MJ: A Rational Module System for Java and its Applications

John Corwin David F. Bacon David Grove Chet Murthy

IBM

ABSTRACT

While Java provides many software engineering benefits, it lacks a coherent module system and instead provides only packages (which are primarily a name space mechanism) and classloaders (which are very low-level). As a result, large Java applications suffer from unexpected interactions between independent components, require complex CLASSPATH definitions, and are often extremely complex to install and maintain. We have implemented a module system for Java called MJ that is implemented with class loaders, but provides a much higher-level interface. High-level properties can be specified in a module definition and are enforced by the module system as new modules are loaded. To experimentally validate the ability of MJ to properly handle the complex module inter-relationships found in large Java server systems, we replaced the classloader mechanisms of Apache Tomcat 4.1.18 [27] with 30 MJ modules. The modified Tomcat is functionally identical to the original, but requires no CLASSPATH definitions, and will operate correctly even if user code loads a different version of a module used by Tomcat, such as the Xerces XML parser [31]. Furthermore, by making a small change to the Java core libraries enabled by MJ, we obtained a 30% performance improvement in a servlet microbenchmark.

Categories and Subject Descriptors

D.2.3 [**Coding Tools and Techniques**]: Object-oriented programming; D.3.2 [**Language Classifications**]: Object-oriented languages—*Java*; D.3.3 [**Language Constructs and Features**]: Modules, packages

General Terms

Design, Experimentation, Languages, Performance

Keywords

Java, Modularity, Components, Language Design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'03 October 26–30, 2003, Anaheim, California, USA
Copyright 2003 ACM 1-58113-712-5/03/0010 ...$5.00.

1. INTRODUCTION

When constructing large software systems, it is desirable to decompose the system into collaborating components with well-defined interfaces between them. Each component should specify declaratively, using mechanisms that can be statically checked and dynamically enforced, what functionality of other components it depends on and what functionality it makes available to other components. Components must be hierarchical: a component must be able to contain an arbitrary number of other components. Internal components may either be hidden or exposed as part of the containing component's interface.

The Java™ programming language [12] provides classes and packages for encapsulation and static scoping, and classloaders [18] for dynamic namespaces. Unfortunately, the combination of these language features does not adequately support large scale, component-oriented programming. Classes are too fine-grained of a mechanism for defining components; even small components will typically contain a number of classes. Java packages are not hierarchical and thus prevent the definition of nested components.

As a result of the limitations of Java's static mechanisms, the common practice in large Java applications is to attempt to control component visibility and inter-dependencies through clever usage of classloaders and CLASSPATHs. The main disadvantages of this approach are its complexity and the complete lack of static checking. In particular, the CLASSPATH used to statically compile the classes of a component must be compatible with the runtime CLASSPATH of the classloader that loads the component. If there are discrepancies, then when the component attempts to locate a class on which it depends that is provided by another component it may fail with a NoClassDefFoundError. Even worse it may find a different version of the class than was found at static compile time (possibly even a completely unrelated class that just happens to have the same name!) and fail with any number of linkage or runtime errors. A yet more troubling error, which happened in the field, is infinite-loops of class-delegation between classloaders whose delegation relationships form a proper (i.e. not acyclic) graph, a structure which is permitted by the Java 2 classloading model.

Based on our experience with large Java server applications, we have identified several common problems caused by the lack of a sufficiently strong module system in Java. The proliferation of classloaders and CLASSPATHs makes large Java applications complex to develop, deploy, understand, and maintain. The implicit dependency between compile time and runtime CLASSPATHs is particularly pernicious and the source of subtle failures. Finally, there is no mechanism to define a single component that provides a wider interface to a specified set of trusted components than the "safe" interface it provides to all components. This is the root cause

of many inefficiencies in the Java core libraries and in Java application servers.

One of the critical characteristics of any acceptable solution to these problems is that it *must* be possible to retrofit it to the large and fast-growing body of extant and deployed commercial Java code. A solution that requires nontrivial changes to Java code is, *a priori*, not a solution. The design of MJ was purposely constrained by the requirement that, aside from code which explicitly manipulated classloaders, all other code could be used unmodified, usually down to the level of the classfiles themselves. Likewise, it must be possible using straightforward modifications, to make code that does manipulate classloaders, compatible with MJ.

Furthermore, any acceptable solution must provide a comprehensible modularity semantics, as well as the ability to perform static checking (and eventually perhaps, static compilation). The static checking we have in mind must include the ability to verify that static references will be satisfied – that is, at a minimum, that no `ClassNotFoundException` or `NoClassDefFoundError` will occur as a result of static dependencies. This pretty much rules out any mechanism in which the set of classloaders is not fixed and solely controlled by the module system itself. To require that application-server or IDE implementors also understand the complexities of the module system implementation (in order to write their own classloaders) is unacceptable.

To address these problems, we have implemented a module system for Java that is implemented with classloaders, but provides a much higher-level interface. High-level properties can be specified in a module definition and are enforced by the module system as new modules are loaded. The main contributions of our work are

- The definition of the MJ module system. MJ draws upon previous work in module and component systems to define a simple, easy to use, yet powerful module system that can easily be adopted in legacy Java code.
- An implementation of MJ including a module compiler, a module-enabled `javac` which does static import-checking, and a custom classloader that implements the dynamic aspects of the system.
- An experimental validation of the ability of MJ to properly handle the complex module inter-relationships found in large Java server systems with minimal source code changes. We have completely replaced the classloader mechanisms of Apache Tomcat with MJ modules. Doing so required changing approximately 400 of its 166,000 lines of Java source code.
- To demonstrate the potential efficiency gains enabled by MJ's ability to safely export richer interfaces to selected components, we made a small change to the Java core libraries that resulted in a 30% performance improvement in a servlet microbenchmark.

The next section describes in more detail the weaknesses of Java's existing modularity mechanisms. Section 3 presents the MJ module system and Section 4 details how MJ is implemented by using Java's existing classloader mechanism. Section 5 relates our experiences in modularizing Apache Tomcat. Sections 6 and 7 discuss related and future work and Section 8 offers our conclusions.

2. THE PROBLEM

In this section we discuss why the existing modularity mechanisms in Java are insufficient. The fundamental weakness is that in a Java program, when a class `C` refers to another class `A`, the reference is processed by a *classloader*, regardless of whether the reference is within the same module or component, or whether the reference is from some other module. Invocations of `java.lang.Class.forName`, which is used to dynamically resolve references to classes, are processed in the same way.

Precisely put, there is no distinction between *linking* (the process of resolving a textual reference to a name, found in some piece of code which is in some sense a part of the *current* component), and *component activation or invocation* (the process of resolving a reference to a name, imported from some *other* component). Java packages do not provide a solution to this problem – while the Java package system provides some measure of scoping and naming control during compilation, it is wholly inadequate as a basis for a module system – any class exported from its package (`public`) is automatically accessible to all other packages.

The remainder of this section illustrates several kinds of problems that arise in Java programs due to the lack of a sufficiently strong module system. We begin by showing that packages do not adequately support abstract datatypes that need to expose different interfaces to different clients. The next two sections discuss complexities that arise when an application needs multiple versions of the same class and the resulting explosion in custom classloaders and classpaths that are common practice in large industrial Java programs. The last section discusses optimization opportunities that can be missed because the programmer is unable to communicate certain kinds of static knowledge to a JIT compiler.

2.1 Abstract Datatypes

The Java programming language is one in which every object is dynamically and strongly typed. In short, whenever an object has a concrete type `C`, even if a method is passed the object with a declared "abstract" type `A`, as long as the method "knows" `C` (which, in this context, technically means that it can reference the class `C` statically), it can "downcast" the object from class `A` to class `C`. The only constraint on the ability of a method to downcast an object from a declared type, to its actual concrete implementation type, is that, if the object's actual implementation class is not available, then the downcast cannot occur. This can happen for two reasons:

- The method in question was compiled without access to the code (source or classfile) of the implementation class. This is an unlikely scenario and could easily be worked around by compiling against a dummy classfile.
- The implementation class `C` is provided by a classloader which is not accessible from the method attempting the downcast; in this case, even if the classloader of the current method *can* reference a class named `C` which is constructed from a classfile identical to that of the implementation class `C` of the object, these two classes will not be equal, since they are loaded by two different classloaders.

This latter mechanism is the only means in Java to ensure that an object can have an implementation class `C`, providing only an abstract interface `A` to some subset of its invokers, while providing a more extensive interface to some other subset of invokers. Note that the oft-used mechanism of *delegation*, in which the concrete implementation is wrapped in a *delegator* class, exposing a weakened interface, does not suffice, since once we make the method which allows access to the concrete contained object available to *some* other classes, we make it available to *all* other classes, effectively defeating our desire for enforced abstraction.

Consider the class `java.lang.String`, which encapsulates a pointer to a **`char`**-array, as well as an integer offset and length.

```java
package java.lang;

public final class String {
    private char value[];
    private int offset;
    private int count;
    ... various methods ...
    protected static final char[] getValue(String s) {
        return s.value;
    }
    protected static final int getOffset(String s) {
        return s.offset;
    }
    protected static final int getCount(String s) {
        return s.count;
    }
}

public class StringInternals {
    protected static final char[] getValue(String s) {
        return String.getValue(s);
    }
    protected static final int getOffset(String s) {
        return String.getOffset(s);
    }
    protected static final int getCount(String s) {
        return String.getCount(s);
    }
}

// Default implementation
public class java.io.OutputStreamWriter {
    ... other methods ...
    public void write(String str, int off, int len)
        throws IOException {
        /* Check the len before creating a char buffer */
        if (len < 0)
            throw new IndexOutOfBoundsException();
        char cbuf[] = new char[len];
        str.getChars(off, off + len, cbuf, 0);
        write(cbuf, 0, len);
    }
}

// New implementation using StringInternals
public class java.io.OutputStreamWriter {
    ... other methods ...
    public void write(String str, int off, int len)
        throws IOException {
        if (len < 0)
            throw new IndexOutOfBoundsException();
        write(StringInternals.getValue(str), off, len);
    }
}
```

Figure 1: A skeleton of `java.lang.String` illustrating how `StringInternals` can be used to define more efficient I/O routines.

Java's package-level visibility only allows classes in `java.lang.*` to see package-visible instance-variables, yet, for efficiency, we would like the I/O hierarchy in `java.io.*`, char-to-byte conversion classes in `sun.io.*`, and WebSphere / WebLogic / Tomcat-specific classes in further-distant packages to be allowed access to instance-variables of `java.lang.String`, while forbidding such access to all (servlet / EJB) application code. This would allow us to write significantly more efficient code to output String objects to the network, amongst other things, and, we believe, significantly improve the efficiency of Java server applications.[1]

Concretely, as shown in Figure 1, it would be extremely useful to gain access to a `String`'s instance variables, from classes such as `java.io.OutputStreamWriter`, so that, rather than invoking a copy operation to acquire the contents of a String, they could directly copy from the String's `value[]`.[2]

The problem with this approach is that (barring clever use of classloaders) if any class outside package `java.lang` can see the class `java.lang.StringInternals`, then *all classes* can see it.

A different means of arriving at the same effect would be to have a final class `StringImpl`, which would be a subclass of the class `String`. The instance-variables currently on `String` would move down to the subclass, and would become `public`. The problem, again, is that for `StringImpl` to be visible to selected other packages, it must be visible to all other packages, which, again, leads to the aforementioned failure of abstractness of the datatype `String`.[3]

2.2 Multiple Versions of the Same Class

In any modern web-application server, there is copious use made of XML, and therefore one almost invariably finds an implementation of the Xerces XML parser somewhere in the classpath. However, this version is often a version incompatible with application code. For example, the Apache Axis SOAP implementation requires a different XML parser from that found in WebSphere 4.0.3. While it is possible to work around these problems, it would be much simpler if there were a way to control visibility of such components as the XML parser, to make it simple to control which version was seen by various application code.

In a like manner, whenever dealing with large legacy distributed transaction-processing deployments, one often finds that one must deal with multiple CORBA [5] ORBs. Unfortunately, in this case, the class `org.omg.CORBA.ORB`, as recently as the JDK 1.3, defines a "singleton" instance-variable to hold the only copy of the ORB in the JVM address-space (figure 2). This means that if we wish to have two ORBs in the same address-space (for example, to allow calls to both IONA Orbix [15] and Visigenic Visibroker [28] servers from a WebSphere application-server or client), the two ORBs must be so written (or configured) to co-operate in their use

```
package org.omg.CORBA;

abstract public class ORB {
    static private ORB singleton;
    ...
}
```

Figure 2: A skeleton of org.omg.CORBA.ORB

of this singleton instance-variable – an unlikely occurrence.

What is needed here is

- A set of client classes, C1, C2, C3, which use the ORB named `com.vendor1.ORB`; this will necessitate a *copy* of the classes associated with `org.omg.CORBA.ORB` that are initialized such that the static singleton ORB variable points at the instance of type `com.vendor1.ORB`.
- Another set of client classes, D1, D2, D3, which use the ORB named `com.vendor2.ORB`; this will also necessitate a copy of the `org.omg.*` classes in the same manner; this time, the static singleton ORB variable must point at the instance of type `com.vendor2.ORB`.
- Some set of "common" client classes, A1, A2, A3, which *access* both C1, C2, C3, and D1, D2, D3, but *not* either of the ORB classes. These common classes are the driver which access both back-end servers, thru the respective sets of access classes.

If an application does need to engage in CORBA communications with more than one back-end server, code must be delicately written using separate classloaders to control the visibility of classes in the `org.omg.CORBA` package to the above classes.

2.3 Shared Class Libraries and Application Server Classpaths

Worse, even if we can arrange for such "co-existence", we are further faced with ensuring that each piece of application code is "linked" against the appropriate set of ORB classes; indeed, it is problematic to even ascertain which classes will be used, from case to case, unless custom classloaders are used – a recourse fraught with peril, as custom classloaders are not trivial to implement or use correctly.

An issue that always arises when using multiple classloaders is correctly defining where each classloader searches for binary class files, i.e. its classpath. Java application servers typically have several custom classloaders, and thus contain many classpaths; for instance, the WebSphere J2EE server searches, in order

- a module classpath, for each EJB
- an application extensions classloader
- a standard Java classloader
- a WebSphere `ws.ext.dirs` classloader

Some classes, e.g., database drivers, must be added to the `ws.ext.dirs` directory, rather than in other locations or classpaths. The anecdotal evidence for inordinate complexity in placing code in the proper classpaths is large, as is the body of documentation on proper placement of classes – it appears clear that enough programmers and administrators get this wrong, often enough, that copious whitepapers [25] have been written to try to teach programmers how to do this.

[1] Our industrial experience with customer code confirms this hypothesis – we have sped up customer code by 2.5x thru a combination of copy-avoidance and memory-allocation-avoidance in strings, string-buffers, and output-streams, alone, and have found that several large programs in the retail brokerage community are gated by their output-string-manipulation performance.

[2] Since the accessors in `java.lang.StringInternals` are static final methods, we can presume that a properly aggressive JIT would inline them.

[3] Of course to make this abstract datatype design effective, we would need to perform significant change to the current JVM code, as well as to introduce static factory-functions to create strings, in place of the current constructors; the example is meant only to be expository; we implemented the first scheme.

In a general sense, the proliferation of "classpaths" and "custom classloaders" in Java application-servers leads to great confusion on the part of application-writers, and there is ample evidence in the enterprise application-server community, both amongst server developers, and application developers, that some simpler and more straightforward mechanism is necessary.

2.4 Missed Optimization Opportunities

When implementing JIT (just-in-time) compilers, it is often the case that one would like to make assumptions about the final-ness (precisely put, the "non-overridden-ness") of various methods. Indeed, some recent JIT compilers [7, 16] have taken to assuming that many methods which are not marked `final` are indeed so, until such time as a class is loaded which overrides the method in question. At that point, the previously JITed code is flushed, and if the previously JITed methods are still in progress, they are completed in interpretive mode. There are both obvious efficiency as well as complexity issues in such a strategy.

Yet, in fact, especially for Java server applications, we *know* the subclassing relationships statically, albeit scattered thru large collections of jar-files. If we could somehow express, in a static manner "near" a jar-file, that certain classes in that jar-file could only be subclassed by code in certain other jar-files (or, better, not at all), the JIT could use such information to compile much more efficient code, without needing to make provision (and contain code) for flushing invalid method-bodies when optimistic assumptions about the class hierarchy are invalidated.

Even for client-side Java systems that load code dynamically, a large body of the Java code is statically available, and, again, there are strong restrictions on which classes can be subclassed, which all lead, again, to static information which, if collected and presented to a JIT, could result in better compiled code.

3. A MODULE SYSTEM FOR JAVA

In this section we present the MJ module system. We begin by describing the module language and give a small example of an MJ module description. We then turn to the distinction between the static and dynamic module graph and to the compilation and execution of MJ programs. Finally, we discuss some unimplemented extensions to the module system.

3.1 A Programmer's View

Instead of specifying a classpath to the JVM, or creating custom classloaders for various components, we will instead create a *component registry*. The component registry contains both descriptions of modules (metadata) and archives of the modules' provided classes.

A module's metadata consists of the following information:

- Which classes this module provides, and where these classes are archived
- Which other modules does this module depend on, and which classes do we require from each of these modules
- Access control for this module's provided classes – which classes are made visible to other modules, which classes do we allow subclassing of in other modules, and which package-prefixes do we restrict
- Initialization code, called when a module is first loaded

Module descriptions are provided by .jm files, one per module. The format of a .jm file is shown in Figure 3. The salient details of the module descriptions are

```
jm ::= statement-list
     | statement-list module-initializers

statement-list ::= statement ';' statement-list
                 | e

statement ::= path quoted-string
            | class class-scope
            | resource quoted-string
            | provides quoted-string
            | import         class-scope from module
            | export         class-scope to   module
            | hide           class-scope from module
            | seal           class-scope in   module
            | unseal         class-scope in   module
            | forbid         class-scope in   module
            | unforbid       class-scope in   module
            | dynamic export class-scope to   module
            | dynamic hide   class-scope from module

class-scope ::= class-name | package-prefix

class-name ::= package '.' class-name
             | class

package-prefix ::= package '.' package-prefix
                 | '*'

module ::= module-name | '*'

class ::= literal
package ::= literal
module-name ::= literal | guid

module-initializers ::= 'module' module-name
                        '{' <java code> '}'
```

Figure 3: Module Description Format Specification

- `path` statements, each naming a .jar file or directory from which class files provided by this module are loaded.

- `class` statements, each listing a class-name or a package prefix, e.g.

```
class Foo;
class MyPackage.*;
```

  The class statements together constrain the class archives specified by the `path` statements.

- `provides` statements, which are a concise means of specifying a `path` statement and multiple `class` statements. Like a `path` statement, a `provides` statement names a .jar file or directory of class files, but also adds a `class` statement for each class contained in the .jar or directory.

- `import` statements, each listing some class or package-prefix, which specify that the named class, or classes matching the specified prefix, exported from the named component, are imported into this component. If two import statements both provide classes with the same name, then the later statement overrides, but one could imagine that that would be an error.

- `export` and `hide` statements, each listing some class or package-prefix and a named module. `export` specify that the named class, or classes matching the specified prefix, in this component, are exported, possibly to only explicitly-specified components. `hide` is the functional opposite of `export`, and specifies that the given class, or classes matching the given prefix, are not exported to the given module.

- `seal` and `unseal` statements, which list a class or package-prefix and a module. Often we wish to allow reference to classes in some framework, but to not allow subclassing. Sometimes, when there are *no* subclasses of some class, we can mark the class `final`; however, when we wish to do this to a hierarchy, there are no linguistic mechanisms available. We propose the ability to *seal* a hierarchy, optionally to specific components, e.g.

```
seal COM.ibm.db2.jdbc.app.* in webapp;
```

  as a means of ensuring that, though the exporting component has an entire hierarchy of classes implementing a JDBC driver, the named target component, while it may *reference* the designated classes, cannot subclass them.

  `unseal` statements are the opposite of `seal`, making previously sealed class hierarchies available for subclassing.

- `forbid` and `unforbid` statements, again specifying a class or package-prefix and a module. The classloader implementation rule that a child classloader must first allow its parent classloader to provide a class, before it attempts to resolve the class itself, is based on the desire to prevent child classloaders from providing different implementations of base JVM classes such as `java.lang.String`. More generally, there are important behavioral invariants embedded in informal rules, like, "the package COM.ibm.db2.jdbc.app.* is controlled by the DB2-JDBC implementation, and application code must not insert classes there." To enforce such rules, we can imagine a "Forbid" directive,

```
forbid java.* in *;
forbid COM.ibm.db2.jdbc.app.* in webapp;
```

```
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() {
    System.setProperty(
      ``javax.xml.parsers.SAXParserFactory'',
      ``org.apache.xerces.jaxp.SAXParserFactoryImpl'');
  }

  public static void main(String[] args) {
    org.apache.catalina.startup.Bootstrap.main(args);
  }
}
```

Figure 4: Example Module Description

  The first directive forbids any importer of the component in which it is found, from implementing, or importing, any class in "java.*", from any package other than this one. The second directive does the same, but only for the specified importing package. As with the other directives, one can imagine an "inverse" directive, "Unforbid", with the obvious meaning. It is important that an importing component can neither implement, nor *import* classes in the forbidden package-hierarchy, since otherwise, it would always be possible to work around the constraint by putting such classes into a separate module and importing them.

- `dynamic export` and `dynamic hide` statements, like `export` and `hide`, specify a class or package-prefix, but are used for dynamic class resolution via `Component.forName`.

- (Optional) Module initialization code, consisting of the static methods `load()`, called when a module is first loaded, and `main()`, called when invoking a module from the command-line using the `loader`.

An example module description is shown in Figure 4. This file defines a module called `catalina`. The `provides` statement indicates that this module provides each of the classes found in "catalina.jar". Next, the three `import` statements together specify this module's dependencies – in this example, our module depends on the `xerces`, `bootstrap`, and `tools` modules, and further specifies that only classes with the `com.sun.tools` prefix will be loaded from `tools`.

The next three statements first `hide` all provided classes, then make classes with the `org.apache.catalina` prefix visible to the `webapp` module, and classes with the `org.apache.catalina.servlets` prefix visible to the `servlets` module. Thus, `webapp` and `servlets` are the only modules that are allowed to import from `catalina`.

Next, we `forbid` the prefix `org.apache.catalina`, which means that if a module imports from `catalina`, it is not allowed to import classes with the prefix `org.apache.catalina` from any other modules.

Finally, we come to the module's initialization code. First, we specify a static `load()` method, which, in this case, sets a system property to name the class implementing the SAX parser factory. Next, we define a static `main()` method, which will be called when running this component from the command-line.

3.2 A System View

The MJ module system induces a certain notion of *naming* of modules, which has been implicit throughout this paper, and which we will make somewhat precise in this subsection. In the JVM, classes are named by their classname, with an added component, which is their *classloader*. This classloader object is known purely at runtime, and has no static status. In MJ, as with some other modularity systems for Java [8], the vast majority of (or, even, *all*) classloaders, have a static existence, and are named – they are the modules of MJ – by their classname and MJ module-name (which we refer to as a module-qualified classname (MQCN)). In short, an MJ module is something equivalent to a classloader, and its (static) relationships to other modules are represented by import and export relationships amongst classloaders. Likewise, class-objects in MJ are equivalent to their MQCNs, and thus, any class in a statically-defined module has a statically-defined name and meaning.[4]

The vast majority of such relationships are purely static – some importing class C depends on some field "x" of some exporting class A. We can think of this exportation graph as being "static," and MJ handles this in much the same manner as any static linker would. We can think of these importation/exportation relationships as forming a reference graph, much as we find in most static linkage systems, with the caveat that we specify exportation at the module-level (i.e., to "which" modules we export).

However, even in statically compiled languages, there are special facilities for dynamically resolving symbols, and we need to provide support for these as well.

A classic problem with such dynamic support, is that if all symbols in some library are dynamically resolvable, then none of them can be compiled (for instance) with special private linkages. To ensure that we can clearly distinguish which names are dynamically resolvable, from those which are note, we introduce a notion of "dynamic export", which is described in the next subsection.

The proper way to think about this facility, is to relate it to static import/export. In that case, we know, statically, which symbols each module exports, and (optionally) to which modules. Likewise, we know which symbols each module imports, and from which source modules.

With dynamic exportation, we know the former fact – which symbols each module exports, and (optionally) to which modules. We simply do not know the latter fact, until the importing module resolves symbols. By carefully constraining the set of dynamically exported symbols, the designer of a module interface can leave maximum flexibility to the implementor and compiler, while allowing the full dynamicity desired by users.

At a system-level, there are two distinct, but similar, graphs; the static reference graph, which any runtime/compiler is free to implement in as static a manner as desired, and the dynamic reference graph, for which the runtime must provide a much more dynamic resolution mechanism, whose public interface is described in the next subsection.

Finally, dynamically-created modules are *identical* to statically defined modules – the only difference between the two is their provenance. This means that it is (normally) only thru their dynamic exports, that the classes of a dynamically-created module can be accessed. Of course, if a dynamically-created module depends on statically-created modules, the runtime will ensure that those statically-created modules are linked, and that the static import/export graph remains consistent.

[4]Thus, the bytecode, structure, and all static external class references, of any class in a statically-defined MJ module, can be statically computed.

```
// Resolves the given class name in the given
// module, verifying that the resolution
// is allowed.
public static Class forName(String module,
                            String className);

// Expects a string containing a module
// identifier and a class name,
// separated by a ``:''.
// Class resolution occurs as in the
// two-argument form of the method.
public static Class forName(String mqcn);
```

Figure 5: `java.lang.Component` interface

3.3 Dynamic Modules

MJ was intentionally designed to minimize the number of modifications to existing Java code. Modifications are only needed for two purposes, dynamic class resolution and dynamic module management.

In the same manner as `Class.forName` dynamically resolves a class, we added a static method, `Component.forName`, that resolves module-qualified class names (MQCNs). A MQCN consists of a component-key or component name and a class name, and is sufficient to resolve a class in a named component.

Component.forName takes an MQCN as an argument and returns a `Class` object. The MQCN argument must have been dynamically exported to the module within which the class which is invoking Component.forName is found. This creates a distinction between static import/export (and associated Class.forName), and dynamic import/export. Figure 5 shows the interface to `java.lang.Component`.

We also provide an interface to dynamically create modules, using the module machinery we implemented for static modules in a programmatic manner. Such dynamic module management can be used, for instance, to run dynamically generated code such as that which is generated to run JSPs, or, generally, to create multiple namespaces in which independent instances of programs with static variables can be run. This interface takes as arguments everything necessary to describe a module. After a dynamic module is created, classes can be resolved from the module using the `Component.forName` interface. The dynamic module interface is shown in Figure 6.

3.4 Module and Source File Compilation

We have written two utilities, `modjavac` and `javamodc`, to generate a module from a module description file and compile a java source file with the static constraints of a module repository, respectively.

Before a module can be used, its description must be compiled with `modjavac`. Compiling a module involves parsing the module description file, locating the jar and class files the module provides, generating and compiling a java source file containing the module's static initializers, and installing the module into a repository. `modjavac` performs all of these tasks automatically. As an example, to compile and install the `catalina` module, specified by the de-

```
public class MJ.Dependency {
        public static final int IMPORT = 0;
        public static final int EXPORT = 1;
        public static final int HIDE = 2;
        public static final int SEAL = 3;
        public static final int UNSEAL = 4;
        public static final int FORBID = 5;
        public static final int UNFORBID = 6;

        public Dependency(int type, String target, String componentID);
}

public class MJ.Modules {
        public static void createDynamicModule(
                String id,
                String contract,
                String[] classPath,
                Dependency[] control,
                Dependency[] dynamicControl,
                String[] classList);
}
```

Dynamic module creation is done through `MJ.Modules.createDynamicModule()`, which takes the following arguments:

id the component ID for the new module (described in section 4.2)

contract a human-readable alias for the component ID

classPath an array of .jar files and directories in which this module will look for class files

control the module's static control statements, which are processed exactly as they are when using a textual module definition

dynamicControl the module's dynamic control statements, restricted to `export` and `hide`

classList a list of package-prefixes and class names provided by this module

Figure 6: `MJ.Modules` dynamic module interface

scription file `catalina.jm`, into the repository in the directory "R", we would use the following command:

```
% modjavac -repository R catalina.jm
```

In the same way that, at runtime, we use the component registry to control which modules can import which classes from which other modules, we can do the same thing at compile-time. When compiling a class `FooServlet.java`, we know which module the class is part of, `MyWebapp`, and thus it would be desirable to enforce the static constraints of the module at compile-time. Given that our module repository lives in the directory "R", we can compile the file using

```
% javamodc -d classes -repository R \
  -module MyWebapp FooServlet.java
```

`javamodc` is a wrapper around `javac` that both checks static module constraints and sets up the appropriate classpath for compilation.

3.5 Running MJ Applications

To run a top-level program, one provides the path to the component-registry and the component-id or module name in which will be found the "main" program. The "loader" will initialize the component system, resolve the named module, and invoke the module's static main method.

```
% java loader -repository components \
  catalina start
```

3.6 Extensions

A capability we considered, but have not yet implemented, is "pass-thru" modules – modules which import classes from provider modules, and re-export them, to other client modules. A simple application of this capability would be "aggregation" – collecting the exports of a number of provider modules into a single aggregate module, which clients would import; thus, the provider modules could be restructured, without modifying the clients at all.

Pass-thru modules could also be used in the other direction to virtualize the targets of `export`, `hide`, `seal`, `unseal`, `forbid`, and `unforbid` directives. This would allow a module to specify the interface it exposes to different types of clients without having to know in advance the actual names of the target client modules. For example, `java.lang.StringInternals` could be exported to a single logical `trusted` module and hidden from all other modules. Application and site-specific policies could then be applied to define the `trusted` module and thus control which other modules were allowed to gain access to `StringInternals`.

4. IMPLEMENTATION

This section describes how MJ is implemented in terms of Java's existing classloader mechanisms. Before explaining how we use classloaders in our implementation, we provide some background information on Java classloaders.

4.1 A Short Tutorial on Java Classloaders

Java classloaders are the mechanism by which `Class` objects are created in the Java virtual machine, inter-class references are resolved, and by which quite a few of the security and associated naming constraints are enforced. The semantics of classloaders are described in the JVM specification [19] and in more depth by Liang and Bracha [18].

Informally, every class in a JVM has an associated classloader that defined it. When the JVM executes a bytecode of some method `m` that requires the resolution of an inter-class reference, the process begins in the defining classloader of `m`'s class. The classloader first checks to see if it has already loaded the desired class. If it has not, it then asks its parent classloader to find the class, and if its parent cannot, then looks for the class itself. If it cannot find the class, than a `ClassNotFoundException` is raised. Classloaders may search for a class file in the local file system, on some remote file system, or may even generate it dynamically via any arbitrary process.

There are some constraints on the behavior of classloaders to ensure type safety. The exact details can be found in [18]. Informally, some of the key rules are

- Because all classloaders must delegate to their parent classloader before defining a class themselves, a child classloaders should never implement a class that is already implemented by their parent. For example, no *child* classloader should implement `java.lang.String`, since it is defined by the system classloader.
- Given the same name, a classloader must always return the same `Class` object.
- If a classloader `L1` delegates the loading of a class `C` to another loader `L2`, then for any type `T` that occurs as the direct superclass or a direct super-interface of `C`, or as the type of a field in `C`, or as the type of a formal parameter of a method or constructor in `C`, or as a return type of a method in `C`, `L1` and `L2` should return the same `Class` object.

Thus, each classloader represents a namespace, and the delegation behavior of child classloaders to their parents is a form of importation from parent to child.

4.2 Module-Aware Classloaders

Based on the observation that classloaders represent namespaces and classloader delegation can be viewed as importation, there is a fairly natural implementation of MJ in terms of the existing Java classloader mechanisms. Absolutely no changes are required to the JVM itself to implement MJ.

As discussed in Section 3.2, each MJ module is associated with exactly one classloader instance. The module (and thus classloader instance) is named by a unique component ID, a GUID in our impelementation, and also has a human-readable module name. The module's classloader instance loads and defines all classes provided by the module. Intra-module class references can be directly resolved by the module's own classloader. The classloader for `component0` (the core Java libraries) is the system classloader. Each classloader delegates the resolution of all inter-module class references to a *component classloader* that uses the information in the component-registry (provided on JVM invocation) to resolve them.

To resolve an inter-module reference, the component classloader performs the following steps:

- It queries the component registry to get the import lists for the module that initiated the class resolution request.
- Each import list entry is checked in turn to see if the module in question exports a class whose name matches the class reference being resolved.
- If a match is found, then the `export` and `hide` declarations of the exporting module are checked to ensure that the initiating module is allowed to import the class from the exporting module.

- If this check succeeds, then the providing module is loaded and initialized (if it has not already been initialized) and then the desired class is loaded and prepared.
- Finally, it checks the superclass and all implemented interfaces of the newly loaded class against their providing module's `seal` and `unseal` directives to ensure that none of the subclassing control statements have been violated.
- If this final check succeeds, then the corresponding `Class` object is returned to the initiating classloader and the intermodule class reference has been resolved.

By forcing all inter-module class references to be resolved using the above steps, the system ensures that the visibility, import/export, and seal/unseal directives specified by the module system are enforced at runtime. However, this does require that the programmer must forgo most other uses of custom classloaders. This, combined with a few judicious `forbid` and `seal` directives in `component0` prevent malicious programs from avoiding the MJ inter-module classloading rules by creating classloaders that do not follow this protocol.

5. MODULARIZING APACHE TOMCAT

To experimentally validate the ability of our component system to properly handle the module-interrelationships found in large Java server systems, we replaced the classloader mechanisms of Apache Tomcat 4.1.18 with the module system herein described, removed all system classpaths (excepting the boot classpath of the JVM), and were able to produce a functionally identical Tomcat implementation.

To modularize Tomcat, we not only had to divide Tomcat into modules, but also modularize each of the supporting libraries Tomcat depends on. We chose to separate the application into one module for each jar file in the original Tomcat distribution and one module for each supporting library. Dividing the code in this manner seemed natural, as it allowed us to take advantage of the way the original authors grouped related classes into logical units. This gave us a final set of about 30 modules, which are described in Figure 7.

To determine the proper import/export relationships between the modules, we first created each module with a permissive set of import and export statements, allowing each module to reference any class in any other module. Next, we used the class-reference logging feature of our implementation to generate a graph of actual class usage at runtime. Given this information, we could observe which modules depended on which other modules, and thus generate the appropriate import and export statements for each module. The resulting module graph is shown in Figure 8.

We then removed all uses of classloaders from the Tomcat code and supporting libraries, replacing their use either with appropriate calls to the module system, or, in some cases, with nothing at all. In total, we changed approximately 400 of the 167,000 lines of Java code in the Tomcat sources. We found that classloaders were being used for several different purposes, each requiring a different strategy for removal.

- Thread context classloaders:

  Frequently, the Java convention of child classloaders delegating class requests to their parent classloaders breaks down. The typical case is when code loaded at a leaf node in the classloader tree, such as a servlet, calls a library loaded by a higher level classloader, such as JNDI, and this library needs to load a new class provided by the leaf classloader.

Java's solution to this problem is to associate a classloader with the currently executing thread. Thus the above example would proceed as follows:

1. The servlet will get and store the current thread context classloader:

```
Classloader oldCCL =
    Thread.currentThread().
        getContextClassloader();
```

2. The servlet will then set the thread context classloader to its own classloader:

```
Thread.currentThread().
    setContextClassloader(
        this.class.getClassLoader());
```

3. With the thread's classloader set, the servlet will call the desired function in JNDI
4. JNDI will use the current thread context classloader to load classes:

```
Classloader ccl =
    Thread.currentThread().
        getContextClassloader();
Class c = ccl.loadClass(className);
```

5. Upon return from the call to JNDI, the servlet will restore the saved classloader:

```
Thread.currentThread().
    setContextClassloader(oldCCL);
```

This method works in practice, but tends to be error-prone. First, developers have to know when to set the thread's context classloader before making a call to certain libraries, which is often non-intuitive. Second, more subtly, failing to restore the original thread context classloader can lead to obscure bugs, a situation that is surprisingly easy to create, given that an uncaught exception can bypass the code restoring the classloader. In fact, we found several examples of code in Tomcat that did not properly handle exceptions in these spots.

Let's review the underlying purpose for this usage of thread context classloaders. Each classloader is a namespace, and classloaders "lower down" in the tree (like a user application) inherit and extend their parent classloaders' namespaces (like the JNDI module). The use of thread context classloaders is thus an attempt to allow code running in a higher namespace (JNDI) to create objects whose classes are in lower namespaces (a user application) – in short, to hand that code a reference to the namespace in which the desired class will be found, as well as, usually, a desired classname.

Using MJ, since modules have names, we simply specify both the module-name and the class-name (of the application-specific naming factory), using a module-qualified class name (MQCN), and arrange for the higher-up code (in JNDI) to have import access to that classname, from that module. The dependency can be static, or dynamic as necessary – but in any case it is explicit via the use of this MQCN.

Since JNDI is a module which was compiled and delivered with the JDK, its static imports are fixed. JNDI needs to be able to *dynamically* import the specified MQCN from its provider module. There are two steps for this:

component0	Basic Java system classes and JDK
catalina	Core servlet engine
bootstrap	Startup and initialization code for catalina
servlet	Servlet APIs
xerces	Xerces XML parser 2.1.0
tools	JDK tools, including `com.sun.tools.javac.Main`
jasper_compiler	JSP compiler
jasper_runtime	JSP runtime support
commons_logging	Apache Commons Logging library 1.0.2
commons_digester	Apache Commons Digester 1.3
commons_beanutils	Jakarta's Java Beans utility functions
commons_collections	Data structures used by the commons_* modules
tomcat_coyote	Interfaces between Tomcat and protocol handlers
tomcat_http11	Tomcat HTTP/1.1 protocol handler
tomcat_jk2	Tomcat JK2 protocol handler
tomcat_util	Utility functions for the tomcat_* modules
servlets_default	Basic servlet services
servlets_invoker	Servlet invoking functionality
servlets_manager	Servlet management
servlets_webdav	Servlet WebDAV support
servlets_common	Common servlet functionality
naming_common	Apache JNDI implementation
naming_resources	JNDI Directory Context implementation
naming_factory	JNDI object factories
ant	Java build tool
catalina_ant	Tomcat-specific additions to Ant
jndi	Java Naming and Directory Interface
warp	Tomcat WARP connector
log4j	Java logging library

Figure 7: Apache Tomcat Modules

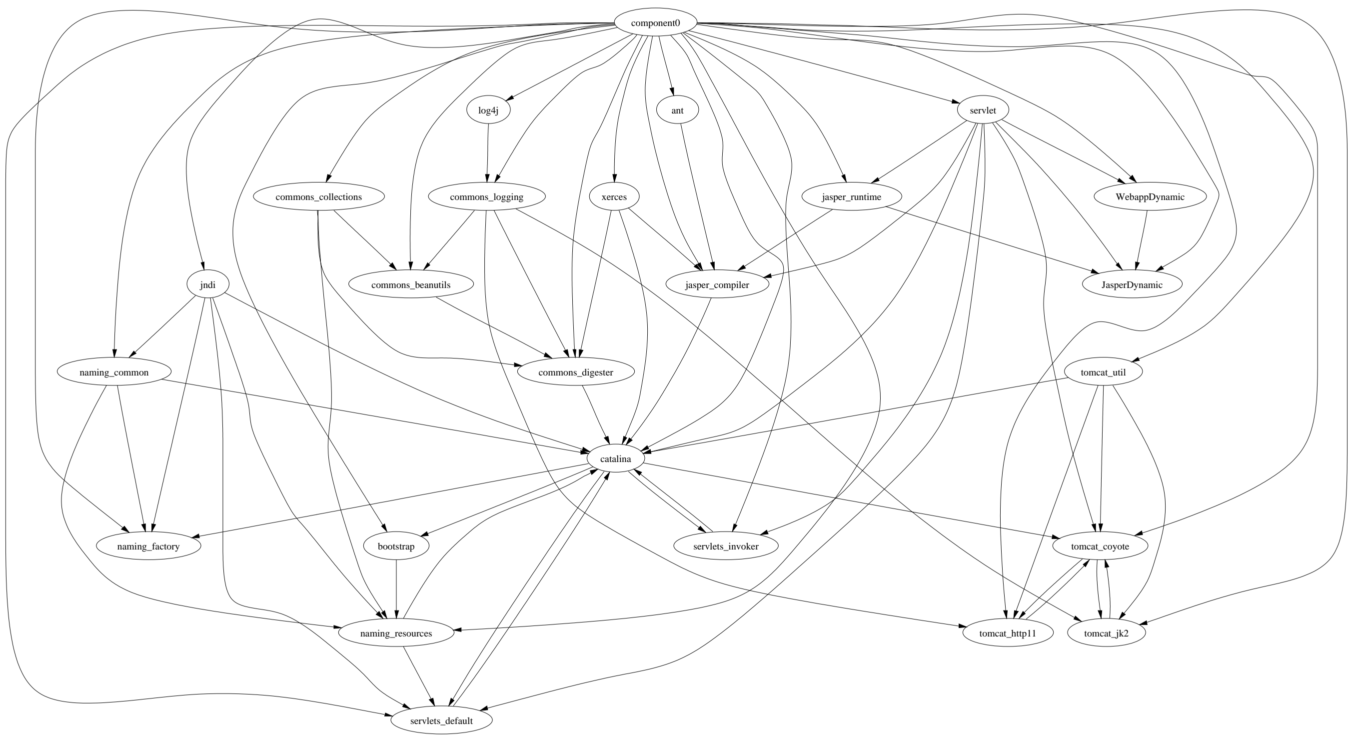

Figure 8: Module relationships in Apache Tomcat

- the provider module must export the class using the `dynamic export` statement
- JNDI must resolve the MQCN using `Component.forName`

- Ad-hoc modularity:

  There were several spots in Tomcat where delegation to particular classloaders based on a class' prefix was used as an ad-hoc form of modularity. Consider the following example:

```
if (name.beginsWith("org.apache.jsp"))
    class = jspLoader.loadClass(name);
else
    class = parentLoader.loadClass(name);
```

  These cases can be replaced seamlessly with `Class.forName` when using the module system.

- Dynamic loading of servlets and JSPs:

  There are instances in Tomcat where module information cannot be made available statically. One important case is JSPs; java source files for JSPs are generated and compiled on the fly, and new JSPs can be inserted into a running instance of Tomcat.

  We must emphasize that this is one of a very small number of examples where dynamic loading is actually necessary. Regardless, we can solve this problem by dynamically creating a module providing the JSP classes via dynamic exports. Servlets work similarly to JSPs, with the exception that a servlet's information is known at *deploy*-time, so the use of dynamic modules is not strictly necessary.

- Unnecessary references to classloaders:

  A much simpler, but common, use of classloaders is when code obtains a pointer to its own classloader, then uses this classloader to resolve a class, as in the following example:

```
this.getClass().getClassloader().
   loadClass(className);
```

  As long as the code does not make any other use of this classloader, such as handing the pointer to some other class, we can replace this code with the more straightforward `Class.forName`.

Finally, we ensured that no application code made use of or reference classloaders in any way by adding the statement

```
hide java.lang.ClassLoader in *;
```

to `component0`'s description file.

Since we could now control the visibility of particular classes, we implemented the `StringInternals` class described earlier (in Figure 1) and modified `OutputStreamWriter` to take advantage of it, improving the speed of an example output-intensive servlet by 30% with only a few lines of code change in internal libraries.

Our microbenchmark consisted of a servlet which generated a multiplication table formatted as HTML. For the case of a 100x100 table, we generated requests of approximately 180 KBytes of HTML. On an IBM Thinkpad T23 Pentium III 1.2GHz machine on Debian Linux (2.4.16 kernel) with 512M of memory, running the IBM JDK 1.3.1 and Tomcat 4.1.18, we observed that an unmodified JDK would deliver 29.83 requests per second, while with just the single and simple OutputStreamWriter modification, we achieved 43.3 requests per second – a speedup of approximately 30%.

While this test is not representative of all Java benchmarks, anecdotal evidence suggests that String manipulation is a significant part of the performance problem in Java server implementations. The OutputStreamWriter optimization was one of about three which we applied to an actual application that achieved a 2.5x speedup. But in that case we lacked MJ, and therefore, the resulting code had to break the rules of the JDK.

More generally, we expect that our technique, applied to other spots in the JDK and J2EE API-set, should yield nontrivial speedups, while abiding by the rules of the JDK and J2EE.

- For instance, our technique makes it possible to give compiled CORBA stubs access to the internals of IIOP message-buffers, without also giving such access to application code.

- Likewise, container-managed persistence beans can be (automatically) generated to transfer data directly in and out of buffers managed by a (pure Java) JDBC type 3 or 4 database driver [17], again without allowing application code to access that buffer.

- `java.io.PrintWriter` objects used in servlet-response output can be "rewound", using private interfaces to `PrintWriter` which are not available to other applications, allowing them to re-use their internal buffers.

In all these cases, today, absent complex classloader tricks, one would have to provide function-call-only interfaces, in order to protect the integrity of the underlying datastructures.

Given our experience with controlled exportation of "unsafe" classes to trusted modules, we expect that we can equally easily modify the internals of EJB servers and servlet engines to take advantage of the performance enhancements available thru our abstract data-type mechanisms.

6. RELATED WORK

MJ is based on many well known ideas from previous module and component systems. Our contribution is not novel language design, but the synthesis of well known ideas from a number of previous systems into a module system that enables the solution of a number of problems that plague large scale industrial Java programs.

6.1 Component and Module Systems

The most closely related work to MJ is in component systems such as the Microsoft COM object-model [22]. As in COM, components are managed by a component registry and the main task of the module system is to enable components to be dynamically linked together with the proper version of other components on which they depend. The Units module system [11, 9] for Scheme also addresses many of the same underlying concerns as MJ.

Languages such as Ada95 [2], Modula-2 [30], Modula-3 [13], Haskell [14], and Java [12] all include some notion of `packages` to control static scoping of names. ML's functors [24, 20] enable module composition and abstraction, but are purely static and prohibit certain kinds of cross-module relationships. The goal of MJ is to address issues, such as those described in Section 2, that pure package-based systems fail to handle.

6.2 Java Extensions

There have been a number of proposed extension to Java that add module or component systems. The primary difference between MJ and this prior work is that the goal of MJ is to make the

smallest possible changes to existing Java code to obtain the desired modularity. Our goal is to be able to practically apply MJ to large scale (multi-million line) industrial Java programs.

Jiazzi [21] applies the ideas of Units to Java, and also adds support for mix-ins and open classes. ComponentJ [26] is focused on black-box components. It allows components to contain Java classes, but forbids cross-component inheritance relationships as the Java classes contained by a component are not exposed to other components. Bauer et al. proposed a Java module system motivated by security concerns that at the high level is very similar to MJ [3]. However, their implementation changes Java's dynamic loading semantics and makes reflection APIs such as `forName` and `getName` unusable.

There has also been work on augmenting Java with a ML-style module system that supports much stronger separation between implementation and interface than MJ does. JavaMod [1] constructs a true ML-style module system on top of a Java-like core language. Multi-Java [4] extends Java with multi-methods and a module system based on the Dubious core language [23].

Finally, as described in Section 1, it is common practice to control the visibility of particular classes by using specialized classloaders that delegate to each other in various patterns. In the early days of Java, the only supported pattern was an inheritance tree, but with Java 2, classloaders may delegate in much more complex ways. Iona's firewall classloaders [10], WebSphere [29] 5's graph of classloaders, and Eclipse [8]'s graph of classloaders, are three examples of this practice, though we expect there to be many other systems which use graphs of classloaders in this way.

7. FUTURE WORK

As described in Section 5, we were able to achieve a 30% speedup in a servlet micro-benchmark by exploiting the new `StringInternals` class in `OutputStreamWriter`. There are obviously other classes in the core Java libraries that would benefit from similar changes. We plan to explore other such opportunities.

Although Tomcat is a realistic test case to demonstrate that MJ is practical and effective, an even more convincing demonstration would be to use MJ in a commercial application server such as WebSphere [29]. Based on our Tomcat experiences, we believe this is feasible and should result in significant performance improvements. However, WebSphere is a significantly larger piece of software than Tomcat and doing so will be a non-trivial effort and may hold unanticipated challenges.

Implementing pass-thru modules (Section 3.6) will be necessary for making MJ usable in practice.

We also have not yet implemented versioning of modules, and version-level compatibility constraints. We believe, based on preliminary investigation, that we can do this by following more or less faithfully in the footsteps of the many packaging/versioning systems for UNIX and Linux, e.g., Debian's DPKG[6]. One delicate issue in versioning, is that, unlike UNIX packaging systems, we can keep multiple versions of a module around at one time, and choose, based on the version compatibility constraints, which version we would like to activate.

Finally, one might imagine other profitable uses of the information rendered explicit in the component registry. For example, a classic means of introducing separate compilation into a system which does not support it is to control the importation/exportation relationships. Then, one can verify that the compiled module is only usable in execution environments where the importation and exportation environments are either compatible with, or identical to, that of compilation. Thus MJ may provide a natural approach to separate native compilation for Java programs, because it allows the system know precisely which classes, from which other modules, can be referenced by the classes being compiled.

8. CONCLUSION

This paper presents MJ, a module system for Java. MJ supports the encapsulation and modularity semantics required for large scale component-oriented programming but yet can still easily be retrofitted into existing Java programs. Code that does not explicitly manipulate classloaders can be used unmodified; code that does can be modified to work with MJ in straightforward ways. MJ's modularity mechanisms can be statically checked and dynamically enforced. Furthermore the compile time environment is explicitly communicated to the runtime via a component registry, avoiding many of the thorny issues with current classloader/classpath based approaches to component-oriented programming in Java.

We have implemented MJ and experimentally validated the ability of MJ to properly handle the complex module inter-relationships found in large Java server systems, by replacing the classloader mechanisms of Apache Tomcat 4.1.18 with 30 MJ modules. The modified Tomcat is functionally identical to the original, but requires no `CLASSPATH` definitions, and will operate correctly even if user code loads a different version of a module used by Tomcat, such as the Xerces XML parser. The changes required to Tomcat were extremely small (approximately 400 lines in a code base of 167,000 lines were changed). Furthermore, by making a small change to the Java core libraries enabled by MJ, we obtained a 30% performance improvement in a servlet microbenchmark.

Acknowledgments

We would like to acknowledge Glyn Normington, for both suggesting the use of pass-thru modules for aggregation, as well as for providing significant feedback on our entire design. We also thank Daniel Yellin and the anonymous reviewers for their detailed comments and constructive feedback.

9. REFERENCES

[1] ANCONA, D., AND ZUCCA, E. True modules for Java-like languages. In *15th European Conference on Object-Oriented Programming* (2001).

[2] BARNES, J. *Programming in Ada 95*. Addison-Wesley, 1996.

[3] BAUER, L., APPEL, A. W., AND FELTEN, E. W. Mechanisms for secure modular programming in Java. Tech. Rep. Tech Report TR-603-99, Princeton University, Department of Computer Science, July 1999.

[4] CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. MultiJava: modular open classes and symmetric multiple dispatch for Java. *ACM SIGPLAN Notices 35*, 10 (Oct. 2000), 130–145. Published as part of the Proceedings of OOPSLA'00.

[5] Corba. http://www.omg.org/gettingstarted/corbafaq.htm.

[6] Debian. http://www.debian.org/.

[7] DETLEFS, D., AND AGESEN, O. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming* (June 1999), pp. 258–278.

[8] Eclipse. http://www.eclipse.org.

[9] FINDLER, R. B., AND FLATT, M. Modular object-oriented programming with units and mixins. *ACM SIGPLAN Notices 34*, 1 (Jan. 1999), 94–104.

[10] Advanced classloading in J2EE. http://www.theserverside.com/resources/articles/AdvancedClassLoading/article.html.

[11] FLATT, M., AND FELLEISEN, M. Units: Cool modules for HOT languages. *ACM SIGPLAN Notices 33*, 5 (May 1998), 236–248. Published as part of the Proceedings of PLDI'98.

[12] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. Addison Wesley, 1996.

[13] HARRISON, S. *Modula-3*. Prentice Hall, 1991.

[14] HUDAK, P., AND WADLER, P. Report on the programming language Haskell. Tech. Rep. Yale/DCS/RR777, Yale University, Department of Computer Science, Aug. 1991.

[15] IONA Orbix. http://www.iona.com.

[16] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a Java^{TM} Just-In-Time compiler. *ACM SIGPLAN Notices 35*, 10 (Oct. 2000), 294–310. Published as part of the Proceedings of OOPSLA'00.

[17] Jdbc drivers: How do you know what you need? http://archive.devx.com/dbzone/articles/dd_jdbc/sosinsky-2.asp.

[18] LIANG, S., AND BRACHA, G. Dynamic class loading in the Java Virtual Machine. *ACM SIGPLAN Notices 33*, 10 (Oct. 1998), 36–44.

[19] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification Second Edition*. The Java Series. Addison-Wesley, 1999.

[20] MACQUEEN, D. Modules for standard ml. In *Proceedings of ACM Conference on Lisp and Functional Programming* (1984), pp. 409–423.

[21] MCDIRMID, S., FLATT, M., AND HSIEH, W. C. Jiazzi: new-age components for old-fashioned Java. *ACM SIGPLAN Notices 36*, 11 (Nov. 2001), 211–222. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[22] Microsoft COM: Component object model. http://www.microsoft.com/com.

[23] MILLSTEIN, T., AND CHAMBERS, C. Modular statically typed multimethods. *Information and Computation 175*, 1 (May 2002), 76–118.

[24] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. MIT Press, 1990.

[25] ROBINSON, R. Developing and deploying modular J2EE applications with WebSphere Studio Application Developer and WebSphere Application Server. http://www7b.software.ibm.com/wsdd/library/techarticles/-0206_robinson/robinson.html.

[26] SECO, J. C., AND CAIRES, L. A basic model of typed components. In *14th European Conference on Object-Oriented Programming* (2000).

[27] Tomcat. http://jakarta.apache.org/tomcat/.

[28] Visigenic. http://www.borland.com/corba/index.html.

[29] WebSphere application server. http://www.ibm.com/software/websphere.

[30] WIRTH, N. *Programming in Modula-2*. Springer-Verlag, 1983.

[31] Xerces. http://xml.apache.org/.