RZ 3644 (# 99654) 01/18/2006
Computer Science 16 pages

Research Report

Babel: A Version Management System for the Java Programming Language

Thomas Gschwind and Michael Moser

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

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 (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

Babel: A Version Management System for the Java Programming Language

Thomas Gschwind and Michael Moser

IBM Research
Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon, Switzerland

Abstract. For the deployment of large software systems powerful mechanisms for module and version management are essential. For both areas, the popular Java programming language, however, provides only limited support. Module management is supported by means of Java's package and jar file mechanisms which is mostly a packaging mechanism that just allows to amend packages in a jar file with some meta information via an included manifest file. The only version management functionality supported is to allow developers to query a package for its version number, after it has been loaded. Based on that, an application can try to make assumptions whether that version might be compatible or not. However, if the currently running Java application uses a module that depends on a different and incompatible version of a component that is already loaded, Java users are out of luck since Java does not allow to load two different versions of the same class. In this paper, we present the Babel Version Management System that tackles all of these issues.

1 Introduction

Today's software systems strive to be modular and extensible. One common approach to achieve modularity is the integration of third party software components through standardized interfaces. Examples of such software systems include the Eclipse Platform, the Apache Jakarta Tomcat Servlet Engine, or Enterprise Java Beans Servers such as JBoss or IBM's Websphere Application Server. Because these software systems integrate or compose software components that come from several different sources, they have to be able to cope with different versions of the same software component. For instance, while one third party component may depend on one version of the Xerces Java Parser, another one might depend on a newer but incompatible version of the same parser.

The authors of this paper encountered exactly this situation when switching to the 3.0 version of the Eclipse Integrated Development Platform. Eclipse versions 3.0 and higher require a Java Runtime Environments of version 1.4 or higher. This version of the Java runtime environment already includes an XML library. A plug-in that we depended on, however, required an older, incompatible

version of the Xerces XML parser. Since the two XML libraries could not co-exist in a single application we either had to stick to an older version of Eclipse or not to use the plug-in [2, 1]. Unfortunately, neither choice was an option.

The basic reason for this type of problem is that the Java programming language does not allow to have more than one version of an element with the same name. The standard mechanism to sidestep this dilemma is the use of Java class loaders. Class loaders are responsible to find and load classes and can allow individual modules and plug-ins to run within their own sandbox [11] and thereby to protect a software system from buggy or malicious code [7].

The drawback of this approach, however, is that it further complicates Java's class path mechanism since every component or plug-in to be used in the software system now has to provide its own class path setting and to many users it is unclear which jar file should be put into the class path of which module. Putting a jar file into the class path of the parent application has the advantage that modules can share the same jar file and hence it only has to be loaded into memory once, thus conserving memory. However, if different modules depend on different versions of a jar file, they cannot be put into the parent application's class path but instead need to be put into the class path of the individual modules. Worse, if some version has been put into the parent application's class loader there is typically no way to identify this issue, instead the application fails unexpectedly during run-time.

In this paper, we present the Babel Versioning Management System that solves the above problems. Section 2 presents some background information of what kind of versioning information is currently provided by the Java programming language. In Section 3, we give an overview of the Babel Version Management System, which is our solution to that problem. In Section 4, we explain the implementation and in Section 5 the evaluation results of Babel. We compare our work to related work in Section 6 and draw our conclusions in Section 7.

2 Background

The Java programming language provides some support to ensure backwards compatibility by allowing developers to deprecate methods allowing software developers to gradually upgrade their components [8, 12]. This approach, however, only works as long as user's of third party component's manage to get hold of a newer version of the component before the deprecated method has been finally removed.

Deprecating methods, however, only warns developers of methods that are going to be removed. Removing methods is not the only modification that can cause software to break. If a method is added to an interface, every software component that implements that interface needs to be changed as well. Again, this is only possible of the software component's source code is available.

The developers of the Eclipse Software Development Kit [6] addressed this issue by ensuring that interfaces are never modified. If an interface needs to be extended, a new interface either extending or replacing the original interface

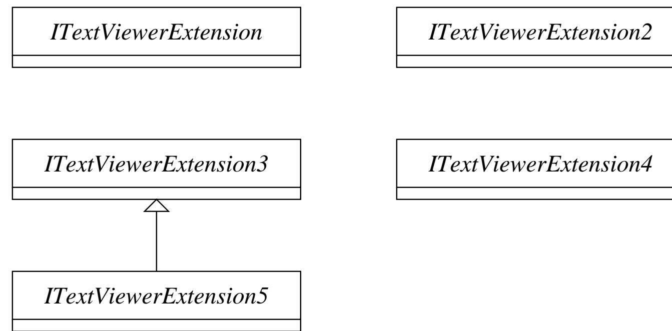

Fig. 1. The `org.eclipse.jface.text.ITextViewerExtension` Interfaces.

```
class SomeDocumentProviderExtension implements
    ITextViewerExtension, ITextViewerExtension2,
    ITextViewerExtension4, ITextViewerExtension5 {
    ...
}

class SomeDocumentProviderUser {
    ...
    ITextViewerExtension viewer = someObject.getViewer();
    ((ITextViewerExtension2)viewer).doThis();
    ((ITextViewerExtension5)viewer).doThat();
    ...
}
```

Fig. 2. Using the `ITextViewerExtension` Interfaces.

is used. In order to achieve unique names, newer interfaces are suffixed with numbers (Figure 1). Sometimes a higher numbered version completely replaces the older version or covers an additional role or perspective of that class. In the latter case newer interfaces extend older interfaces.

An example for this can be found in the declaration of extension points that had to provide more and more functionality with each new version (Figure 1). The implicit assumption is that the latest implementation implements *all* interfaces such that the programmer can be sure that it is possible to cast a `Name`-object to a `Name`n-object as shown in Figure 2. Newer code then can refer to higher numbered interfaces while older code continues to refer to a lower numbered or unnumbered original version of the interface.

As can be seen, if this step is repeated a couple of times it becomes cumbersome, not only for the implementor but also for the user of these interfaces. This approach requires lots of distinctions, type checks using `instanceof`, and type casts in the code, and hence reduces the code's readability, clarity, the programmer's oversight, and thus is prone to errors. Another downside is that methods

can never be removed nor their signatures changed, i.e. even if the internals of some classes change, all historic methods have to be maintained. Additionally, this approach only works if all the external libraries that the code depends on follows the same approach.

Java archives (jar files) provide some simplistic version information on a per package level. That is, the manifest file of the jar file may specify a title, version and vendor of any package provided by the jar file. Information about a given package can be requested using `Package.getPackage(packagename)`. This approach, however, has several shortcomings:

– It allows developers only during run-time to query whether a given package is available in a specific version. Hence, we cannot statically determine whether the system is configured correctly.
– The `isCompatibleWith("version")` method only returns whether the provided version number is larger assuming backwards compatibility which is not always the case. Unfortunately, backwards compatibility is only guaranteed if the interfaces provided by a component never change.
– Java's version management assumes that there is a one-to-one mapping between package names and modules which is not always the case. If two jar files contain classes pertaining to the same package, the version information provided by the jar file loaded first overrides the version information for the same packages in the other jar file.

These shortcomings show that the version information provided by the manifest file does not help to resolve the name clash if two different versions of the same module need to be loaded. It allows the application to identify a version conflict but it does not provide a solution to resolve it.

Before we describe, how different class loaders can solve the versioning problem, let us have a look at a large software system that has been extended using plug-ins. Figure 3 shows an application that uses the software component `xyz`. Additionally, the application has been extended with `PlugIn1` and `PlugIn2`. These plug-ins use the components `bar` and `foo`. `PlugIn2` itself has been extended with `PlugIn3` (using the component `bar`) and `PlugIn4`. If all of the plug-ins use the same versions of the respective components, there is no problem at all. However, if `PlugIn1` and `PlugIn2` use different versions of software component `foo`, the application will not execute correctly.

Class loaders provide a separation of namespaces by allowing multiple packages or classes with the same name to exist within different class loaders, hence preventing name collisions and avoiding that elements with the same name but different versions can refer to each other.

The aim of class loaders is usually to enforce a tree-shaped visibility structure [7] like the example shown in Figure 3, where elements within `PlugIn1` and `PlugIn2` cannot see (or refer to) any element that is part of the other but both see the `BaseApplication`. This allows both plug-ins to contain a different version of element `foo` without interfering with each other. Likewise `PlugIn3`, a child plug-in of `PlugIn2`, can contain an element `bar` which does not interfere

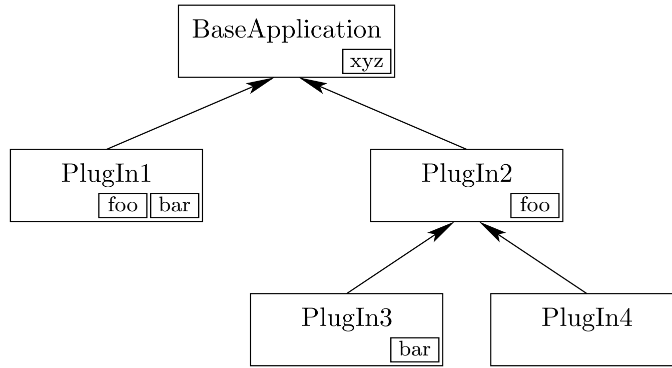

Fig. 3. Classes with Visibility

Base App.	`apps/base:apps/sharedlib:libs/xyz`
PlugIn 1	`apps/bin/plugin1:libs/bar-1.0:libs/foo`
PlugIn 2	`apps/bin/plugin2:libs/foo`
PlugIn 3	`apps/bin/plugin3:libs/bar-2.0`
PlugIn 4	`apps/bin/plugin4`

Table 1. Sample Class Path Setting

with the element `bar` inside `PlugIn1`. And all elements can, for instance, refer to the element `xyz` in the `BaseApplication` but there cannot be another `xyz` in any other plug-in.

Strongly associated with the above namespaces is the Java classpath issue. Since different element versions must be loaded from different files one must make sure, that the code belonging to each subtree is searched for and loaded from a specific, corresponding location in the system's file space.

In the above example the search path for files of the different class loaders is shown in Table 1. As can be seen from this simple example this approach very much complicates Java's class path mechanism since every module or plug-in to be used in the system has to have its own well-defined class path setting and to many users it is unclear which jar file should be put into the class path of which module and in which order.

Putting a jar file into the class path of the base (or higher up) has the advantage that plug-ins can share the same file and hence conserve memory by loading it only once into memory. This is possible since a class loader before loading a class should check whether its parent class loader already provides the class [18]. This approach, however, has several disadvantages:

– If different modules depend on different versions of the jar file, these cannot be put into the parent application's class path but rather need to be put into the class path of the individual modules.

– If an identically named element is already present in the parent parent application's class loader (e.g., if the `BaseApplication` had already loaded a version of `bar`), then other libraries (e.g., `PlugIn1` and `PlugIn3`) would later not be able to load their required version of the library and thus one or even both of them may fail unexpectedly during runtime.
– If `PlugIn1` uses a class from the `BaseApplication` which triggers the loading of a new class then (for security reasons) this new class will be loaded with the class loader of the base application. This can be cumbersome if the newly loaded class needs to cooperate with any of the software components loaded using `PlugIn1`'s class loader.
– Additionally, using different class loaders works fine if the plug-ins implement interfaces defined by the base application as is typically the case for plug-ins. However, if the interfaces were defined as part of `PlugIn1` and `PlugIn2` (which would be the case if they were nested software components), their interfaces would be loaded with the base application's class loader, unless we split up their jar files manually.

Unfortunately, there is no practical way to identify the above issues in advance (e.g. by doing statical or load time checking) and generically solving the naming conflicts. In fact this entire topic is complicated enough that entire papers such as [17] have been written about it.

In [4], Corwin et al. present an approach that tries to address these issues. Their approach is based on using a separate class loader for each module to be loaded. If a class requests another class to be loaded, the class loader checks which class is trying to load the class and whether it is responsible to load the class. If not, it locates the appropriate class loader which is achieved by a table of the class loaders and the modules they are responsible for.

Their approach not only checks the parent class loader as recommended by Sun Microsystems but also other class loaders. Hence, this approach requires the class loaders to be aware of each other so that a class can be loaded by the class loader defined for the module the class belongs to and not by the class loader of the class accessing this class for the first time. As a consequence, this implementation cannot deal well with the not so rare case that a plug-in comes with its own class loader. This class loader would not have any knowledge about the other class loaders and hence would interfere with their operation.

This and all previous scenarios above are handled nicely by our Babel Class Loading System.

3 The Babel Version Management System

The Babel version management system resolves version conflicts by transparently renaming the involved components instead of using different class loaders for loading the different components. In order to identify such version conflicts, our Babel system makes use of meta-data that describe the components, their version numbers, and their dependencies. This version information can be provided as

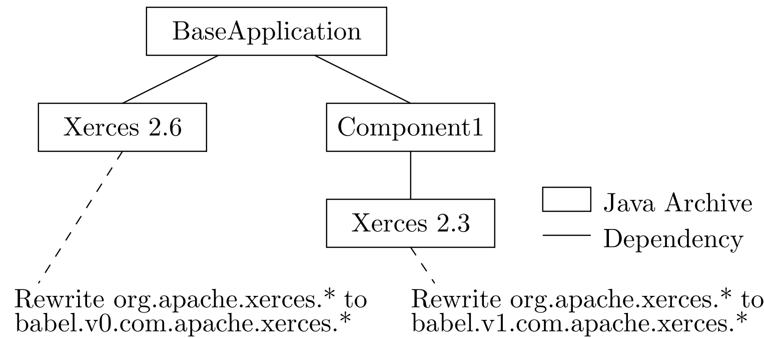

Fig. 4. An Application Using Different Versions of the Same Library

part of a separate configuration file as used in [4] or derived automatically from a jar file's manifest file [19]. Both approaches have their respective advantages.

Based on the version information, Babel can identify whether two different components (plug-ins, or modules) depend on different versions of the same component. Whenever this is the case, Babel renames the conflicting components and rewrites all the components making use of the conflicting component. This way, it does not matter whether a plug-in depends on a different version of a component than its base application.

To illustrate our approach, Figure 4 shows a simple application and its dependencies. The base application depends, say, on version 2.6 of the Xerces XML parser and on a `Component1` which depends, say, on Xerces version 2.3 and due to some incompatibilities cannot use Xerces 2.6. Since version 2.3 and 2.6 of the cannot both be loaded by the Java's main class loader. In this case, Babel would rewrite the package names of the different implementations so that they no longer collide. This is also shown in the figure.

Once Babel has analyzed the dependency description, it starts loading the classes. In our example, it first loads the base application's classes and rewrites them in such a way that all package references to the library in question are renamed from `org.apache.xerces` to `babel.v0.org.apache.xerces`. Similar, all classes of version 2.3 of the Xerces library are renamed the same way as soon as they need to be loaded by the base application. In order to avoid name clashes, all references in `Component1` to Xerces (version 2.6) classes are also renamed. Here the package name is changed from `org.apache.xerces` to `babel.v1.org.apache.xerces` and all the classes of version 2.6 of the Xerces library are renamed the same way when they are loaded.

Components can be either rewritten offline or on demand while they are being loaded. In the former case, all of the components used by the system need to be known a-priori. Otherwise our version management system would not be able to identify the components whose names are to be rewritten. Hybrid approaches are also conceivable as well as other tricks or optimizations.

If the components are to be rewritten on-demand, the application has to be loaded using Babel. Babel is necessary to intercept Java's class loading mechanism in order to rewrite the class files. How this is being accomplished will be presented in the next section. If a conflict between classes of different components is detected (e.g., due to the need to use a different but incompatible version of a component), our class loader assigns a new and unique name to the classes in question and adapts the references in all classes referring to the renamed classes. This even works if there is already a version of that library with the same name loaded and active in the current system.

Given a flexible specification (e.g. some form of mapping language) one can rename arbitrary elements at different levels of granularity: entire packages, individual classes or subsets of classes, even individual methods or fields only, or any mixture thereof.

4 The Implementation of Babel

For the purpose of Babel, we liberally define a component as a jar file. Babel uses the jar file's manifest file in order to identify the component's version number and subsequently operates on the bye code of the jar file's classes. An alternative implementation might have been to operate on the source code but as we have said before we assume that this is not always available.

The classes are renamed in a way such that the different versions of a given class can coexist with each other. Instead of renaming the actual class names, Babel renames the package names. This way it can ensured that the rewritten name does not accidentally overlap with another name by reserving a namespace below, for example, the namespace `com.ibm.babel`.

Before we present the detailed implementation of our approach, we give a brief overview of the layout of Java class files. A more detailed presentation of the Java class file format can be found in [12].

4.1 Java's Class File Format

The layout of Java class files is depicted in Figure 5. A Java class file first starts with a magic number indicating that this is a Java class file, followed by the minor and major version number of the Java Virtual Machine for which the file is valid. After that the class file stores the constant pool, the access rights for the class defined and the indices of this and the super class within the constant pool. Finally, the class file contains the interfaces implemented by the class, the fields and methods it provides and a set of attributes pertaining to the class.

Constant Pool The constant pool contains all the constants that are used within a Java class file. This includes all the strings, class names, interface names, field names, method names. Hence, if we want to rename the name of a class or a method, we only need to identify the respective entry inside the constant pool and change it.

```
CLASSFILE := magic minor major CONSTANTS access this super
             INTERFACES FIELDS METHODS ATTRIBUTES
CONSTANTS := constcount (CONSTANT)*
CONSTANT  := tag_class name | tag_fieldref class name_type |
             tag_methodref class name_type |
             tag_ifmethodref class name_type |
             tag_string content | tag_int int |
             tag_float float | tag_long long |
             tag_double double | tag_name_type name type |
             tag_utf utf
INTERFACES:= ifcount (interface)*
FIELDS    := fldcount (access name desc ATTRIBUTES)*
METHODS   := methcount (access name desc ATTRIBUTES)*
ATTRIBUTES:= attrcount (name attrlen (byte)*)*
```

Fig. 5. Format of a Java Class File

Interface Section This section contains one index into the constant pool for each interface implemented by the class.

Field Section The field section contains all the fields of the class. Each field is described using an access flag, an index into the constant pool for the name of the field, another one for the description of the field (i.e., the type), and finally a set of attributes pertaining to the field. The attributes are stored in the same format as the class attributes (see below).

Method Section The method section looks like the field section except that it describes methods of the underlying class instead of fields.

Attribute Section The attribute section contains a number of attributes each having an index into the constant pool describing the name of the attribute, a length field and then length bytes describing the attributes. The format of the attribute depends on its name. Using this attribute section, Java class files can be extended with additional information not available in earlier versions of Java's Virtual Machine without confusing earlier versions of the JVM.

An excerpt of the constant pool of a simple Java class implementing the Sieve of Eratosthenes is shown in Figure 6. For instance, the `Sieve` class itself is stored at index 7, other classes referenced by the `Sieve` class are stored at indices 2, 7, 12, and 28. Constants are referenced by the byte-code of a class using their respective indices. The byte-code of a sample method is shown in Figure 7.

4.2 The Babel Class Loader

Our Babel Version management system uses the manifest file to identify if different versions of the same component are being required. The manifest file is typically packaged as part of a Java archive and stores among other things the name, version, and vendor of the specification implemented by the Java archive.

```
 1(methodref): class=#28, name_type=#58
 2(class): name=#59
 4(fieldref): class=#7, name_type=#60
 7(class): name=#64
 9(methodref): class=#7, name_type=#67
12(class): name=#71
28(class): name=#88
29(UTF): sieve
30(UTF): [B
31(UTF): <init>
32(UTF): (I)V
39(UTF): Lcom/ibm/sample/sieve/Sieve;
42(UTF): Lcom/ibm/sample/primes/Primes;
53(UTF): ()V
58(name_type): name=#31, type=#53
59(UTF): com/ibm/sample/primes/Primes
60(name_type): name=#29, type=#30
64(UTF): com/ibm/sample/sieve/Sieve
67(name_type): name=#31, type=#32
71(UTF): java/lang/StringBuffer
88(UTF): java/lang/Object
```

Fig. 6. Part of the Constant Pool of a Simple Java Class

```
public byte[] getSieve() {
  //  0  0:aload_0
  //  1  1:getfield #4  <Field byte[] sieve>
  //  2  4:areturn
}
```

Fig. 7. The Byte-Code of a Sample Method

The manifest file, however, does not provide a standardized way of defining the Java classes and versions that it depends on. In order to store this information, we use the user-defined field `Depends-On`. This field stores the name of the specification and version number that the Java archive depends on. If multiple entries need to be specified, they may be separated using a semi-colon. A sample such manifest file is shown in Figure 8.

Once the conflicting components are being identified, it rewrites the classes to use one package name for the classes of one version of a component and another, different, package name for another version of the same component. Thanks to Java's simple class file structure all the above can be implemented in a straight forward manner by simply changing the strings identifying the class names within the class files' constant pools.

```
Manifest-Version: 1.0
Specification-Title: Sieve
Specification-Version: 1.0
Specification-Vendor: IBM
Main-Class: com.ibm.sample.sieve.Sieve
Depends-On: Primes/1.0
```

Fig. 8. A Sample Babel Configuration File

Figure 8 shows a Java archive that relies on version 1.0 of the Primes component. If another part of our application relies on version 2.0 of the Primes component, Babel renames all the references to any `com.ibm.sample.primes.*` class. However, Babel also needs to keep track of these renamings because at one point in time, the Java Virtual Machine will try to load the renamed class and then, Babel needs to identify the component to which the class belongs to.

4.3 Reflection

One of the drawbacks of the babel class loader is that in its current version it does not deal well with all forms of reflection. That is, if a program uses reflection in order to create or access classes from a given library, there might be a problem. We believe, however, that these problems do not occur for a typical software product and can be fixed in most cases by also instrumenting the reflective code. A frequent use of reflection is for the emulation of a kind of method pointers since Java does not provide this language feature. Code that makes use of this typically looks as follows:

```
Class c=anObject.getClass();
c.getMethod(methodname).invoke(...);
```

Code like the above, fortunately, poses no problem with our Babel class loader because Babel does not change the name of methods.

A situation that can pose some problems, however, is if code that has been renamed tries to get hold of the class name of an object as illustrated in the following:

```
System.out.println(anObject.getClass().getName());
```

Such code frequently occurs in log statements. This line prints out the rewritten name instead of the original name. In many cases this might actually be desirable, since it allows developers that are looking at the log files to distinguish between the versions of the library used. Of course there might be some cases where this may lead to problems, for instance, if the name of the class is bound to some external entity or property and an exact match of the class names is required.

A similar problem occurs if developers use `Class.forName("pkg.Class1")` to instantiate a class. In this case, the Babel class loader, depending on the configuration, would either load another instance of the `Class1` of either version of

the library or would throw a `ClassNotFound` exception. The former case might in some cases be desirable. To solve this problem, we provide an option that allows the replacement of the class name in any kind of string which we assume works fine in most cases but cannot be guaranteed to always work.

A solution to the above problem is to rewrite all calls to the `getName` and `forName` methods and rewrite the renamed class name to the original class name (and vice-versa) as necessary. This way we can guarantee correct behavior as long as the programmer does not use reflection in order to invoke these methods in order to get hold of a method name. If necessary, however, this problem can be solved by wrapping the `Class` class and rewriting the code to call our `Class` wrapper instead of the original `Class` class.

4.4 User-Defined Class-Loaders

A problem that we have not tackled yet are user-defined class loaders. If a class makes use of such a class loader, without any precautions, it would circumvent our Babel Version Management System. Since every class, however, is being loaded using Babel, we can identify if a custom class loader is being loaded. As soon as this is the case, we can wrap the class loader's `findClass` and make sure that before the class loader defines the class any references to external components are being adapted as necessary.

5 Evaluation

In order to evaluate the Babel Version Management System, we have implemented a simple application which we used for our evaluation. This application uses a component that implements the Sieve of Eratosthenes which uses version 1.0 of a Primes component and another component that counts prime numbers which uses version 2.0 of the same Primes component. Additionally, we assume that the sieve, prime counter, and Primes components are third party components that have been packaged within a Java archive.

Since we do not have the source code of the components, we cannot change their code in order to load each individual component using a separate class loader as would be required by MJ [4]. Similarly, loading the sieve and Primes components using a custom class loader is not an option either since this would not allow us to access their functionality other than using reflection or wrapping them with our own interfaces. This makes this application a perfect use case for our Babel Versioning System.

Since Babel, needs to identify the component dependencies, we had to put the dependency information into the manifest file of the components. This can be achieved easily, even with third party components: one simplys unpack the Java archive, adds the required `Depends-On` line as shown in Figure 8 and repackages the Java archive.

For our first experiment, we assumed the availability of the complete dependency information. That is, every manifest file listed the components it depended

on. Once the dependencies had been set up, we had to instruct the Babel class loader to run the application as follows:

```
String classpath="...";
String mainClassname="com.ibm.sample.BaseApp";
BabelClassLoader babel=new BabelClassLoader(classpath);
babel.run(mainClassname, new Object[]{new String[]{}});
```

After looking up the main class, Babel analyzed the manifest file, and the dependency information and successfully reserved new namespaces for the individual components used by our application. Our only complaint is that we have been a bit to active in creating new namespaces for the components. That is, the package name of every component was rewritten into a new package name in order to ensure uniqueness. In our next version, we plan to put components into a new namespace only, if one of their class names collides with a class name present in another component.

In our next experiment, we assumed that only the dependency information of the sieve and prime counter components were available. Again, Babel successfully managed to run the components. In order to locate the classes of the sieve and prime counter components, Babel fell back to using Java's mechanisms of locating classes but once Babel identified that these components depended on another component, it set up a new namespace for these components and located them not only using the class name but also using their version numbers and subsequently changed their package names into a new and unique one.

Finally, we wanted to get a rough estimate of the overhead posed by our Babel class loader (Table 2). In order to compute the overhead, we implemented our example a second time where we used different package names for the two prime number components. In the first run, we wanted to see the pure overhead caused by our class loader. In the second run we wanted to see how the overhead changes when the application performs some computations.

	With Babel	Without Babel
First Run (only class loading)	275ms	29ms
Second Run (full execution)	1107ms	856ms

Table 2. Overhead Imposed by Babel

As shown in Table 2, the absolute overhead added by Babel did not increase. This is not surprising, since the only overhead added by Babel is when a new class is being loaded due to the renaming of the package names. During runtime, the length of the package names, under most circumstances, is irrelevant since Java refers to the classes via references and their indices in the constant pool.

Our application consisted of 6 classes that needed to be renamed. This makes a per-class overhead of 40ms. In a large application, this may pose some overhead during the application's startup when a large number of classes needs to be

renamed. On the other hand, our current implementation has not been optimized at all; it iterates sequentially through all the namespaces in order to determine the jar file to which a class belongs. And for rewriting the classes, it iterates again through the list of classes to be rewritten for every entry in the constant pool. Clearly, there is room for optimization.

6 Related Work

Several other approaches exist that use multiple class loaders in order to solve the versioning problem such as MJ [4], or Websphere's graph class loader [9]. These approaches are based on using a separate class loader for each module to be loaded. Their solution, however, requires the class loaders to be aware of each other so that a class is loaded by the class loader defined for the module the class belongs to and not by the class loader of the class instantiating it (i.e. accessing that class for the first time). As a consequence, their implementation cannot deal well with user-defined class loaders, a case which is handled by our Babel System. Additionally, in order to use Websphere's graph class loader, an application has to be modified in order to internally make use of the class loaders whereas Babel only requires the application to be loaded using the Babel class loader.

Whether MJ or our approach is superior depends on the application domain. If an application makes already use of class loaders in order to load plug-ins or EJBs, such as application servers, it might be more beneficial to retrofit it to use MJ because it allows modules to be unloaded. On the other hand, if the system with all the components loaded is intended to run forever, our approach is superior.

A different approach is taken by Jiazzi [13] and MultiJava [3] that both support open classes. Instead of allowing different versions of the same class to be loaded, Jiazzi supports the addition of features to classes without editing their source code. Hence, using their approach, any new method that has been added to an interface, could be added to the classes implementing that interface. Providing an implementation for such method, however, requires at least some basic knowledge about the component to be extended.

The versioning problem is not new, and has already been addressed by earlier programming languages such as Ada [10] or Modula-2 [22]. Their techniques never really took off on a larger scale, probably because they required corresponding language runtime support and failed as soon as code and libraries from different origin and/or written in different languages came together. Another reason may have been, that the approach used by many Modula-2 compilers was too rigid. For instance, if the name of a parameter within an interface was changed, the interface was considered to be of a different version and hence every module making use of that interface had to be recompiled. Using different versions of a module was not supported.

The versioning issue is not only an issue of programming language but of computing environments in general. In [20], Vinoski describes that several state-

of-the-art middleware environments such as COM and CORBA are lacking any decent versioning support.

An exception seems to be the .NET environment [21, 15, 14]. .NET provides side-by-side execution with strong version support via mechanisms such as strong assembly names (which include the version number), full and partial assembly references, configuration files, including an application configuration file, publisher policy file, and machine configuration file plus a CLR (common language runtime) and an execution environment which enforces these versioning policies.

On the operating system level, most systems provide some degree of versioning support, but mostly on a coarse-grained library basis. Component versions are typically checked by installation or package managers which provide different levels of install-time dependency and version checking. Some are pretty elaborate and their checks include dependency tree analysis based on version number and hashes or checksums of all involved libraries [16, 5], others only rely on rather unsafe methods like file naming conventions.

7 Conclusions

Our approach allows several different versions of the same class to be loaded into the same Java Virtual Machine without requiring any changes to the application's source. The only requirement is the availability of the dependency information of the dependency graph inside the manifest file or within an external configuration file. As we have shown, however, adding this information to the manifest file is straight forward and can even be done for third party Java archives.

Unlike other approaches, our approach solves the versioning problem by loading the application with our class loader that transparently rewrites the package names of conflicting elements during loading such that their names no longer clash. The advantage of our approach is that we do not have to modify any third party components in order to resolve any versioning conflicts.

The only environment that seems to address the versioning problem properly seems to be Microsoft's .NET platform, at least when we talk about code libraries on a single system and for code that is based on the .NET runtime. A standardized general solution for open environments, however, is still badly lacking! For the Java platform, our Babel Version Management system is the first step towards this direction.

References

1. John Arthorne and Chris Laffra. FAQ108 in Official Eclipse 3.0 FAQs. `http://www.eclipsefaq.org/chris/faq/html/faq108.html`, 2004.
2. John Arthorne and Chris Laffra. *Official Eclipse 3.0 FAQs*. Addison-Wesley, June 2004.
3. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of*

the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 130–145. ACM, October 2000.
4. John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: A rational module system for java and its applications. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–254. ACM, October 2003.
5. Eric Foster-Johnson. *RPM Guide*. RedHat, 2005. `http://fedora.redhat.com/docs/drafts/rpm-guide-en/`.
6. The Eclipse Foundation. The Eclipse Software Development Kit, 2005.
7. Li Gong. Secure Java class loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.
8. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, June 2005.
9. IBM. Websphere. `http://www.ibm.com/software/websphere/`.
10. ISO/IEC. *Ada Reference Manual: Language and Standard Libraries*, 2000. ISO/IEC 8652.
11. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44. ACM, 1998.
12. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.
13. Sean McDirmid, Matthew Flatt, and Wilson C Hsieh. Jiazzi: New-age components for old-fashioned java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM, November 2001.
14. Microsoft Corporation. *.NET Framework Developer's Guide: Assembly Versioning*, 2005. `http://msdn.microsoft.com/library/en-us/cpguide/html/cpconassemblyversi%oning.asp`.
15. Microsoft Corporation. *.NET Framework Developer's Guide: Using Side-by-Side Execution*, 2005. `http://msdn.microsoft.com/library/en-us/cpguide/html/cpconside-by-sidee%xecution.asp`.
16. Microsoft Corporation. *Platform SDK: Windows Installer*, 2005. `http://msdn.microsoft.com/library/en-us/msi/setup/windows_installer_sta%rt_page.asp`.
17. Rick Robinson. *Developing and Deploying Modular J2EE Applications with WebSphere Studio Application Developer and WebSphere Application Server*. IBM, June 2002. `http://www.ibm.com/developerworks/websphere/library/techarticles/0206_r%obinson/robinson.html`.
18. Alan Sommerer. The java tutorial: The extension mechanism. Technical report, Sun Microsystems, 2005.
19. Sun Microsystems. Jar file specification. `http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html`, 2005.
20. Steve Vinoski. The more things change.... *IEEE Internet Computing*, 8(1), February 2004.
21. Damien Watkins, Mark Hammond, and Brad Abrams. *Programming in the .NET Environment*. Addison-Wesley, 2003.
22. Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 2nd edition, April 1983.