RC23954 (W0605-059) May 10, 2006
Computer Science

IBM Research Report

A Java Framework for Building and Integrating Runtime Module Systems

Olivier Gruber
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Richard S. Hall
Laboratoire LSR

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

A Java Framework for Building and Integrating Runtime Module Systems

Olivier Gruber and Richard S. Hall
IBM Research ogruber@us.ibm.com
Laboratoire LSR richard.hall@imag.fr

28th of April, 2006

Abstract

We present the design of core mechanisms for building interoperable module systems from reusable components. Our approach promotes flexibility and modularity, separating the different fa-cets of a module system: an interface to load classes or resources for applications, a delegation model between modules, and class reification. Our design can model and integrate the module system of popular platforms such as GBeans, NetBeans, or the OSGi Service Platform.

1 Introduction

In recent years, there has been growing interest in runtime module systems within the Java community [6, 7]. Java has the class loader concept but it is a very simplistic module system. Unfortunately, class loaders mix three facets: a way to load classes or resources for applications, a delegation model between class loaders, and the actual class reification (i.e., reifying class-file bytes into class objects). To support other runtime components and services, different Java communities have proposed, above Java class loaders, different runtime module systems such as GBeans [3] or OSGi technology [1].

There seems to be overall agreement that a runtime module is a namespace for Java types and resources. Furthermore, there is usually a delegation model where modules may delegate to other modules for loading types or resources. However, there is almost no agreement on the details of runtime modules, such as the granularity of delegation (e.g., Java types, packages, or entire modules) or encapsulation capabilities. We also find very different module packaging, from simple to extended Java archive (JAR) files as well as other formats such as JXE files for the IBM J9 virtual machine [5].

This paper introduces our work on a comprehensive Java framework for building and integrating runtime module systems, which work continues previous work in this area [4]. It provides core reusable and extensible mechanisms that enables the building of specific module semantics through plug-in policies. The benefits are that module systems could be more easily built and understood. First, module systems are designed in terms of the same core concepts, thereby promoting reuse. Second, common reusable policies could be created so that specific module systems could be developed by simply reusing or customizing existing policies. Third, our approach enables interoperability between different module systems when installed within a single Java Runtime Environment (JRE).

This framework is the result of having to deal with diversity of module semantics through the releases of Felix, the Apache incubator implementation of the OSGi framework. We designed our mechanisms

to support implementations of the OSGi framework through its Release 3 and 4 specifications. We realized that in fact these mechanisms could be used to implement a large number of existing module semantics. We further realized that with minor modifications to the current Felix design we could enable interoperability between heterogeneous module systems.

This position paper is structured as follows. In Section 2, we introduce the overall architecture of our approach for runtime modules. In Section 3, we discuss our framework from the perspective of integrating different module systems in a single Java Runtime Environment. In Section 4, we focus in contrast on the building of a module system using reusable mechanisms. We will also discuss concrete examples of known module systems. In Section 5, we conclude.

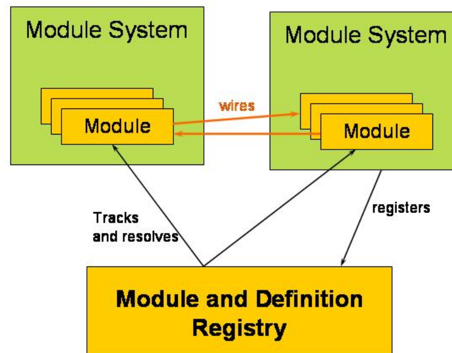

Figure 1: Integrated Module Systems

2 Overall Architecture

Our focus is on the design of the low-level framework for building and integrating various module systems, as depicted in Figure 1. Different module systems—such as the module layers of the NetBeans, GBeans, or OSGi platforms—could be developed using our core mechanisms, which would allow them to easily integrate within a single Java Runtime Environment.

The module registry (*IModuleRegistry*) is at the center of our approach. It is a shared registry, one per Java Runtime Environment (JRE). The registry tracks module systems, module instances, and module definitions in that JRE.

Each installed module system (*IModuleSystem*) registers itself with the module registry, using a unique key identifying that module system. Each module system provides a certain semantics for runtime modules. Runtime modules (*IModule*) are namespaces for accessing classes and resources. Each module system reifies module definitions into actual runtime modules. A module definition (*IModuleDefinition*) is an abstract description of a version of a module, identified by a module name and version.

The registry provides a shared place for different module systems to find module definitions they wish to reify into runtime module. We do not impose any restriction on this reification process. In most environments, a module definition will be instantiated only once, bringing in memory only one version of a module. Other environments will need to be able to load different versions of certain modules. In rare situations, one may want to instantiate one definition multiple times. Our framework allows it, but we discourage it.

Although module instantiation is specific to each module system, our framework provides core reusable and extensible concepts for structuring the internals of modules, as depicted in Figure 2. We do not impose these internal concepts, any module system is free to implement its runtime modules as it chooses. However, our concepts are meant to ease the development and understanding of module systems through reuse of well-defined concepts. Furthermore, the adoption of these concepts will allow finer-grain interoperability among module systems.

Any created module is tracked by the module registry, which supports the integration of module systems. First, the registry gives a place to find existing runtime modules, which promotes sharing these modules, even among module systems. Second, the registry is the guardian of the overall lifecycle of modules. A runtime module goes through two states: unresolved and resolved. A module is resolved when

it has all its dependencies satisfied. Dependencies among modules are carried by wires (*IWire*) and materializes the delegation among modules.

This resolution process of runtime modules is two-level recursive process. One level is local to each module system and it involves a specific resolving algorithm in an homogeneous world, among modules instantiated by this module system. Each module system will have its own approach for this first level. The second level however is the coordination among module systems and it is the responsibility of the module registry.

3 Integrating Module Systems

This section focuses on the integration of module systems through our module registry. We discuss the details of module definitions, modules and wires. We also discuss the details of the module resolution process. The central actor is the module registry, whose details follow:

```
public interface IModuleRegistry {

  boolean addModule(IModuleDefinition md,IModuleSystem ms);
  boolean removeModule(IModuleDefinition md);
  IModuleSystem getOwnerModuleSystem(IModuleDefinition md);

  IModuleDefinition[] find(String name);
  IModuleDefinition[] find(String name, String version);

  IModule makeInstance(IModuleDefinition md,
                       boolean isResolving);
  void destroyInstance(IModule module);

  void resolve(IModule module) throws ResolveException;
  boolean isResolving(IModule module);

  IModule[] getInstances();
  IModule[] getInstances(String name);
  IModule[] getInstances(String name, Version version);
  IModule getInstance(String instanceId);

  IModuleSystem getModuleSystem(String key);
  void addModuleSystem(String key, IModuleSystem framework);
  void removeModuleSystem(String key);
}
```

The module registry is a mechanism for integration that we can characterize in different dimensions. The first dimension is a registry of module systems, where installed module systems are registered with a unique key. It is important for the registry to track installed module systems to coordinate them during the resolution process for created runtime modules. The second dimension is a registry of known module definitions, registered by module systems. Notice that when a module definition is added to the registry, it is associated with the module system registering it, called its owner module system. For each module definition, the registry will use its owner module system as a factory to create runtime modules from that module definition. The third dimension is the coordinator of the overall resolution process across module systems.

A module definition (*IModuleDefinition* describes one version of a given module; it is identified by the name and the version of that module.

```
public interface IModuleDefinition {
  String getName();
  Version getVersion();
  Object[] getImports();
  Object[] getExports();
  Map getAttributes();
  IModuleDefinition getSubmodule(String name);
}
```

The concept of a module definition is abstract and minimal; it represents the lowest common denominator among the different views of what a module is across different module systems. Fundamentally, it describes a module version, with its name, its version and both its imports and exports. The module metadata embodied in module definitions allows different module systems to locate specific module definitions they are looking for. This process is most often based on the module name and version.

It may, however, also use the imports and exports. This requires some understanding of these imports or exports, which requires knowing something about the kind of module definitions. This can be obtained from the owner module system, itself obtained from *IModuleRegistry.getOwnerModuleSystem()*. Knowing the owner usually enables knowing the actual format of imports and exports. It may also enable downcasting *IModuleDefinition* into a more concrete interface with more information.

For example, one may look for a particular export, like a Java package, rather than a particular module. One would therefore need to look at the exports of all module definitions it can understand. Looking at imports is also useful in certain cases as it might

help selecting among several acceptable module definitions. For instance, one would prefer a module definition with resolvable imports over one that is not resolvable (satisfied imports).

3.1 Modules and Wires

A module instance (*IModule*) is an actual runtime concept, providing a namespace for classes and resources. It replaces the traditional Java class loader concept[1] as a way for Java code to find classes or access resources (see methods *getClass* and *getResource*). The details of the runtime module are below:

```
public interface IModule {
  String getInstanceId();
  IModuleRegistry getInstanceRegistry();
  IModuleDefinition getDefinition();
  IContent getContent();
  IWire[] getUsedBy();
  IWire[] getUses();
  IWire createWire(IModule importer);
  IWire createWire(IModule importer, Object export);
  void revokeWire(IWire wire);
  boolean isResolved();
  Class getClass(String name)
    throws ClassNotFoundException;
  URL getResource(String name)
    throws ResourceNotFoundException;
}
```

A runtime module may load classes and resources solely from a local content. The local content (see below) is an abstraction over the physical content of the module. It provides minimal read-only support for listing and reading the contents of the module. This provides access to the actual bytes that runtime modules use to reification of classes as well as accessing resources. Furthermore, this is necessary for certain module platforms that actually allow access to a module's content as byte-oriented entries.

```
public interface IContent {
  void open();
  void close();
  boolean hasEntry(String name);
  byte[] getEntry(String name);
  InputStream
    getEntryAsStream(String name)
    throws IOException;
  Enumeration
    getEntryPaths(String path);
}
```

[1]This would typically suggest that a Class refers to its module instance as opposed to its class loader.

We do promote however more complex modules than modules providing only a local access to their content. The rationale is to allow modules to depend on other modules and therefore be able to delegate class or resource loading to these modules they depend on. Delegation is based on the concept of wires (*IWire*) that provide access to classes and resources:

```
public interface IWire {
  IModule getImporter();
  IModule getExporter();
  Object getExport();
  Class getClass(String name)
    throws ClassNotFoundException;
  URL getResource(String name)
    throws ResourceNotFoundException;
}
```

We support delegation through wires at different granularities. At the very least, we promote module-level delegation where modules can create wires for exporting themselves. Such a wire is created through the *createWire* method, passing only an importer module. For finer-grain wires, one would use *createWire* method, passing both an importer module and an export object. The result is a wire for that particular finer-grain export.

Note that the wire is created by the exporting module and used by the importing module, not the other way around. The rationale is that an export has its semantics defined by the exporting module, not the importing module. In simple terms, the importer needs what the exporter offers, the exporter decides what that is.

Further note that finer-grain wires do require that importers have some deeper understanding of the exporting module, beyond the module name and version. Indeed, an importer needs to be able to choose the correct export object, even though exports are untyped, see the method *getExports* on *IModule*.

For example, assume a module system wants to integrate with an OSGi module system that provides both module-level and package-level exports. Each OSGi implementation would define its own actual format for its exports. One generic approach would be to have exports as strings following the syntax of the OSGi specification. Any module system understanding that specified syntax could look at the exports and decide which one is necessary to satisfy a given

import, asking it to create a wire to that particular export.

This OSGi example also illustrate why the wires are created by exporters and not importers. In OSGi, an exported Java package may be associated with a class-level filter, allowing it to control which classes are exported and which are not. This filtering role is achieved the wire. Note that this does not apply solely for finer-grain exports, but could also be used to control in a similar spirit what is exported at a module level. Indeed, a module may want to filter what packages or classes it exports as a whole.

3.2 Resolving Modules

The resolution of modules is the process of finding matching exports for imports. This resolution process is a shared responsibility between the module registry and the installed module systems. For a module to be resolved, it needs all its imports wired to exports of resolved modules. This is therefore a recursive process since module definitions, along with their imports and exports, describe a directed graph of dependencies, with potential cycles.

The resolve process starts with attempting to resolve one module, calling the *resolve* method on the module registry for that given module instance. This starts the resolve process as a two-phase protocol between the registry and installed module systems:

```
public interface IModuleSystem {
  IModule
    makeInstance(IModuleDefinition md);
  boolean
    prepareResolve(IModuleRegistry ir, IModule module);
  boolean
    completeResolve(IModuleRegistry ir, boolean result);
}
```

The first phase of the resolution process is called the *prepare phase*. When the registry is asked to resolve a module, it forwards the request to the module system that created that module through calling *IModuleSystem.prepareResolve()*. Resolving that module usually depends on other modules being resolved as well, because of imports.

The current module system needs to attempt to match these imports to acceptable exports. A given import may match many exports from different modules, potentially from different module systems. Each module system will implement its own policy for matching imports to available exports. This matching process may yield several candidate exports per import; one candidate must be selected. Such selected exports must have their module resolved so that a wire can be created. If any such modules are not resolved, the process recurses through calling *IModuleRegistry.resolve()*.

Note that we do not impose any selection policy, each module system is entirely free to adopt its own approach. It can use a constraint solver, often called a resolver. It can also use some wiring logic such as scripts. Scripts can be either man-made or generated from some metadata or some analysis of available imports and exports from known module definitions. Usually, policies will favor selecting exports from already resolved modules. If no resolved module is available, policies will often favor trying to resolve existing modules rather than creating new instances from module definitions. But again, those are only best practices and module systems are free to do as they see fit.

During this prepare phase of the resolution process, we potentially face cycles among modules. Cycles are detected through keeping track whether module instances are *resolving* or not. Whenever the registry is asked to resolve a module instance, it sets its state to resolving. The state can be queried through *IModuleRegistry.isResolving(IModule)*. To break cycles, the registry optimistically answers that an module instance is resolved if it is already resolving. At some later point, the cycle unwinds and a real decision is made.

Once the recursive prepare phase ends, a global decision is made for the second phase, called the *completion phase*. The completion phase is linear; the registry sequentially invokes all module systems that were involved in the prepare phase through *IModuleSystem.completeResolve()*. This call carries the global decision: complete or abort. The global decision is an *abort* if any module needing to be resolved could not be resolved. The final decision is a *commit* if all modules needing to be resolved could be resolved. This means that all module instances

needing to resolve must be set to a state of resolved.

In other words, the entire resolution process is atomic, it entirely fails or entirely succeeds. Moreover, it is isolated; the registry ensures that there is only one such process happening at any given time. When the process commits, this also means that wires may have to be created for the newly resolved module instances, materializing the delegation on selected exports during the prepare phase. If wires were eagerly created during that first phase, they need to be dropped if the global decision is an abort.

4 Building Module Systems

After focusing on the integration of different module systems, we will look at the internals of module systems. There seems to be overall agreement that a runtime module is a namespace for Java types and resources. Furthermore, there is usually a delegation model where modules delegate to other modules for loading types or resources. However, there is almost no agreement on the details of runtime modules, such as the granularity of delegation (e.g., Java types, packages, or entire modules) or encapsulation capabilities. We also find very different module physical packaging (such JAR files or others). In this section, we present the concepts of our framework related to building such diverse module systems. At the end, we will briefly sketch examples of how to build existing known module systems using these concepts.

4.1 Concepts

Our concepts separate the different facets of a module system: actual class loading from a physical container (*IContentLoader*), delegation model (*ISearchPolicy*), and the interface for Java developers to load classes and resources (*IModule*). The corresponding architecture is depicted in Figure 2.

The central concept is the module, the actual runtime object from which Java code gains access to classes and resources through *IModule.getClass() and IModule.getResource()*. As previously introduced, a module is an instance created from a module definition (*IModuleDefinition*), but we can see now that it

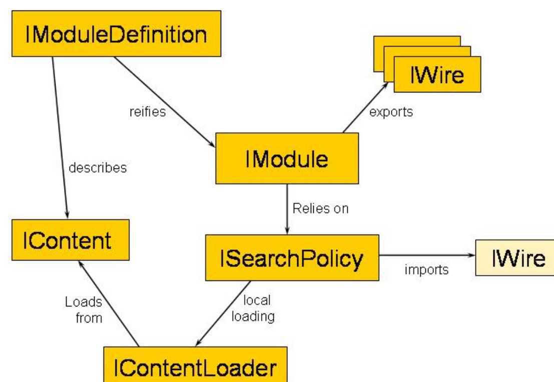

Figure 2: Integrated Module Systems

also owns a content.

Each module system manages contents in its own way. This is one of the area of wide diversity among approaches; almost no two module platforms manage module content the same way. Some download JAR files from the network but keep them as JAR files in the file system. Some download JAR files but explode them into directory structures. Some JAR files are complex, with embedded JAR files, introducing further layout challenges on the local file system. Some platforms support or promote other more specific formats such the JXE files for the IBM J9 virtual machine [5].

Beyond the storage layout of module content, the management of content often has complex rules of visibility or relevance. Some approaches are using chained repositories with the Unix-classical separations among user contents, system-level contents, and core contents (like the bootstrap module for the Java Virtual Machine). Some other approaches further promote the concept of different products installed, potentially sharing some modules but also having private ones. This usually happens through a complex management of locations in the file system.

Facing this diversity, we did not want to impose any given approach. However, we still needed the concept of abstract content for a module. First, the

definition of the *IContent* interface allows each module system to use whatever storage it wants as long as it can be mapped to a set of path-based entries, where entries can be access as byte arrays or input streams. This covers a wide range of storage systems. Second, the *IContent* interface provides an added interoperability level among module systems. Indeed, for some module systems, not all entries are accessible through the *IModule* interface if the module has a concept of a classpath. In such cases, the *IContent* interface provides full access.

Most importantly, the *IContent* interface abstracts the details of the physical storage so that reusable content loaders can be developed, the *IContentLoader* concept details are the following:

```
public interface IContentLoader {
  IContent getContent();
  void setSearchPolicy(ISearchPolicy searchPolicy);
  ISearchPolicy getSearchPolicy();
  void setURLPolicy(IURLPolicy urlPolicy);
  IURLPolicy getURLPolicy();
  Class getClass(String name);
  URL getResource(String name);
}
```

The content loader is the entity that knows how to reify Java types and resources from the actual module content (IContent). On current Java Virtual Machines (JVM), a content loader leverages a Java class loader for this as class loaders are the only way to reify classes in current JVMs. If a JVM would natively support a module system, it would provide an independent way to reify classes, passing the module as a context. This would suggest the eventual deprecation of the *ClassLoader* class as well as the *Class.getClassLoader()* method, which would be replaced by a *Class.getModule()* method.

Even using an internal class loader, the content loader has various policy and layout decisions to address. For instance, class loaders are not able to load classes from nested JARs. Each content loader may therefore provide a different caching policy. For instance, one content loader may be extracting the embedded JARs into a local cache. Another one may be extracting single classes at a time. Native libraries may also require special treatment since most operating systems cannot load libraries from JAR files.

Some JVMs provide specific class loaders for specific content such as the JXE for the IBM J9 virtual machine. The JXE format is the result of ahead-of-time loading and compilation of class files into a memory segment that can be mapped by the JVM in one operation. In this case, one would have its own content loader using internally this specific class loader. This separation of concerns is powerful. It is now easy for a module system to provide this capability on J9 virtual machines, it just needs a different content loader, but the rest of the module system remains unaffected.

Pushing further the separation of concerns, we wanted to separate out delegation among modules. The rationale is that delegation is one of the dimension where much diversity occurs among module systems. It is therefore interesting to have delegation isolated into its own pluggeable object. Furthermore, we hope reusable search policies moving toward further integration of existing delegation models.

In our framework, the concept of a search policy captures the functional role of delegation among modules along wires. We discussed earlier how wires materialize dependencies among resolved bundles. We also discussed how wires provide a core building block to delegation: access a class or a resource from another module. The search policy (*ISearchPolicy*) captures how a module will actually decide which wires to use when a specific class or resource is needed.

```
public interface ISearchPolicy {
  IModule getModule();
  Class findClass(String name)
    throws ClassNotFoundException;
  URL findResource(String name)
    throws ResourceNotFoundException;
  String findLibrary(String name);
}
```

We do not impose a dependency model or search policy. For instance, we have implemented different dependency models, such as OSGi R3 (package-level dependencies) and R4 (more expressive dependencies and module-level dependencies). It is also important to point out that we do not fix any metadata format used to express dependencies; module systems may choose any format they feel suits their needs best.

To understand how the search policy works, we need to look at the two paths leading to the loading of a class in Java. One path is a direct call to the method *IModule.getClass()*. This is typically the case when

some Java code wants to load directly a class. The module directly forwards the request to the search policy through *ISearchPolicy.findClass()*. The search policy will determine where the class should come from. If the class should come from another module, the request is forwarded through a wire to that module. It may be the case that the request may need to be forwarded to different modules through different wires. If the class should come from the local content, the request is passed to the content loader (*IContentLoader.getClass()*).

The other path is when the Java Virtual Machine needs a class. We are assuming standard Java and the internal use of class loaders for implementing content loaders. So each content loader has its own class loader, recipients of the JVM upcalls for needed classes (*ClassLoader.loadClass()*). For each upcall, the content loader checks if it has already loaded the requested class. If it has, it returns it; if not, it delegates to the search policy of its module through *ISearchPolicy.findClass()*. We are back to the case discussed above.

Before we move onto looking at examples of how to build some existing module systems with these concepts, we need to discuss the URL policy (*IURLPolicy*), provided to a content loader and detailed below.

```
public interface IURLPolicy {
  URL createURL(String path);
}
```

The issue comes down to the creation of the URL objects for resources in a module content. Neither the content nor the content loader have enough semantics or knowledge for creating such URL objects. Indeed, it is often the case that the syntax of these URLs are specific to module systems, such as using a specific protocol. Furthermore, they may include module-level information such as the module name or version.

The URL policy is usually implemented in the module system and most often as part of the module instance (*IModule* concrete implementation). The content loader uses the URL policy to upcall with the path of the resource in the local content. From that path, the module instance may create a meaningful URL.

4.2 Examples

This section discusses several different existing model systems. Taken together, they cover a wide range of approaches, which illustrates well the flexibility of our approach as core mechanisms for building module systems.

The GBeans platform is developed in Geronimo, the open-source J2EE-certified Web Application Server from Apache. It uses a module layer based on a tree of modules, faithful to the traditional Java class loading architecture. The deployment unit is a configuration, which is described by an XML file, called a configuration, containing a list of JAR files for the local class path as well as a list of other child configurations.

To support GBeans, one has to first develop a search policy, which is straightforward in this case since the model is simply a tree of modules. A module is therefore resolved if its parent module is resolved. Furthermore, the delegation model is also simple since it is based on the traditional parent class loader delegation. A second step is to write the management of module contents, that is, downloading GBeans configurations. This can be the actual code from GBeans since we do not impose a physical layout in the file system. For the content loader, one only needs to wrap the current GBeans class loading scheme based on a *URLClassLoader*.

The new release of GBeans, called XBean [2], is moving toward a Directed Acyclic Graph (DAG) for its module layer, with module-level granularity. Not much needs to be changed from the above in order to accommodate a DAG. The resolver needs to resolve a module only if required modules are resolved. Because dependencies are on modules, through explicit names and versions, the resolver remains fairly simple. The search policy is also quite simple as it relies on module-level wires.

For OSGi technology, a deployment unit is a bundle, which is a JAR file with a manifest containing module metadata. At runtime, a bundle is a module following a fairly complex dependency model. The OSGi R4 specification defines two levels of granularity for dependencies: Java packages or bundles. Furthermore, the resolution is fairly complex as dependencies are flexible and powerful. Hence, one

major challenge is to develop the resolver for the dependency model; the resolver is basically a backtracking constraint resolver. The Equinox project, from Eclipse, has implemented one in the open source world.

Regarding the bundle content, there is no major difficulty, but different systems use different approaches. Traditionally, OSGi implementations keep bundles as JAR files for class and resource loading, but need to extract embedded JAR files and native libraries. Some implementations also support exploding bundle JAR files into the file system. All of these approaches are easily supported through implementation-specific content loaders.

5 Conclusion

In this paper, we presented a framework for building and integrating diverse module systems. Our approach promotes the separation between mechanisms and policies. It also promotes the ability to provide custom implementations for these core mechanisms. Two key aspects are that it separates the dependency model from actual class loading as well as it separates the reification process for classes from issues of module content management. Overall, this approach provides unprecedented flexibility and reuse in building module systems.

Our approach is the only proposal toward the integration of diverse module systems. Our approach raises the consideration of the desired degree of interoperability among module systems. At a minimum, we promote module-level wiring but our concepts support finer granularity interoperability, such as package-level wiring. One key aspect is that we recognized and provide support for a global resolution process for module dependencies. While we enable integration, we do not impose policies or semantics; each individual module system resolves its modules as it sees fit.

The proposed approach and corresponding interfaces are the results of several years of designing and implementing module layers, with different dependency models and different class loader delegation. We believe that the proposed architecture and design is sound for core mechanisms that could be integrated in the Java Runtime Environment for supporting modules, without imposing an actual model for modules or their dependencies. Overtime, this would suggest the eventual deprecation of the class loader framework and the adoption of flexible and adaptive mechanisms for building and integrating module systems.

References

[1] OSGi Alliance. Osgi service platform core specification release 4, 2005.

[2] Apache Community. Xbean, 2005.

[3] Apache Community. Geronimo beans wiki, 2006.

[4] Richard S. Hall. A policy-driven class loader to support deployment in extensible frameworks. In *Proceedings of the 2nd International Working Conference on Component Deployment*, 2004.

[5] C. Laffra, S. Foley, , and J. McAffer. Packaging eclipse rcp applications, 2004.

[6] Java Community Process. Jsr 277: Java module system, 2005.

[7] Java Community Process. Jsr 291: Dynamic component support for java se, 2006.