RC23927 (W0603-204) March 31, 2006
Computer Science

IBM Research Report

A Java Framework for Runtime Modules

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

Richard 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 Runtime Modules

Olivier Gruber[1] and Richard S. Hall[2]

[1]IBM Research – ogruber@us.ibm.com
[2]Laboratoire LSR – richard.hall@imag.fr

Abstract. We present the design of core mechanisms for building module frameworks from reusable components. Our approach promotes flexibility and modularity, separating the different facets of a module framework: a way to load classes or resources for applications, a delegation model between class loaders, and the actual class loading. Our design, implemented in Felix (Apache), can model the module layer of popular frameworks such as GBeans, XBean, or the OSGi framework.

1 Introduction

In recent years, there has been growing interest in runtime modules within the Java community [2][3]. Originally, Java had no such concept and later adopted class loaders as its runtime module model. 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 loading (i.e., reifying class bytes into class objects). To support other runtime components and services, different Java communities have proposed, above Java class loaders, different runtime module frameworks, such as GBeans[1] or OSGi technology [5].

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 packaging, from simple to extended Java archive (JAR) files as well as other formats such as JXE files for the IBM J9 virtual machine [4].

This paper introduces our work on a comprehensive Java framework for runtime modules, which continues previous work in this area [1]. It provides core mechanisms for supporting heterogeneous module semantics through policies. This framework is the result of having to deal with such diversity 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.

This position paper is structured as follows. In section 2, we introduce the overall architecture of our approach for runtime modules and discuss our interfaces and the underlying design. In section 3, we quickly compare our approach with the state of the art and illustrate the flexibility of our approach. In section 4, we conclude.

[1] http://wiki.apache.org/geronimo/GBeans

2 Architecture and Design

Our focus is on the design of the core mechanisms for building module frameworks from reusable components. Our approach promotes flexibility and modular development, separating the different facets of a module framework: actual class loading from a physical container (`IContentLoader`), delegation model (`ISearchPolicy`), and interface for Java applications to load classes and resources (`IModule`). The corresponding architecture is depicted in figure 1.

For a given framework implementation, one of the first steps is to decide on a dependency model. A dependency model understands the dependencies among modules. It knows how to resolve a module, that is, to satisfy its dependencies. For each resolved module, it can provide a search policy object (`ISearchPolicy`) that represents its delegation model to other modules for class and resource loading.

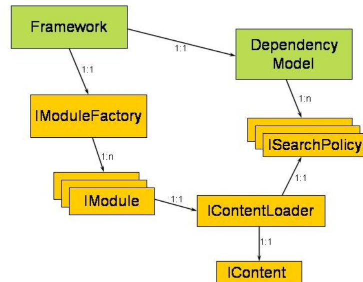

Fig. 1. Architecture

```
public interface ISearchPolicy {
    public Object[] definePackage(String name);
    public Class findClass(String name)
        throws ClassNotFoundException;
    public URL findResource(String name)
        throws ResourceNotFoundException;
    public String findLibrary(String name);
}
```

Fig. 2. Search policy

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 the metadata format used to express dependencies; this is internal to the implementation of the module framework and its dependency model.

The module framework uses a module factory (`IModuleFactory`) to create modules (`IModule`). Once a module has been created, the framework needs a content loader (`IContentLoader`) for it. The content loader is the entity that knows how to reify Java types and resources from the actual module content (`IContent`). A content loader usually leverages a Java class loader for this, but it does not have to; some Java runtime environments provide special support such as the IBM J9 virtual machine.

```
public interface IModuleFactory {
    public IModule[] getModules();
    public IModule getModule(String id);
    public IModule createModule(String id);
    public void removeModule(IModule module);
    public void setContentLoader(IModule module,
        IContentLoader contentLoader);
    public void addModuleListener(ModuleListener l);
    public void removeModuleListener(ModuleListener l);
}
```

Fig. 3. Module factory

```
public interface IModule {
    public String getId();
    public IContentLoader getContentLoader();
    public Class getClass(String name);
    public URL getResource(String name);
}
```

Fig. 4. Module

The `IContent` interface abstracts the storage details of the actual contents of a module, which is one large point of diversity among existing module frameworks. For example, this can be simply downloading JAR files from an HTTP site to a local file system, where they may or may not be expanded. The JAR files may also be complex, embedding native libraries and other JAR files for the module's local class path (e.g., OSGi bundles). Our approach does not mandate any specific download format or local storage policy.

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

Fig. 5. This is the content loader caption

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

Fig. 6. This is the content caption

The last step is to provide the content loader with its search policy object, which supports delegated class loading. The purpose of content loader is the reification of the local classes and resources for the Java Virtual Machine. When a class is needed, the content loader checks that it has not already loaded it, if not, it delegates to its search policy. The search policy knows the delegation model and may ask another module to load the class if necessary. If the search policy does not return a class, the content loader attempts to load it locally, from its own content.

3 Examples

This section discusses several different models for modules from different successful systems. Taken together, they cover a wide range of modularity and illustrate the flexibility of our approach as a low-level mechanism for building module frameworks.

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 agent that knows how to download a configuration and the JAR files it needs. These JAR files can be simply stored in the file system and a simple URLClassLoader will suffice for implementing the content loader.

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. Furthermore, the delegation for class loading has to delegate to those required modules.

[2] http://www.xbean.org

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, which needs to include a backtracking constraint resolver.

Regarding the bundle contents, 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 native libraries. Some implementations support exploding bundle JAR files into the file system. As long as one knows how to create a class loader for the content layout, the approach works.

4 Conclusion

In this short paper, we have presented core mechanisms for supporting modular implementations of module frameworks. The separation between mechanisms and policies, as well as the ability to provide custom implementations for the core mechanisms, provide unprecedented flexibility and reuse. One key aspect is that the approach separates the dependency model from actual class loading. It also separates the reification process for classes from issues of module content management.

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 and their dependencies. Overtime, it would simplify the Java class loaders since they could focus on reification of classes rather than dependency model or storage issues.

5 References

1. R.S. Hall. "*A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks*," Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004), May 2004.
2. Java Community Process. "*JSR 277: Java Module System*," http://www.jcp.org/en/jsr/detail?id=277, June 2005.
3. Java Community Process. "*JSR 291: Dynamic Component Support for Java SE*," http://www.jcp.org/en/jsr/detail?id=291, February 2006.
4. C. Laffra, S. Foley, and J. McAffer. "*Packaging Eclipse RCP Applications*," IBM Corp., 2004.
5. OSGi Alliance. "*OSGi Service Platform Core Specification Release 4*," http://www.osgi.org, August 2005.