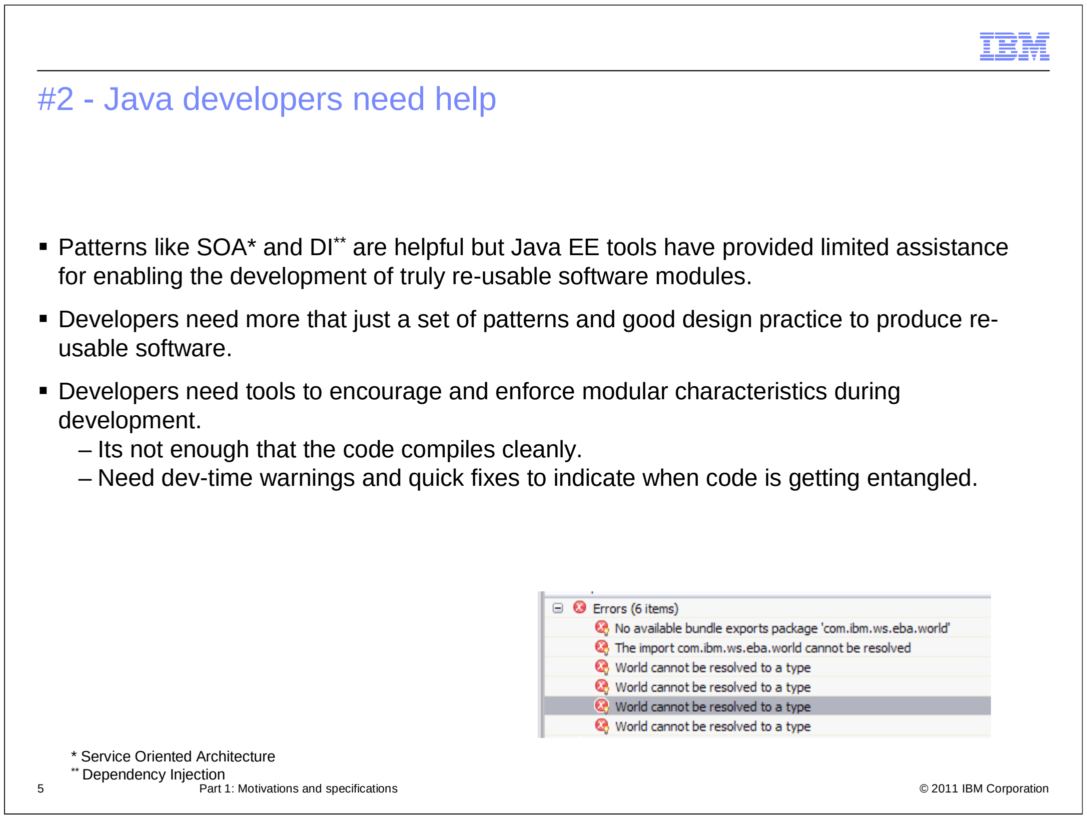
This is where most of the problems start. Object oriented languages like Java provide for modularity at the instance data level, which is a good foundation, but stops short where it really matters. The currency of modularity in any system is the “unit of deployment” and in Java the JAR is the module of deployment: the ideal granularity at which to consider module reuse. But JARs have no modularity characteristics.

There is no “jar scoped” access modifier alongside public, protected and private.

Most JARs consist of multiple packages and, if the JAR represents a cohesive function, there is typically a need for classes in one package to access classes in another which, as a consequence is required to have public accessibility. Immediately that makes these classes visible to classes in any other JAR due to JARs providing no level of visibility control. Even well-behaved applications that only use the classes a JAR provider expects to be used externally are at the mercy of the global Java class path because the desired class could be contained by multiple jars and the one loaded is simply the first on the global class path.

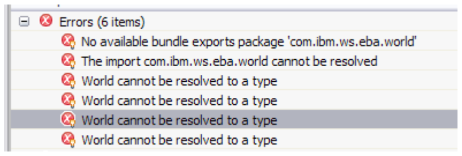
Not only do JARs lack the capacity to scope the visibility of what they contain, they also lack the capacity to declare their own dependencies. Many jars have implicit dependencies on other jars that means these jars cannot be installed or moved around independently. If they are installed without dependencies being present then the first time there is any indication of a problem is at runtime.

Another problem with the global Java class path is its inability to accommodate multiple versions of a class. There can be multiple versions available on the class path but only the first will ever be loaded.



#2 - Java developers need help

- Patterns like SOA* and DI** are helpful but Java EE tools have provided limited assistance for enabling the development of truly re-usable software modules.
- Developers need more that just a set of patterns and good design practice to produce re-usable software.
- Developers need tools to encourage and enforce modular characteristics during development.
  - Its not enough that the code compiles cleanly.
  - Need dev-time warnings and quick fixes to indicate when code is getting entangled.



```
Errors (6 items)
    No available bundle exports package 'com.ibm.ws.eba.world'
    The import com.ibm.ws.eba.world cannot be resolved
    World cannot be resolved to a type
    World cannot be resolved to a type
    World cannot be resolved to a type
    World cannot be resolved to a type
```

* Service Oriented Architecture
** Dependency Injection

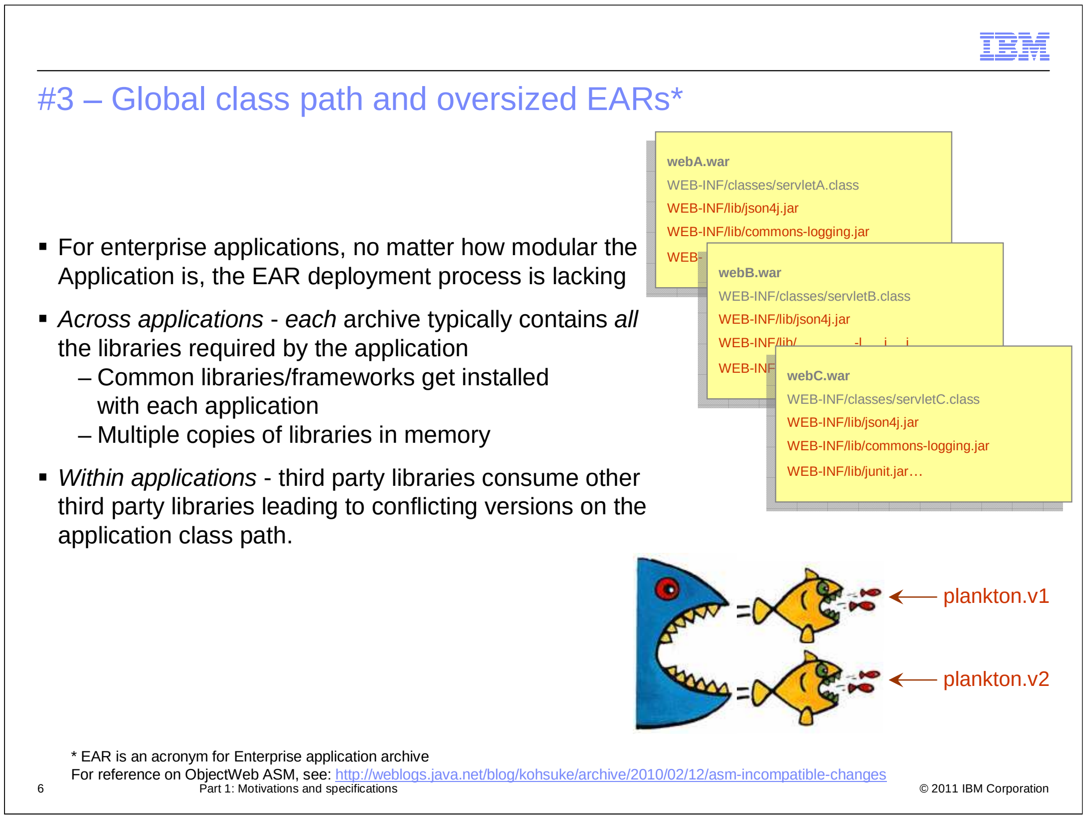
5 Part 1: Motivations and specifications © 2011 IBM Corporation

Java is a good starting point. Developers can further apply patterns like:

Service Oriented Architecture, which helps by defining external contracts and policies while hiding the implementation, and

Dependency injection, which helps by more loosely coupling fine-grained components and simplifying the testing of these component in isolation from one another, and the enterprise environments to which they are deployed.

But historically, while Java development tools have enabled and simplified the use of such patterns, they have done little to really encourage or enforce modular characteristics during development. Today, it is easy to build an enterprise application from a set of projects which represent the modules in the application. Integrated Development Environments tools quickly show developers where compilation errors have occurred and often suggest candidate fixes for those errors. What the tools are not so focused on is distinguishing between the internals, and externals of a project or providing warnings if project internals “bleed out”. Traditional Java tools simply build a project-specific Java class path such that if Project A needs a public class from Project B then Project B is part of the build path of Project A. In which case everything from Project B is part of the build path of Project A. It is all or nothing.



IBM

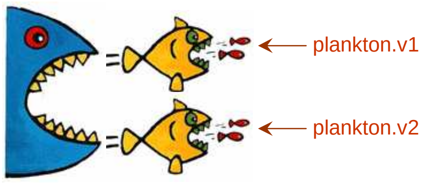
#3 – Global class path and oversized EARs*

- For enterprise applications, no matter how modular the Application is, the EAR deployment process is lacking
- *Across applications* - *each* archive typically contains *all* the libraries required by the application
  - – Common libraries/frameworks get installed with each application
  - – Multiple copies of libraries in memory
- *Within applications* - third party libraries consume other third party libraries leading to conflicting versions on the application class path.

webA.war
WEB-INF/classes/servletA.class
WEB-INF/lib/json4j.jar
WEB-INF/lib/commons-logging.jar
WEB

webB.war
WEB-INF/classes/servletB.class
WEB-INF/lib/json4j.jar
WEB-INF/lib/
WEB-INF

webC.war
WEB-INF/classes/servletC.class
WEB-INF/lib/json4j.jar
WEB-INF/lib/commons-logging.jar
WEB-INF/lib/junit.jar…



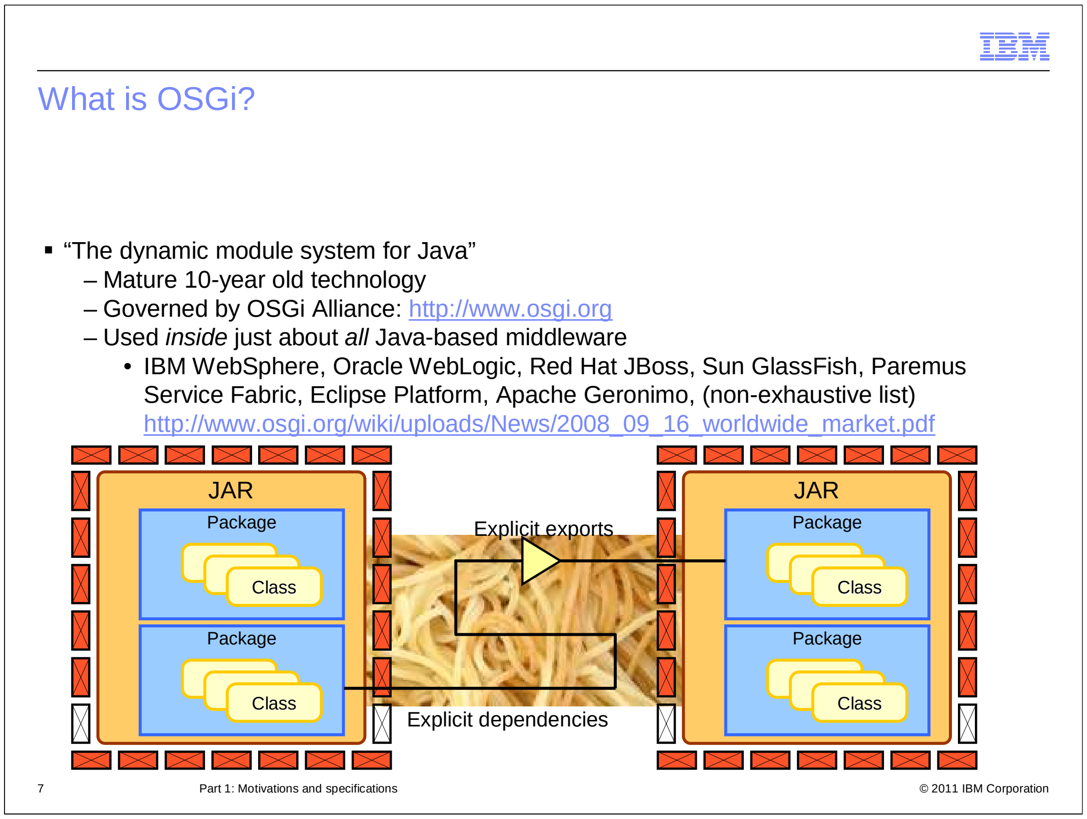
* EAR is an acronym for Enterprise application archive
For reference on ObjectWeb ASM, see: http://weblogs.java.net/blog/kohsuke/archive/2010/02/12/asm-incompatible-changes

6 Part 1: Motivations and specifications © 2011 IBM Corporation

And finally there is Java Enterprise Edition where the deployer, more than anyone else, really needs help. You can follow all the design best practices in the world when developing an enterprise application but the JEE deployer has EARs to work with.

Enterprise applications often make use of third-party Java libraries, either from open source or from an Independent Software Vendor who provides the application. Common examples are Apache Commons libraries, Spring, Hibernate and so on. The simplest way to ensure the coherency of each application is to include all the libraries each application needs in each EAR (or Enterprise Archive). While this makes it easier for EARs to be moved around from one system to another, it also makes for big EARs and multiple copies of the same library in memory for each application.

In addition, within an application you can still only have one version of each class, which can easily become problematic when multiple third-party libraries have dependencies on incompatible versions of some common utility class. The “ObjectWeb ASM” project is a good example of this kind of problem. ASM is a Java byte code manipulation and analysis framework used by many Java frameworks and has made non-compatible changes during its history. The result is, if you have a web application using two Java frameworks both of which need different versions of ASM then your application will not run even though it makes no direct use itself of ASM.



What is OSGi?

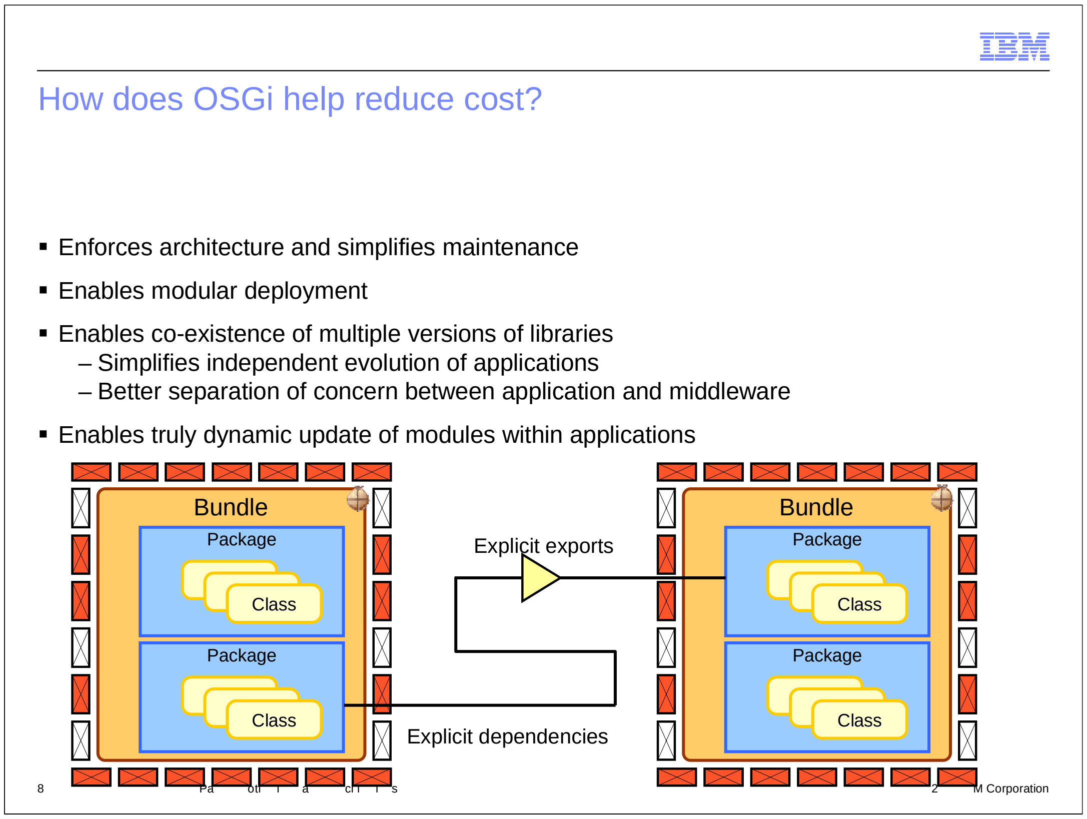
- "The dynamic module system for Java"
  - Mature 10-year old technology
  - Governed by OSGi Alliance: http://www.osgi.org
  - Used *inside* just about *all* Java-based middleware
    - IBM WebSphere, Oracle WebLogic, Red Hat JBoss, Sun GlassFish, Paremus Service Fabric, Eclipse Platform, Apache Geronimo,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

7 Part 1: Motivations and specifications © 2011 IBM Corporation

Solving the Java shortcomings is very straightforward. If you look inside Java client platforms like IBM Lotus® Expeditor or the Eclipse platform, or server-side middleware like the WebSphere platform, you will see that all these runtimes do not just run on a raw JRE, they all run inside an OSGi framework running on the Java Runtime Environment. OSGi is the dynamic module system for Java, governed by the OSGi Alliance, and has been in existence for over 10 years. It solves the Java problems by adding additional metadata to the JARs to first, explicitly declare what should be exposed from a JAR, and at what version, ensuring anything not explicitly mentioned is not visible outside the JAR, and second, explicitly declare what is required by the JAR in order for the JAR to run, and at what version.

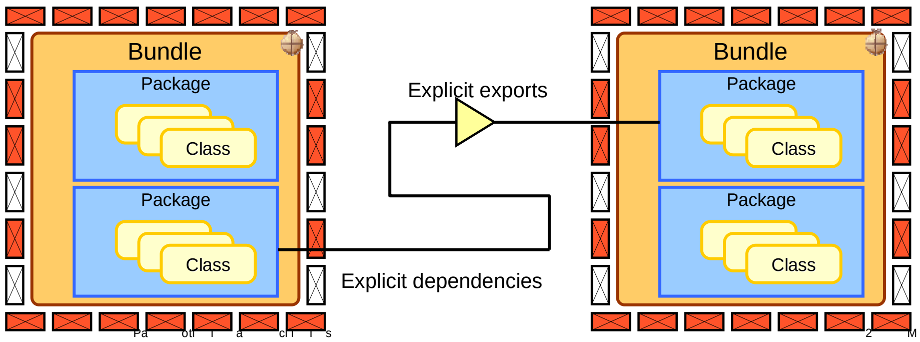
A JAR augmented by such metadata is still a valid JAR, but in an OSGi runtime the class loaders respect the metadata. Such augmented JARs are referred to as OSGi bundles and possess the modular qualities raw JARs lack. A bundle's internals cannot be seen by other bundles and it only starts if all its declared dependencies can be satisfied by other bundles.

A key aspect of modularity is the ability to replace one provider of an API (an Application Programming Interface) with another provider without the consumer needing to care. In practical terms, this means that bundles require a dynamic life cycle that is not tied to the life cycle of the Java Virtual Machine (or JVM) in which they run. OSGi provides explicit bundle life cycle management that paves the way for true in-JVM continuous availability.



How does OSGi help reduce cost?

- Enforces architecture and simplifies maintenance
- Enables modular deployment
- Enables co-existence of multiple versions of libraries
  - Simplifies independent evolution of applications
  - Better separation of concern between application and middleware
- Enables truly dynamic update of modules within applications



8 M Corporation

Because the dependencies are explicitly defined, the wiring between modules is all by design rather than by opportunity. It becomes easier to scrutinize any new dependencies because you need to explicitly update the module metadata.

The declaration of explicit dependencies can be used by a deployment system to enable more modular deployment, vastly simplifying the operational management of large suites of applications. Moreover, the explicit wiring of specific versions of dependencies simplifies both the testing of applications, which can be tested in isolation and still achieve the same results as in a production server environment; and the management of application suites, giving the operations team the freedom to update common dependencies at a pace they can afford to test at.

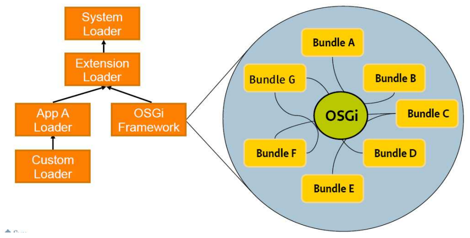
Finally, the dynamic life cycle of OSGi enables truly dynamic update of modules within an application.

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 class path.
  - Class sharing and visibility decided by declarative dependencies, not by class loader hierarchies.
  - OSGi framework works out the dependencies including versions



OSGi defines a dynamic module system for Java. It introduces some simple and yet powerful concepts to Java which eliminate each of the shortcomings just discussed.

The key notion introduced is the "bundle", as the modular unit. The OSGi platform architecture is based upon bundles as the unit of deployment.

A bundle is just a JAR archive with a JAR manifest but the manifest contains additional OSGi metadata that is processed by the OSGi module layer. This metadata describes all the modularity aspects of the bundle.

Some notable metadata in the manifest includes the bundle version header, import package header, and export package header.

The Bundle-Version header is used to qualify the version of the bundle and enables multiple versions of the bundle to be concurrently active.

The Import-Package header declares the external dependencies of the bundle. Specific versions or version ranges can be declared here. In the example, the imported package is required at version 1.1.2 or later.

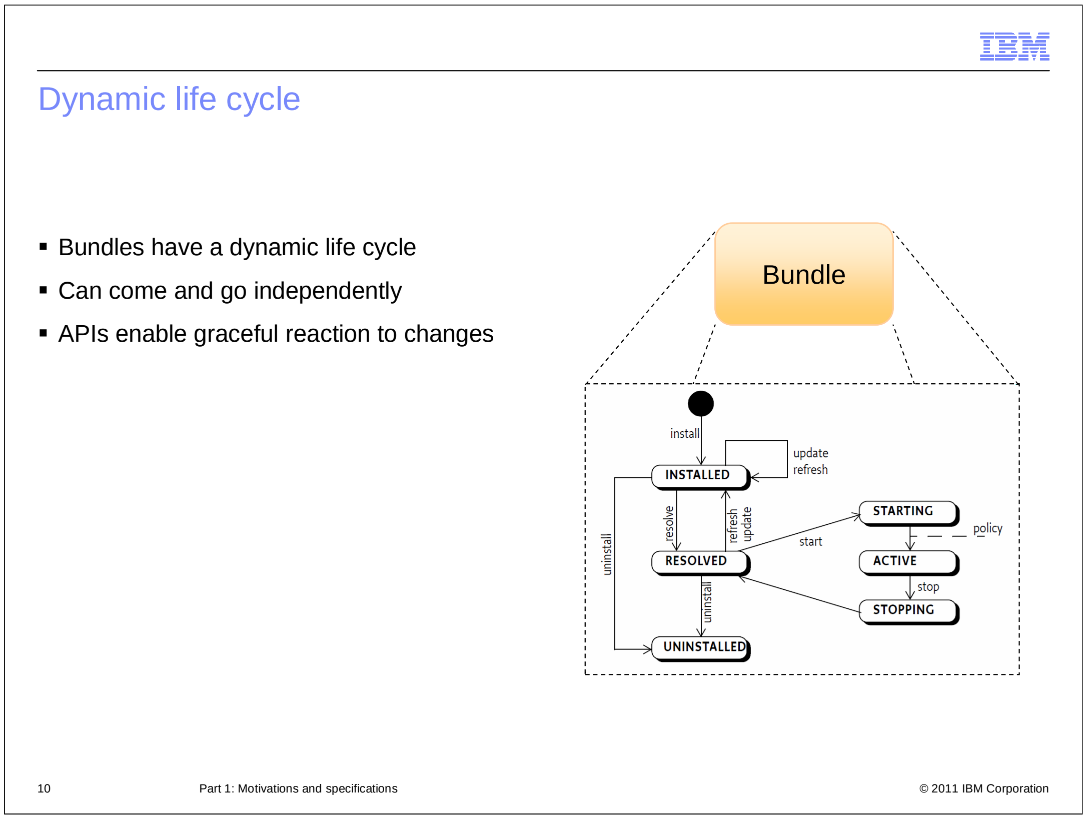
The Export-Package header declares the packages that are visible outside the bundle.

Any package not declared in the manifest has visibility only inside the bundle.

Eclipse and IBM Rational Application Developer tools provide a convenient editor for this manifest.

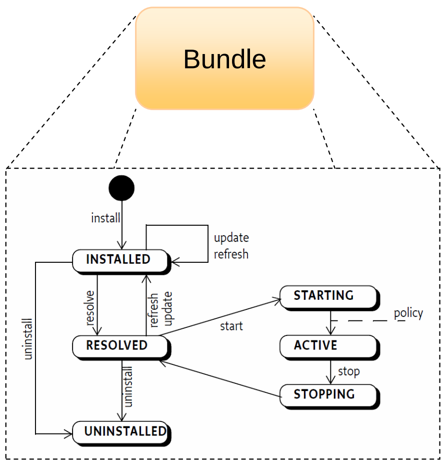
How is this metadata exploited? It is used by the OSGi class loader. There is no global class path in OSGi – the OSGi framework determines the dependencies and calculates the independent class path for each bundle.

Thereby, each of the shortcomings of plain Java class loading are eliminated. Only declared exports are visible outside the bundle, dependencies are resolved to specific versions and multiple versions of packages can be available concurrently for different client bundles, and dependencies are explicit so that bundles will not start if any dependencies cannot be fulfilled.



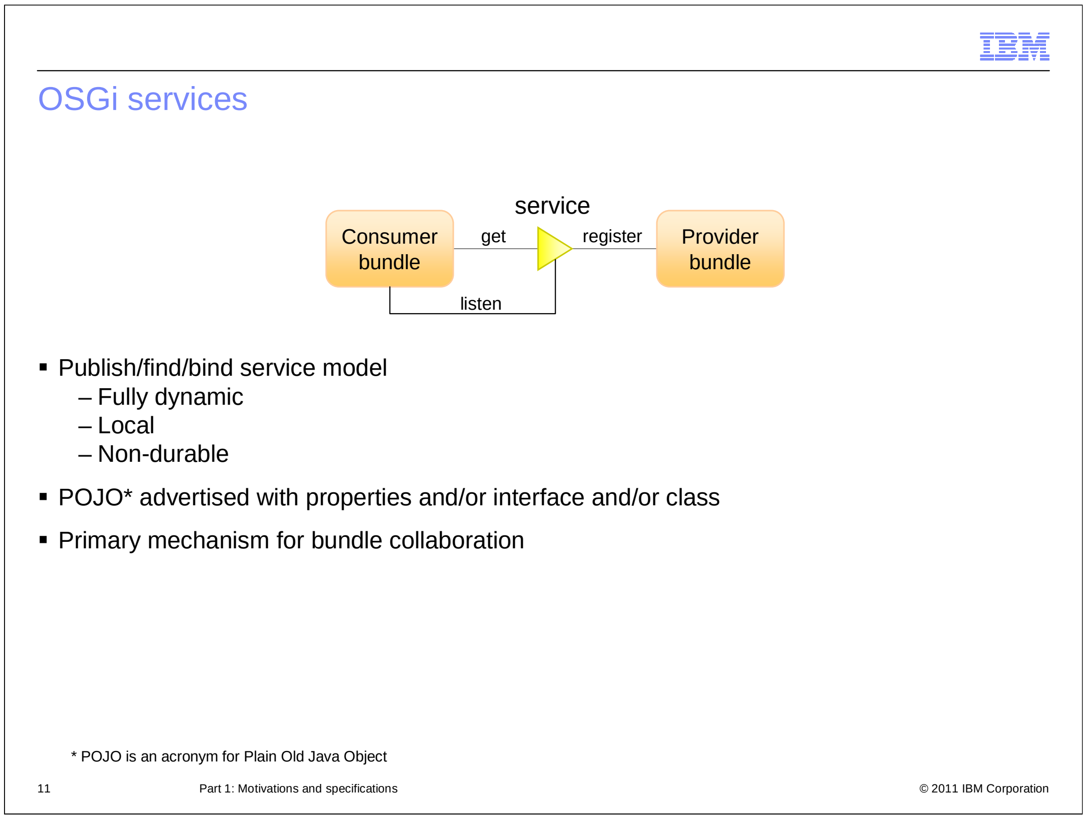
Dynamic life cycle

- Bundles have a dynamic life cycle
- Can come and go independently
- APIs enable graceful reaction to changes

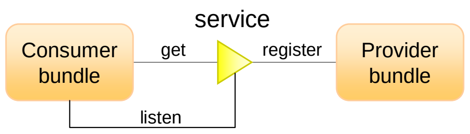


In the OSGi Service Platform, bundles have a dynamic life cycle. A bundle is first 'installed' into the OSGi framework. A process called 'resolution' determines whether all the bundles dependencies can be satisfied. If successful the bundle moves to the 'resolved' state and can be started. In this state classes and resources in the bundle can be used by other bundles. From this state the bundle can be started. It briefly goes through the 'starting' state where the bundle's startup class, called an Activator is executed, then the bundle transitions into the 'active' state.

Bundles can be installed, started, stopped, and uninstalled independently of each other.



OSGi services



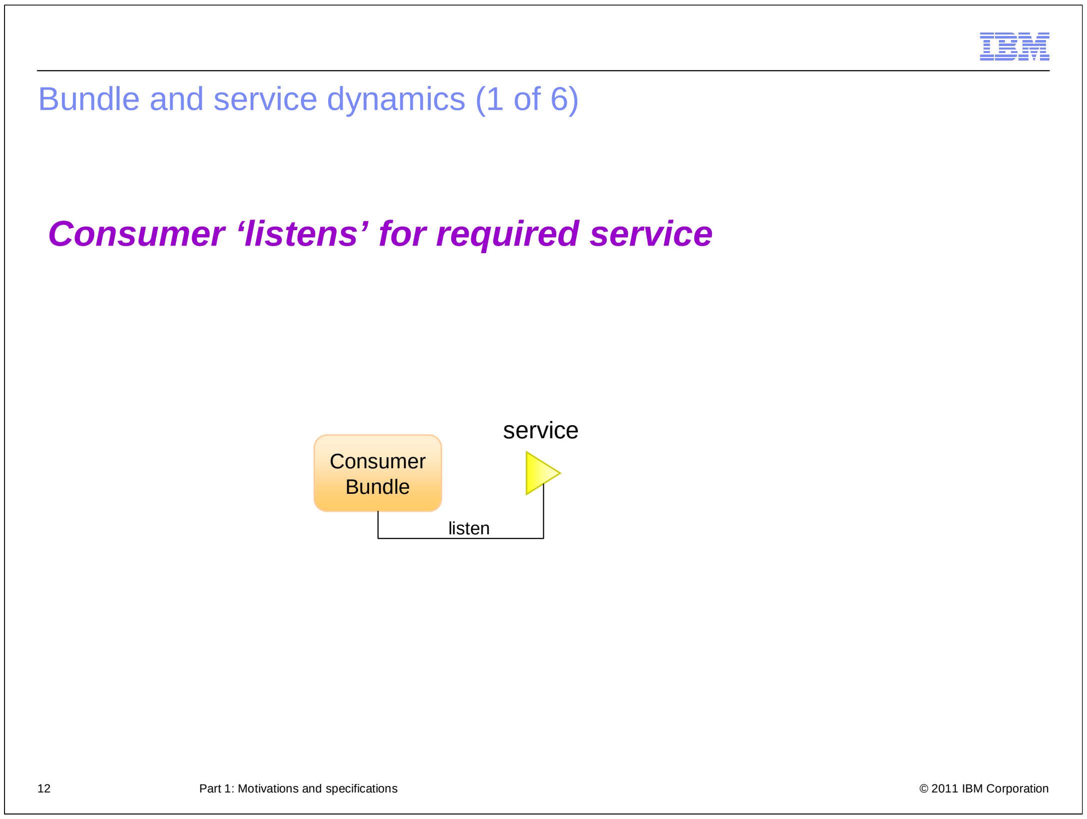
- Publish/find/bind service model
  - Fully dynamic
  - Local
  - Non-durable
- POJO* advertised with properties and/or interface and/or class
- Primary mechanism for bundle collaboration

* POJO is an acronym for Plain Old Java Object

11 Part 1: Motivations and specifications © 2011 IBM Corporation

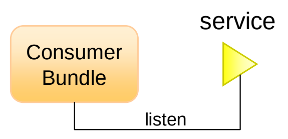
The OSGi Service Platform provides a fully dynamic publish, find, bind service model locally within the context of an OSGi framework running in a JVM. Provider bundles may be stopped and started causing POJO services to be registered and deregistered, independently of the life cycle of the consuming bundles.

The Service Registry is the primary mechanism within OSGi by which bundles collaborate.

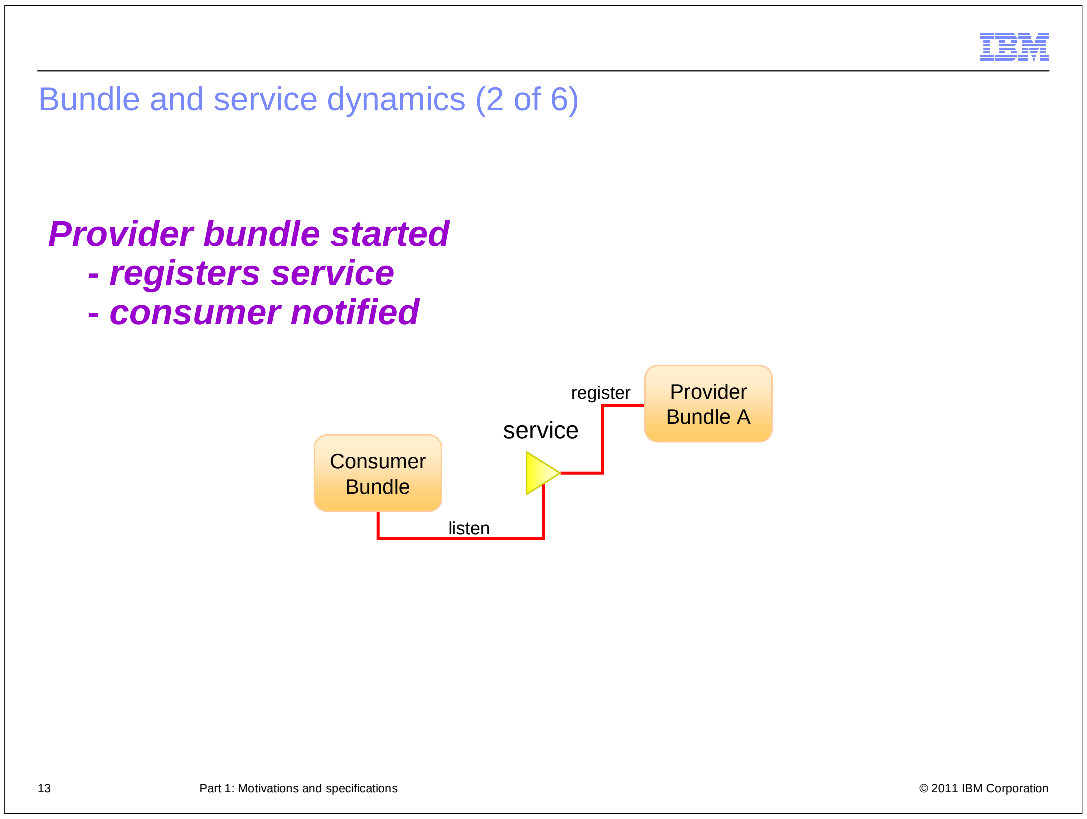


Bundle and service dynamics (1 of 6)

Consumer 'listens' for required service

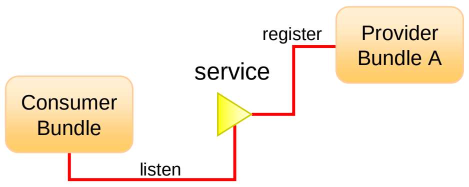


For example, a bundle that wants to consume a service listens for that service to become available in the Service Registry. Initially there are no services available.



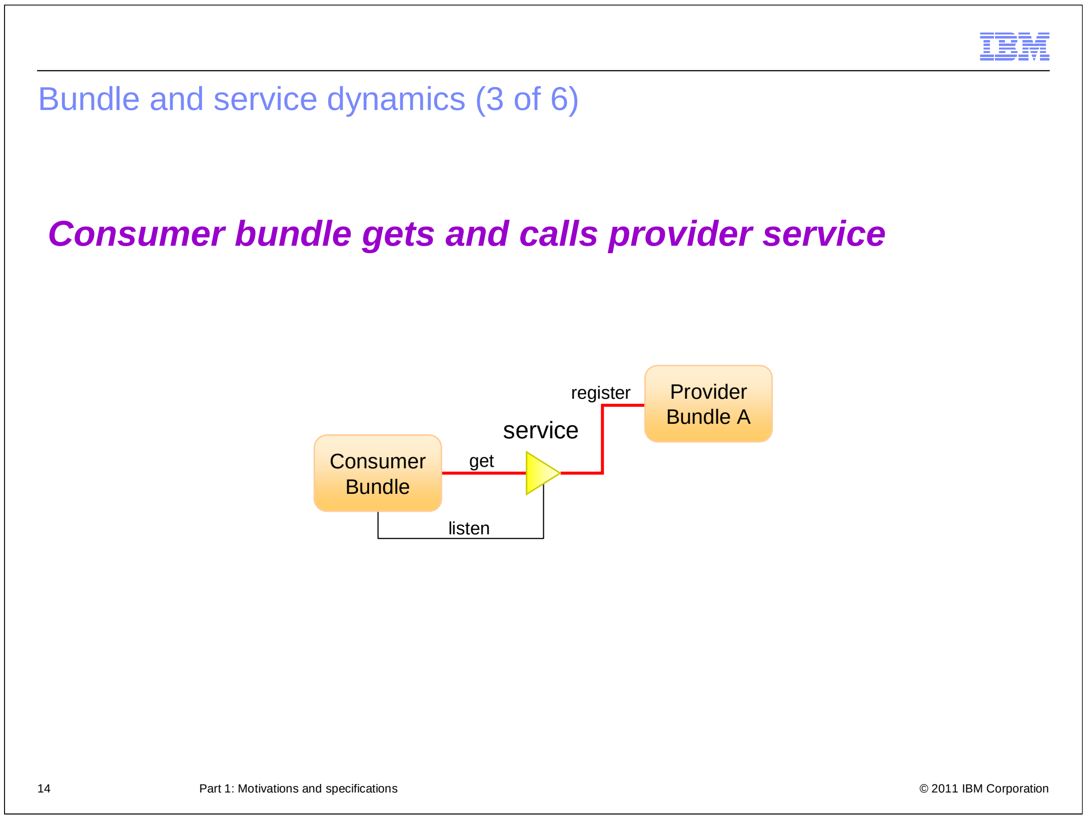
Bundle and service dynamics (2 of 6)

Provider bundle started
- registers service
- consumer notified



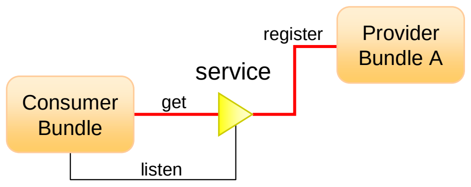
13 Part 1: Motivations and specifications © 2011 IBM Corporation

Then when a provider bundle is started and a service it provides is registered in the OSGi Service Registry, the consumer bundle is notified.

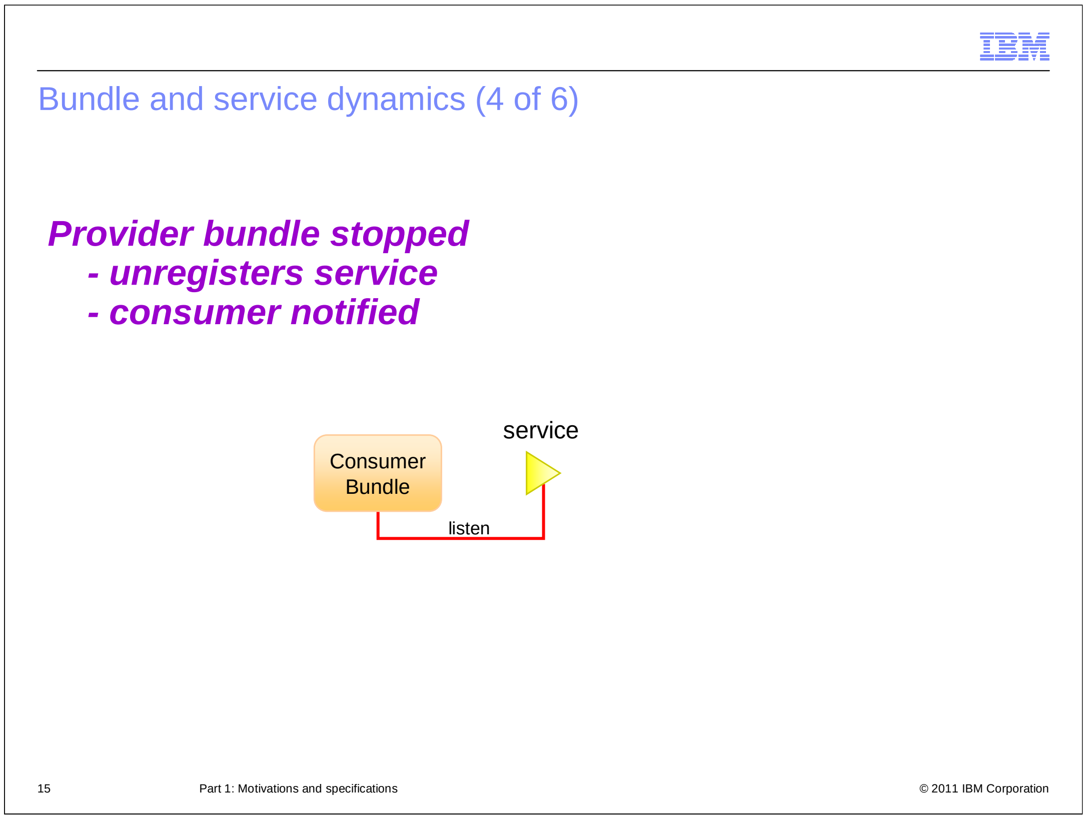


Bundle and service dynamics (3 of 6)

Consumer bundle gets and calls provider service

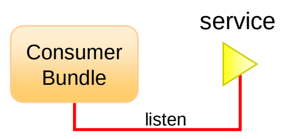


The consumer bundle gets an instance of the service and uses it.



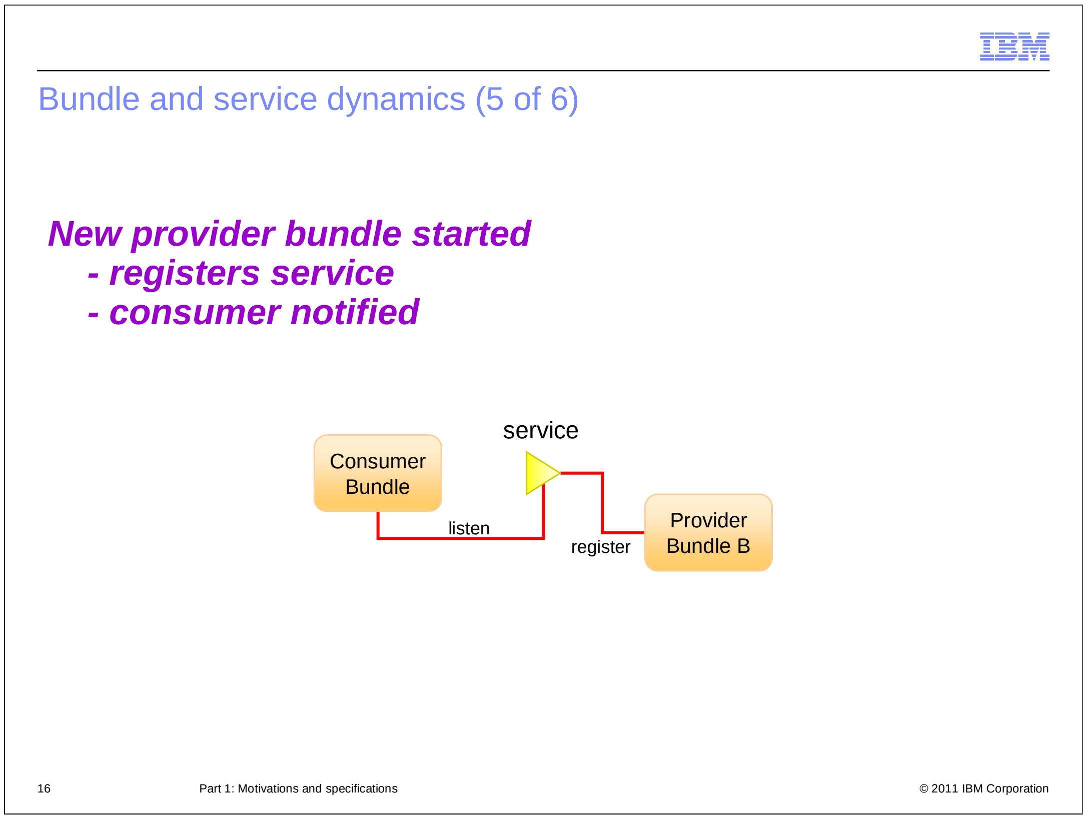
Bundle and service dynamics (4 of 6)

Provider bundle stopped
- unregisters service
- consumer notified



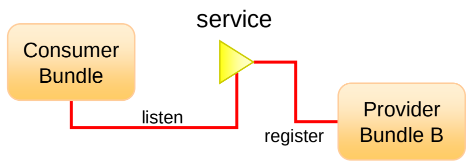
15 Part 1: Motivations and specifications © 2011 IBM Corporation

While the consumer is using the service, the bundle providing the service can be stopped, causing the service to be unregistered from the service registry. The consuming bundle is notified and from this point on, the consuming bundle can no longer use that service.



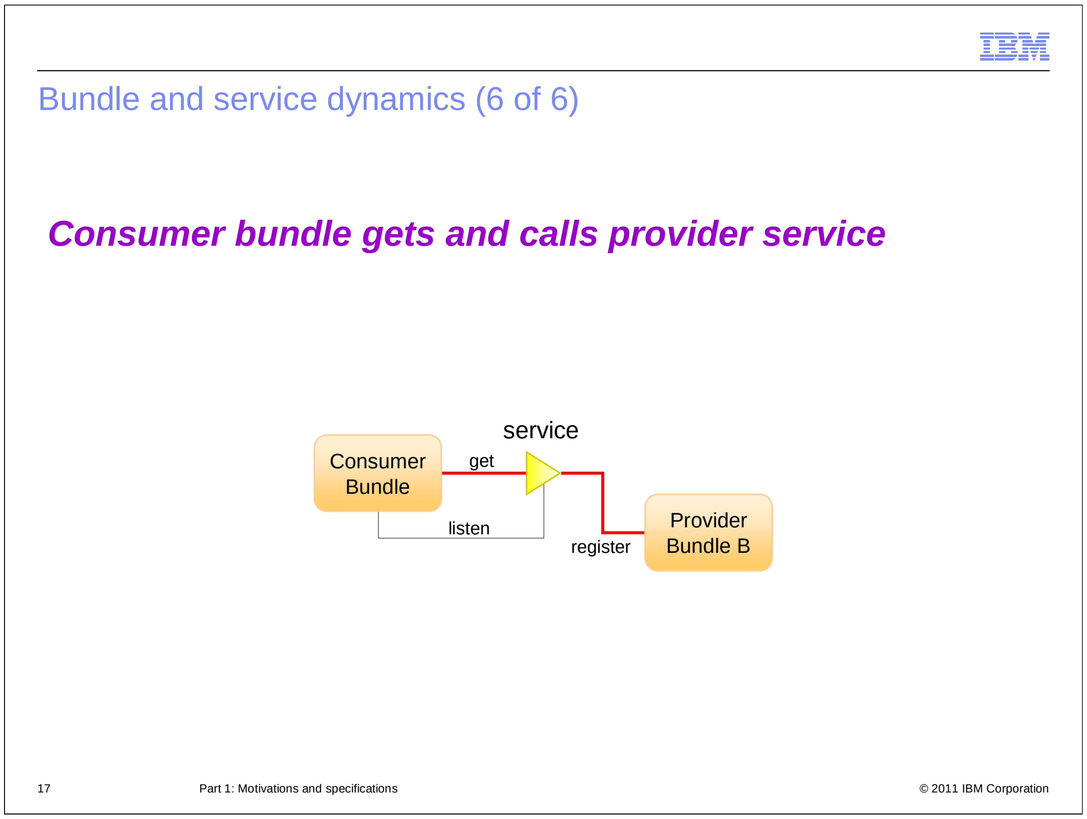
Bundle and service dynamics (5 of 6)

New provider bundle started
- ***registers service***
- ***consumer notified***



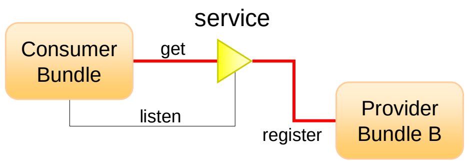
16 Part 1: Motivations and specifications © 2011 IBM Corporation

Subsequently a new bundle may be started and register an alternative implementation of the service interface being listened for by the consumer. The consumer bundle will be notified of this new registration.



Bundle and service dynamics (6 of 6)

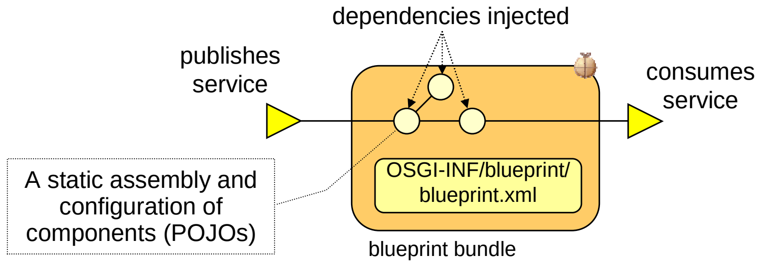
Consumer bundle gets and calls provider service



17 Part 1: Motivations and specifications © 2011 IBM Corporation

The consumer bundle may then get an instance of the service and one will be provided by the new provider.

Declarative OSGi services using blueprint


dependencies injected
publishes service
consumes service
OSGI-INF/blueprint/ blueprint.xml
A static assembly and configuration of components (POJOs)
blueprint bundle

- XML Blueprint definition describes component configuration and scope
  - Optionally publish and consume components to/from OSGi service registry.
  - Standardizes established Spring conventions
- Simplifies unit test outside either Java EE or OSGi r/t.
- **In WebSphere Application Server, the Blueprint DI container is a part of the server runtime (compared to the Spring container which is part of the application.)**

18 Part 1: Motivations and specifications © 2011 IBM Corporation

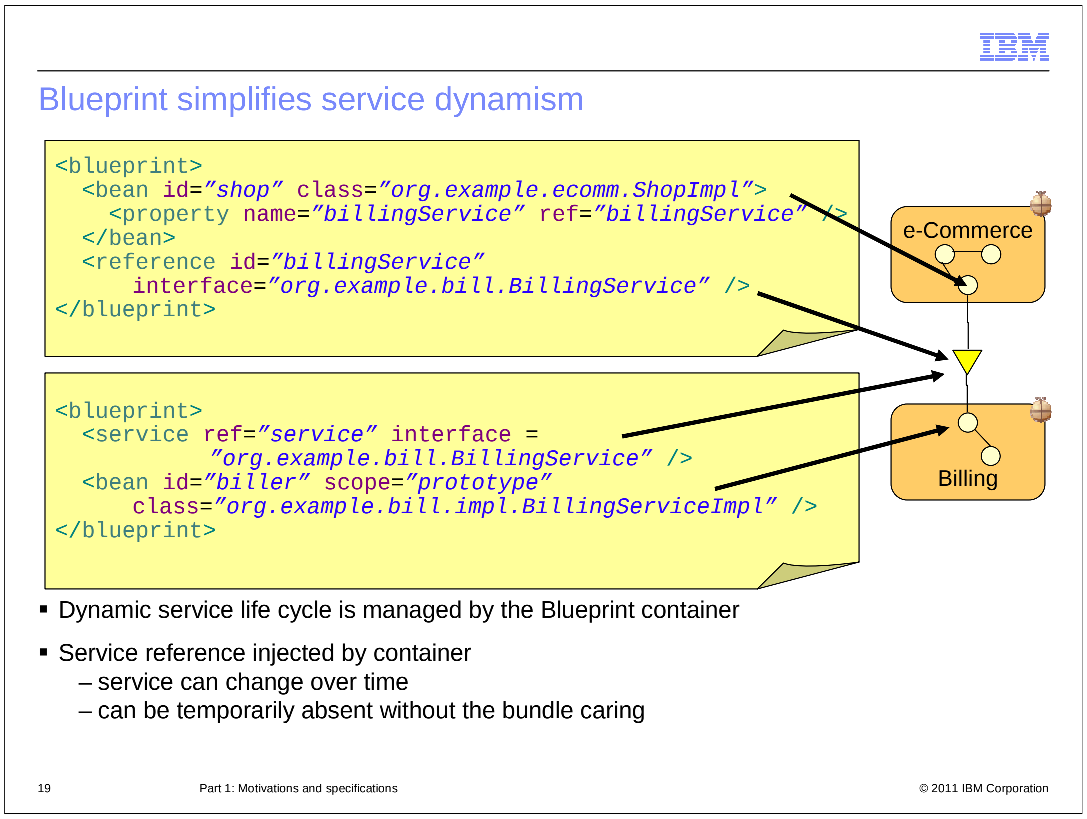
One of the significant features of the OSGi Release 4, Version 4.2 specification is its introduction of the Blueprint component model. This is the result of the standardization activity around the Spring Framework in the OSGi Alliance. A sizeable portion of web applications use the Spring framework today for its POJO component model and dependency injection container. Spring provides a convenient way for business logic to be encapsulated into POJO components, which have all their dependencies injected into them by the Spring container. Since the POJO components have no Java dependencies on the application server it is very simple to unit test the business logic in plain Java Standard Edition or Eclipse environment.

The Spring framework is a container that is packaged as a library with the application. In a Java EE environment the Spring container delegates to the underlying Application server for the management of resources such as database connections and for the application of qualities of service such as transactions and security. In a Java EE environment, Spring is essentially a proxy container to the native web container provided by the Application server and can add a significant amount of path-length.

By standardizing the Spring XML configuration format in the OSGi Alliance and delivering the container as an OSGi bundle, it has become possible to pull the dependency injection container out of the application and into the middleware. The standards-based evolution of the DI container is called the Blueprint container and WebSphere Application Server V8 integrates this container as part of the Application server, delivered and supported by IBM.

The Blueprint XML configuration file has the same structure as the Spring XML configuration file but in an OSGi namespace. The Blueprint XML is a bean definition file for all the beans provided by a single bundle. In addition to the bean definitions that will be familiar to Spring developers, the Blueprint model adds new ‘service’ and ‘reference’ elements as part of the integration with the OSGi environment. Service elements direct the Blueprint container to publish a bean as a service implementation through its service interface class into the OSGi Service Registry. A reference element directs the Blueprint container to locate a service that can be consumed from outside the bundle. The yellow arrows in the figure indicate OSGi services that are published to and discovered from the OSGi Service Registry by the blueprint container. Blueprint provides a declarative way of interacting with the Service Registry. In this manner, Some of the dynamic aspects of that interaction, for example listening to service life cycle events, can be devolved to the Blueprint container.

Ultimately, the Blueprint container manages the life cycle and dependencies of the POJO beans that contain the application logic and the services and references each bundle provides, and ensures references are wired to available services.



Blueprint simplifies service dynamism

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

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

e-Commerce
Billing

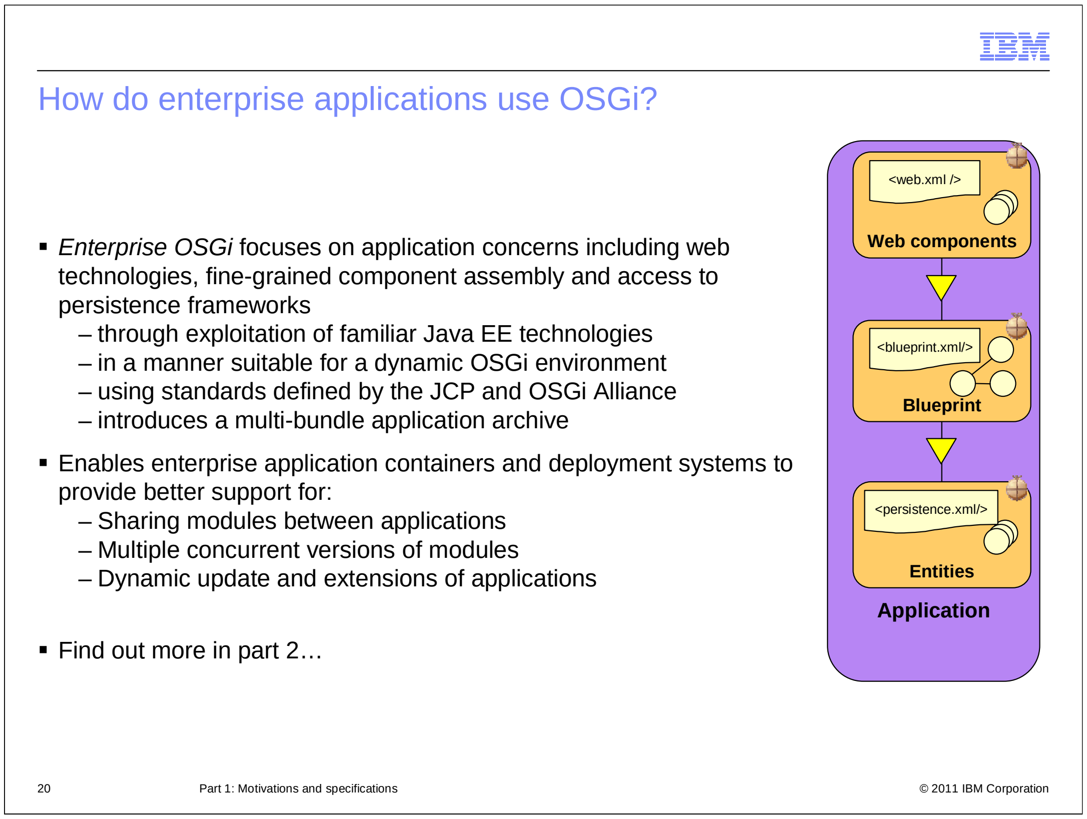
- Dynamic service life cycle is managed by the Blueprint container
- Service reference injected by container
  – service can change over time
  – can be temporarily absent without the bundle caring

Here is a simple example of an eCommerce bundle that contains a “shop” bean that uses a BillingService provided by another bundle. The Blueprint container locates the Billing Service provider and injects it into the shop bean at runtime.

OSGi services are dynamic, so if the service provider is changed then the blueprint container dynamically rewires the reference to a new service without impacting the shop bean instance.

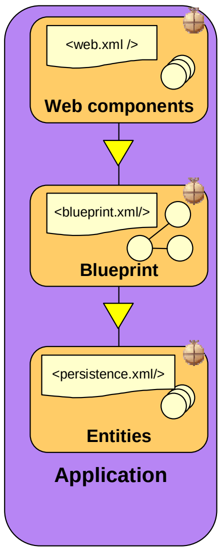
On the provider side, the BillingServiceImpl is another POJO implementing a Java interface for which a service is registered by the Blueprint container when the Billing bundle is started.

By default beans are created with “singleton” scope which means only a single instance is created by the container. If the bean maintains any state then it can be declared with “prototype” scope so that the container creates a new instance each time it needs to inject the dependency into a client.



How do enterprise applications use OSGi?

- *Enterprise OSGi* focuses on application concerns including web technologies, fine-grained component assembly and access to persistence frameworks
  – through exploitation of familiar Java EE technologies
  – in a manner suitable for a dynamic OSGi environment
  – using standards defined by the JCP and OSGi Alliance
  – introduces a multi-bundle application archive
- Enables enterprise application containers and deployment systems to provide better support for:
  – Sharing modules between applications
  – Multiple concurrent versions of modules
  – Dynamic update and extensions of applications
- Find out more in part 2…

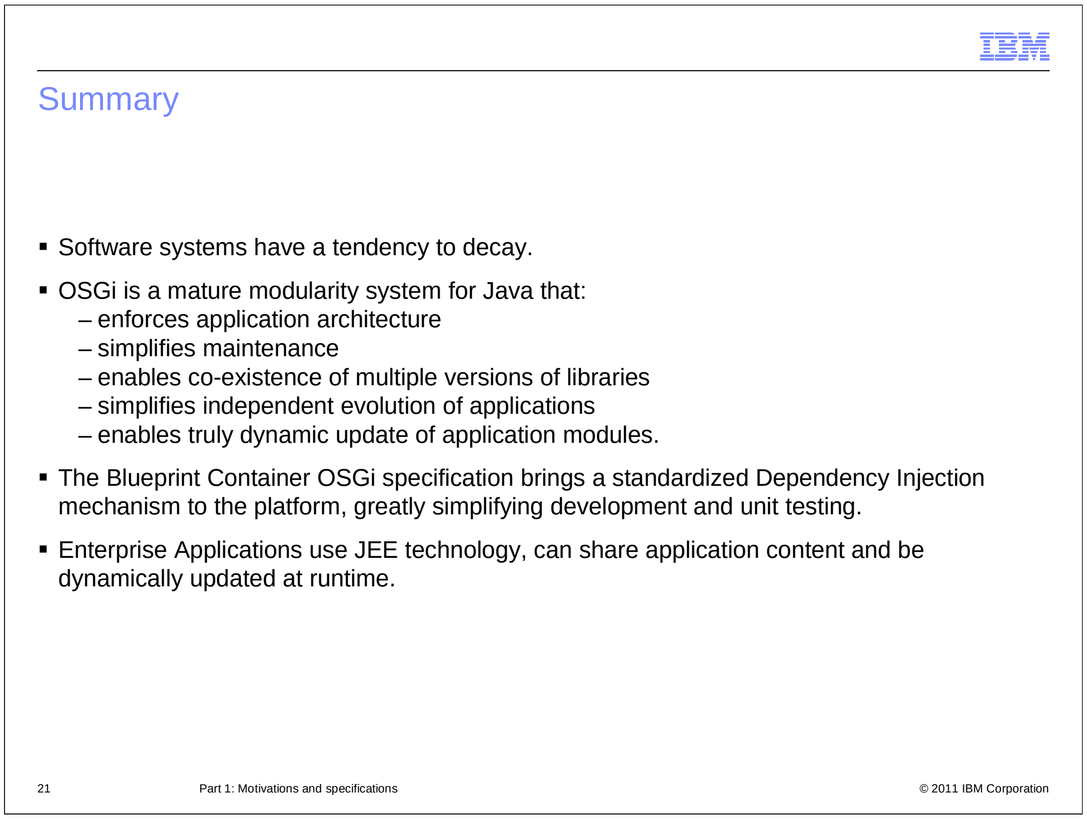


20 Part 1: Motivations and specifications © 2011 IBM Corporation

What does an application that uses OSGi look like?

Actually, it looks a lot like a standard web application using all the familiar technologies and standards that web apps are used to. The focus of the OSGi R4 V4.2 Enterprise specification, published in 2010, was how to augment existing Java SE and EE technologies to make them suitable for a dynamic OSGi environment. For the most part, this means making Java EE services available as OSGi services so their availability can be tracked dynamically. But at its simplest, an OSGi application can be no different from a Java EE application beyond having additional OSGi metadata describing bundles' internal and external dependencies.

A multi-bundle Enterprise application provides a way to group bundles together to form an application that can be deployed to a suitable runtime environment providing support for Java Enterprise Edition technology, such as IBM WebSphere Application Server V8. Modules can be shared between applications. Multiple versions of modules may be running concurrently. Applications can have their constituent bundles updated dynamically, and applications can be extended dynamically, and without restarting the application. Find out more in part 2.



Summary

- Software systems have a tendency to decay.
- OSGi is a mature modularity system for Java that:
  - enforces application architecture
  - simplifies maintenance
  - enables co-existence of multiple versions of libraries
  - simplifies independent evolution of applications
  - enables truly dynamic update of application modules.
- The Blueprint Container OSGi specification brings a standardized Dependency Injection mechanism to the platform, greatly simplifying development and unit testing.
- Enterprise Applications use JEE technology, can share application content and be dynamically updated at runtime.

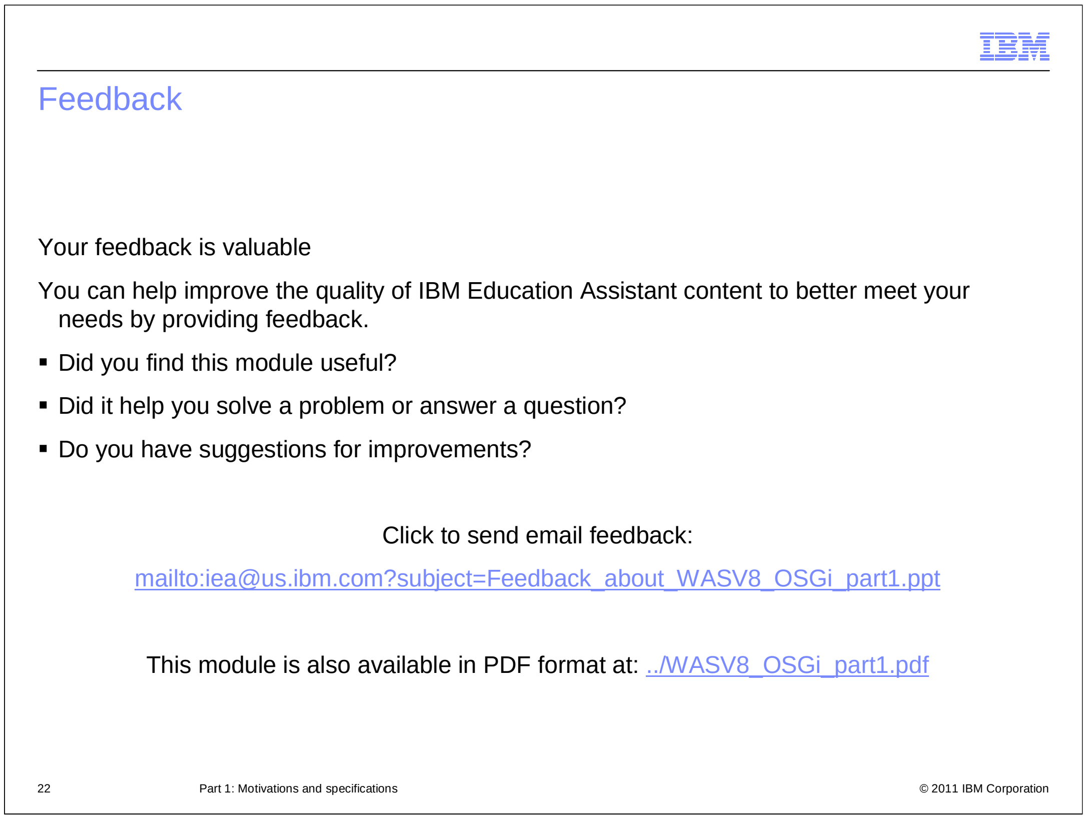
21 Part 1: Motivations and specifications © 2011 IBM Corporation

Software systems have a tendency to decay. Changes of a particular scale become increasingly expensive to perform. The primary cause of that decay is the increasing entanglement of the components that make up that system.

OSGi is a mature modularity system for Java that has been used inside tools and runtime infrastructure for many years
The OSGi platform dramatically improves on system entanglement. It enforces application architecture and simplifies maintenance; it enables modular deployment; it enables co-existence of multiple versions of libraries, simplifying independent evolution of applications and better separating application concerns and middleware concerns; and it enables truly dynamic update of modules within applications achieved through the dynamic bundle life cycle and the Service Registry.

The OSGi Blueprint Container model brings a standardized Dependency Injection mechanism to the platform greatly simplifying development and unit testing of applications.

Enterprise applications made up of bundles exploit existing JEE technology for persistence transactions processing, can share application content for improved memory footprint, and can be dynamically updated at runtime.



Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WASV8_OSGi_part1.ppt

This module is also available in PDF format at: ../WASV8_OSGi_part1.pdf

22 Part 1: Motivations and specifications © 2011 IBM Corporation

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information