MJ: A Rational Module System for Java and its Applications

John Corwin, David Bacon, David Grove, Chet Murthy
Modularity in Java

- Classes
  - OO-style encapsulation and scoping

- Packages
  - Static namespaces

- Classloaders
  - Dynamic namespaces
Problems with Java’s Existing Mechanisms

§ Classes
   – Too fine-grained: a component contains many classes

§ Packages
   – Not hierarchical (no nesting)
   – Expose same interface to all clients

§ Classloaders
   – Very complex to configure correctly
   – No mechanism to ensure that static and dynamic classpath are compatible!
     • NoClassDefFoundError
     • Find a class, but not the same one as was found at compile time!!!
Properties of a Solution

§ Minimal changes to existing Java code and tools
  – Large body of existing deployed Java code
    • *MJ only changes code that directly manipulates classloaders*
  – Use existing IDEs, compilers, JVMs
    • *No syntax changes; modules defined in separate file*
    • *MJ still has packages (in addition to modules)*
  – Fully support reflection and dynamic module creation
    • *Restrictions on name mangling, require dynamic module interface*

§ Basic modularity goals
  – Statically guarantee all static cross module references are resolvable
  – Hierarchical components
  – Finer-grained access control and ability to provide multiple interfaces
Outline

$ The Problem in Detail
$ The MJ Module System
$ Applications & Results
$ Related Work
$ Conclusions
Problem: Packages not Hierarchical

- java.lang is not “related” to java.io

- Performance of java.io suffers because it has the same view of java.lang.String as arbitrary untrusted code.
  - Gets copy of backing char[] instead of accessing directly

- Would like to give java.io a safe backdoor:

```java
package java.lang;
public class StringInternals {
  public static char[] getValue(String s) {
    return s.value;
  }
}
```
Problem: Packages Provide One Interface

- Some trusted code is not in java.*
  - Even hierarchical packages don’t solve this problem

- Enable ApplicationServers (trusted middleware) to use StringInternals

- General problem to be solved
  - Export different interfaces to different clients
  - Visibility of an interface by a given client controlled by external policy
Problem: Multiple Versions

- Large applications often require multiple versions of a package
  - XML parser, XSL processor
    - XML APIs are changing very rapidly
    - Code depends on exact versions of Xerces, etc
  - CORBA ORBs differ dramatically
    - talk to multiple servers written using different ORBs

- Source of many complex configuration problems in the field
  - current "state of the art" is to use classloaders & classpaths
Lesson from History: Linkage is Power

§ AIX XCOFF
   – The basic C language exhibits no real modularity: any programmer can reference any non-static function
   – The AIX Linker can impose controls, at link-time, on which symbols a DLL can import/reference, and which symbols it can export for reference by others

§ CAML-Light
   – In 1990, ML implementations did not support separate compilation in any standard sense
   – ML phrases were loaded into an interpretive 'top-loop', and could reference anything already loaded
   – CAML-Light treated files as compilation/linkage units with lists of other units from which to import, as well as lists of which names (types, functions, values) to export

Disciplined control of the linkage process imposes basic modularity
Linkage in Java

- Classloaders are the linkage mechanism in Java

- Classloaders are lowlevel
  - Useful implementation mechanism
  - Should not be directly manipulated by programmers

- Abstract datatypes, separate compilation, and a limited first-order modularity can be achieved through disciplined, systematic control of the linkage process.

- Cross-component name resolution must be carefully specified and controlled
Outline

$ The Problem in Detail

$ The MJ Module System

$ Applications & Results
$ Related Work
$ Conclusions
MJ Overview

$ Basic tooling
  - Module compiler
  - Module aware javac
  - Module repository

$ Module definitions
  - External to Java source files

$ Runtime system
  - Module classloader
  - java.lang.Component
Basic MJ Tooling

Module Repository

Java File
Class File
Java File
Class File
Java File
Class File
modjavac
javamodc
imports
mar
mar
mar
Module File
Class File
Module File
Class File
Jar File
Sample Module Definition (.jm file)

provides "catalina.jar";

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina {
   public static void load() { .... }
   public static void main(String[] args) { ... }
}


Sample Module Definition: Module Name

provides "catalina.jar";

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;
hide * in *
export org.apache.catalina.* to
  webapp;
export org.apache.catalina.servlets
  to servlets;
forbid org.apache.catalina.* in *

module catalina {
  public static void load() { .... }
  public static void main(String[]
    args) { ... }
}
Sample Module Definition: Java Resources

```
provides "catalina.jar";

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *;
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *;

module catalina {
  public static void load() { .... }
  public static void main(String[] args) { ... }
}
```

$ class statements declare that this module provides a particular class
  class javax.servlet.Servlet;

$ path statements add a .jar file or directory to this module's list of
  locations to search for classes
  path "myApp/tools";

$ provides statements combine the above two steps, adding a .jar file or
  directory to the search path, and providing each class found therein
  provides "catalina.jar";
Sample Module Definition: Import

provides "catalina.jar";

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *

module catalina {
   public static void load() { .... }
   public static void main(String[] args) { ... }
}
Sample Module Definition: Export

provides "catalina.jar";

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *
export org.apache.catalina.* to
  webapp;
export org.apache.catalina.servlets
to servlets;

forbid org.apache.catalina.* in *

module catalina {
  public static void load() { .... }
  public static void main(String[]
    args) { ... }
}
Sample Module Definition: Forbid

provides “catalina.jar”;

import * from xerces;
import * from bootstrap;
import com.sun.tools.* from tools;

hide * in *
export org.apache.catalina.* to webapp;
export org.apache.catalina.servlets to servlets;

forbid org.apache.catalina.* in *

module catalina {
  public static void load() { .... }
  public static void main(String[] args) { ... }
}

Some modules may want strict control over which classes are permitted in a namespace

forbid statements prohibit any importer from implementing a set of classes and from importing such classes from any other module
Sample Module Definition: load/unload/main

- `provides “catalina.jar”;
- `import * from xerces;
  import * from bootstrap;
  import com.sun.tools.* from tools;
- `hide * in *;
  export org.apache.catalina.* to webapp;
  export org.apache.catalina.servlets to servlets;
- `forbid org.apache.catalina.* in *
- `module catalina {
    public static void load() { .... }
    public static void main(String[] args) { ... }
  }

$ Optional parts of module declaration
$ load is called once per module, when the module is first loaded
$ unload is called when the module is unloaded, similar to a finalize() method
$ The main method, which takes a string array as an argument, is called when the module is invoked as a main program
Other Module Declarations

$ seal and unseal declarations to control ability to subclass
    seal MyClass in *;

$ resource statements to add resources to a module

$ dynamic export and hide used by Component.forName
MJ Runtime Support

§ A module classloader that validates classloading requests
  - Uses module graph built from module repository
  - Enforces import/export relationships

§ java.lang.Component
  - forName
  - getResource

§ Use Java classloaders to implement namespaces
  - each loaded module gets its own classloader instance

§ JVM is unchanged
Outline

- The Problem in Detail
- The MJ Module System

- Applications & Results
  - Experiences with Apache Tomcat
  - Servlet microbenchmark

- Related Work
- Conclusions
Modularized Apache Tomcat

- Fully-modularized Apache Tomcat (v4.1.18) and all supporting libraries
  - Removed all explicit manipulation of classloaders
  - Exact same functionality as original code

- Defined 30 modules (following jar file structure of Tomcat distribution)

- Changed roughly 400 of 160,000 lines of code
Changes to Tomcat

$ Use of thread context classloaders
  - Used when library needs to load class using client's classloader
  - Extremely error prone

$ “Ad hoc” modularity via conditional selection of classloaders

```java
if (condition)
    clazz = loader1.loadClass(name);
else
    clazz = loader2.loadClass(name);
```

$ Dynamic loading of servlets and JSPs
  - Done via dynamic component support of java.lang.Component

$ More cases in the paper
Servlet Microbenchmark

- Implemented java.lang.StringInternals class
  ```java
  public class StringInternals {
    public static final char[] getValue(String s) {
      return s.value;
    }
  }
  ```

- MJ makes StringInternals available only to trusted code

- Modified java.io.OutputStreamWriter to exploit StringInternals

- Achieved 30% speedup in Servlet microbenchmark on JDK 1.3.1
  - Run on modularized tomcat
  - Servlet generates an HTML multiplication table
Outline

$ The Problem in Detail
$ The MJ Module System
$ Applications & Results

$ Related Work
$ Conclusions
Related Work

- **Component Systems**
  - Microsoft COM object model
  - OSGi bundles; Eclipse plugins

- **Module Systems**
  - Units [Findler & Flatt]; ML functors [MacQueen]
  - Ada, Modula-2, Modula-3, Haskell

- **Java Extensions**
  - Jiazzi [McDirmid et al]; Component J [Seco & Caires]; [Bauer et al]
  - JavaMod [Ancona & Zucca]; Multi-Java [Clifton et al]
Future Work

- Implement pass thru modules: virtualization
- Apply to truly large scale programs
- Apply MJ to core JDK:
  - 1.5 adds XML support: xerces, xalan
    - Trouble if application needs different version!
  - Exploit multiple interfaces to safely improve performance
Conclusions

 MJ enables
  - Static guarantee that all static cross module references will be resolvable
  - Hierarchical components
  - Fine-grained access control and multiple interfaces to a component
  - Versioning of modules (each mar file is given a unique id)

 MJ can be easily applied to existing Java code
  - Reuse existing Java tooling
  - All dynamic features supported (reflection, dynamic module creation)
  - Only code that directly manipulates classloaders needs to be changed

 Successfully ported Tomcat (160kloc) to MJ changing only 400 lines