Dynamic Class Loading in the Java™ Virtual Machine

Sheng Liang Gilad Bracha
Sun Microsystems Inc.
901 San Antonio Road, CUP02-302
Palo Alto, CA 94303
{sheng.liang,gilad.bracha}@eng.sun.com

Abstract

Class loaders are a powerful mechanism for dynamically loading software components on the Java platform. They are unusual in supporting all of the following features: *laziness*, *type-safe linkage*, *user-defined extensibility*, and *multiple communicating namespaces*.

We present the notion of class loaders and demonstrate some of their interesting uses. In addition, we discuss how to maintain type safety in the presence of user-defined dynamic class loading.

1 Introduction

In this paper, we investigate an important feature of the Java virtual machine: dynamic class loading. This is the underlying mechanism that provides much of the power of the Java platform: the ability to install software components at runtime. An example of a component is an applet that is downloaded into a web browser.

While many other systems [16] [13] also support some form of dynamic loading and linking, the Java platform is the only system we know of that incorporates all of the following features:

1. *Lazy loading*. Classes are loaded on demand. Class loading is delayed as long as possible, reducing memory usage and improving system response time.

2. *Type-safe linkage*. Dynamic class loading must not violate the type safety of the Java virtual machine. Dynamic loading must not require additional run-time checks in order to guarantee type safety. Additional link-time checks are acceptable, because these checks are performed only once.

3. *User-definable class loading policy*. Class loaders are first-class objects. Programmers have complete control of dynamic class loading. A user-defined class loader can, for example, specify the remote location from which the classes are loaded, or assign appropriate security attributes to classes loaded from a particular source.

4. *Multiple namespaces*. Class loaders provide separate namespaces for different software components. For example, the Hotjava™ browser loads applets from different sources into separate class loaders. These applets may contain classes of the same name, but the classes are treated as distinct types by the Java virtual machine.

In contrast, existing dynamic linking mechanisms do not support all of these features. Although most operating systems support some form of dynamic linked libraries, such mechanisms are targeted toward C/C++ code, and are not type-safe. Dynamic languages such as Lisp [13], Smalltalk [6], and Self [21] achieve type safety through additional run-time checks, not link-time checks.

The main contribution of this paper is to provide the first in-depth description of class loaders, a novel concept introduced by the Java platform. Class loaders existed in the first version of the Java Development Kit (JDK 1.0). The original purpose was to enable applet class loading in the Hotjava browser. Since that time, the use of class loaders has been extended to handle a wider range of software components such as server-side components (servlets) [11], extensions [10] to the Java platform, and JavaBeans [8] components. Despite the increasingly important role of class loaders, the underlying mechanism has not been adequately described in the literature.

A further contribution of this paper is to present a solution to the long-standing type safety problem [20] with class loaders. Early versions (1.0 and 1.1) of the JDK contained a serious flaw in class loader implementation. Improperly written class loaders could defeat the type safety guarantee of the Java virtual machine. Note that the type safety problem did not impose any immediate security risks, because untrusted code (such as a downloaded applet) was not allowed to create class loaders. Nonetheless, application programmers who had the need to write custom class loaders could compromise type safety inadvertently. Although the issue had been known for some time, it remained an open problem in the research community whether a satisfactory solution exists. For example, earlier discussions centered around whether the lack of type safety was a fundamental

To appear in the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), Vancouver, BC, Canada, October, 1998.

limitation of user-definable class loaders, and whether we would have to limit the power of class loaders, give up lazy class loading, or introduce additional dynamic type-checking at runtime. The solution we present in this paper, which has been implemented in JDK 1.2, solves the type safety problem while preserving all of the other desirable features of class loaders.

We assume the reader has basic knowledge of the Java programming language [7]. The remainder of this paper is organized as follows: We first give a more detailed introduction to class loaders. Applications of class loaders are discussed in section 3. Section 4 describes the type safety problems that may arise due to the use of class loaders, and their solutions. Section 5 relates our work to other research. Finally, we present our conclusions in section 6.

2 Class Loaders

The purpose of class loaders is to support dynamic loading of software components on the Java platform. The unit of software distribution is a class[1]. Classes are distributed using a machine-independent, standard, binary representation known as the *class file format* [15]. The representation of an individual class is referred to as a *class file*. Class files are produced by Java compilers, and can be loaded into any Java virtual machine. A class file does not have to be stored in an actual file; it could be stored in a memory buffer, or obtained from a network stream.

The Java virtual machine executes the byte code stored in class files. Byte code sequences, however, are only part of what the virtual machine needs to execute a program. A class file also contains symbolic references to fields, methods, and names of other classes. Consider, for example, a class C declared as follows:

```
class C {
   void f() {
      D d = new D();
      ...
   }
}
```

The class file representing C contains a symbolic reference to class D. Symbolic references are *resolved* at link time to actual class types. Class types are reified first-class objects in the Java virtual machine. A class type is represented in user code as an object of class java.lang.Class. In order to resolve a symbolic reference to a class, the Java virtual machine must load the class file and create the class type.

2.1 Overview of Class Loading

The Java virtual machine uses class loaders to load class files and create class objects. Class loaders are ordinary objects that can be defined in Java code. They are instances of subclasses of the class ClassLoader, shown in Figure 1. We have omitted the methods that are not directly relevant to this presentation. The ClassLoader.loadClass[2] method takes a class name as argument, and returns a Class object that is the run-time representation of a class type. The methods defineClass, findLoadedClass and findSystemClass will be described later.

```
class ClassLoader {
   public Class loadClass(String name);
   protected final Class defineClass(String name,
                                      byte[] buf, int off, int len);
   protected final Class findLoadedClass(String name);
   protected final Class findSystemClass(String name);
   ...
}
```

Figure 1: The ClassLoader class

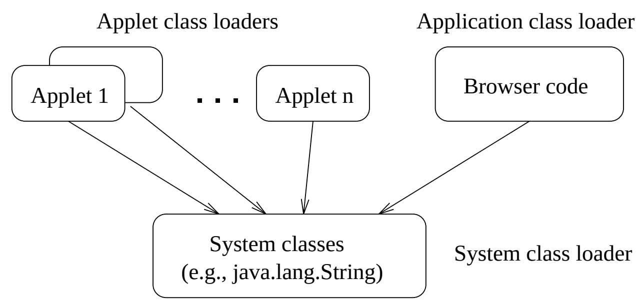

Figure 2: Class loaders in a web browser

In the above example, assume that C is loaded by the class loader L. L is referred to as C's *defining loader*. The Java virtual machine will use L to load classes referenced by C. Before the virtual machine allocates an object of class D, it must resolve the reference to D. If D has not yet been loaded, the virtual machine will invoke the loadClass method of C's class loader, L, to load D:

```
L.loadClass("D")
```

Once D has been loaded, the virtual machine can resolve the reference and create an object of class D.

2.2 Multiple Class Loaders

A Java application may use several different kinds of class loaders to manage various software components. For example, Figure 2 shows how a web browser written in Java may use class loaders.

This example illustrates the use of two types of class loaders: user-defined class loaders and the system class loader supplied by the Java virtual machine. User-defined class loaders can be used to create classes that originate from user-defined sources. For example, the browser application creates class loaders for downloaded applets. We use a

[1] Throughout this paper, we use the term *class* generically to denote both classes and interfaces .

[2] We use the notation X.m to refer to an instance method m defined in class X, although this is not legal syntax in the Java programming language.

separate class loader for the web browser application itself. All system classes (such as java.lang.String) are loaded into the system class loader. The system class loader is supported directly by the Java virtual machine.

The arrows in the figure indicate the *delegation* relationship between class loaders. A class loader L_1 can ask another loader L_2 to load a class C on its behalf. In such a case, L_1 delegates C to L_2. For example, applet and application class loaders delegate all system classes to the system class loader. As a result, all system classes are shared among the applets and the application. This is desirable because type safety would be violated if, for example, applet and system code had a different notion of what the type java.lang.String was.

Delegating class loaders allow us to maintain namespace separation while still sharing a common set of classes. In the Java virtual machine, *a class type is uniquely determined by the combination of the class name and class loader.* Applet and application class loaders delegate to the system class loader. This guarantees that all system class types, such as java.lang.String, are unique. On the other hand, a class named C loaded in applet 1 is considered a different type from a class named C in applet 2. Although these two classes have the same name, they are defined by different class loaders. In fact, these two classes can be completely unrelated. For example, they may have different methods or fields.

Classes from one applet cannot interfere with classes in another, because applets are loaded in separate class loaders. This is crucial in guaranteeing Java platform security. Likewise, because the browser resides in a separate class loader, applets cannot access the classes used to implement the browser. Applets are only allowed to access the standard Java API exposed in the system classes.

The Java virtual machine starts up by creating the application class loader and using it to load the initial browser class. Application execution starts in the public class method void main(String[]) of the initial class. The invocation of this method drives all further execution. Execution of instructions may cause loading of additional classes. In this application, the browser also creates additional class loaders for downloaded applets.

The garbage collector unloads applet classes that are no longer referenced. Each class object contains a reference to its defining loader; each class loader refers to all the classes it defines. This means that, from the garbage collector's point of view, classes are strongly connected with their defining loader. Classes are unloaded when their defining loader is garbage-collected.

2.3 An Example

We now walk through the implementation of a simple class loader. As noted earlier, all user-defined class loader classes are subclasses of ClassLoader. Subclasses of ClassLoader can override the definition of loadClass, thus providing a user-defined loading policy. Here is a class loader that looks up classes in a given directory:

```
class MyClassLoader extends ClassLoader {
  private directory;
  public MyClassLoader(String dir) {
    directory = dir;
  }
  public synchronized Class loadClass(String name) {
    Class c = findLoadedClass(name);
    if (c != null)
      return c;
    try {
      c = findSystemClass(name);
      return c;
    } catch (ClassNotFoundException e) {
      // keep looking
    }
    try {
      byte[] data = getClassData(directory, name);
      return defineClass(name, data, 0, data.length());
    } catch (IOException e) {
      throw new ClassNotFoundException();
    }
  }
  byte[] getClassData(...) { ... } // omitted for brevity
}
```

The public constructor MyClassLoader() simply records the directory name. In the definition of loadClass, we use the findLoadedClass method to check whether the class has already been loaded. (Section 4.1 will give a more precise description of the findLoadedClass method.) If findLoadedClass returns null, the class has not yet been loaded. We then delegate to the system class loader by calling findSystemClass. If the class we are trying to load is not a system class, we call a helper method getClassData to read in the class file.

After we have read in the class file, we pass it to the defineClass method. The defineClass method constructs the run-time representation of the class from the class file. Note that the loadClass method synchronizes on the class loader object so that multiple threads may not load the same class at the same time.

2.4 A Class's Initiating and Defining Loaders

When one class loader delegates to another class loader, the class loader that initiates the loading is not necessarily the same loader that completes the loading and defines the class. Consider the following code segment:

```
MyClassLoader cl = new MyClassLoader("/foo/bar");
Class stringClass = cl.loadClass("java.lang.String");
```

Instances of the MyClassLoader class delegate the loading of java.lang.String to the system loader. Consequently, java.lang.String is defined by the system loader, even though loading was initiated by cl.

Definition 2.1 *Let* C *be the result of* L.defineClass(). L *is the* defining loader *of* C*, or equivalently,* L defines C.

Definition 2.2 *Let* C *be the result of* L.loadClass(). L *is an* initiating loader *of* C*, or equivalently,* L initiates loading *of* C.

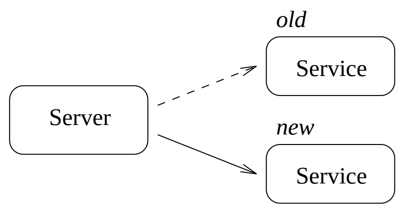

Figure 3: Class Server redirects to a new version of Service class

In the Java virtual machine, every class C is permanently associated with its defining loader. It is C's defining loader that initiates the loading of any class referenced by C.

3 Applications of Class Loaders

In this section, we give a few examples that demonstrate the power of class loaders.

3.1 Reloading Classes

It is often desirable to upgrade software components in a long-running application such as a server. The upgrade must not require the application to shut down and restart.

On the Java platform, this ability translates to reloading a subset of the classes already loaded in a running virtual machine. It corresponds to the schema evolution [3] problem, which could be rather difficult to solve in general. Here are some of the difficulties:

- There may be live objects that are instances of a class we want to reload. These objects must be migrated to conform to the schema of the new class. For example, if the new version of the class contains a different set of instance fields, we must somehow map the existing set of instance field values to fields in the new version of the class.
- Similarly, we may have to map the static field values to a different set of static fields in the reloaded version of the class.
- The application may be executing a method that belongs to a class we want to reload.

We do not address these problems in this paper. Instead, we show how it is sometimes possible to bypass them using class loaders. By organizing software components in *separate* class loaders, programmers can often avoid dealing with schema evolution. Instead, new classes are loaded by a separate loader.

Figure 3 illustrates how a Server class can dynamically redirect the service requests to a new version of the Service class. The key technique is to load the *server* class, *old* service class, and *new* service class into separate class loaders. For example, we can define Server using the MyClassLoader class introduced in the last section.

```
class Server {
  private Object service;
  public void updateService(String location) {
    MyClassLoader cl = new MyClassLoader(location);
    Class c = cl.loadClass("Service");
    service = c.newInstance();
  }
  public void processRequest (...) {
    Class c = service.getClass();
    Method m = c.getMethod("run", ...);
    m.invoke(service, ...);
  }
}
```

The Server.processRequest method redirects all incoming requests to a Service object stored in a private field. It uses the Java Core Reflection API [9] to invoke the "run" method on the service object. In addition, the Server.updateService method allows a new version of the Service class to be dynamically loaded, replacing the existing service object. Callers of updateService supply the the location of the new class files. Further requests will be redirected to the new object referenced to by service.

To make reloading possible, the Server class must not directly refer to the Service class:

```
class Server {
  private Service service;   // This is wrong!
  public void updateService(String location) {
    MyClassLoader cl = new MyClassLoader(location);
    Class c = cl.loadClass("Service");
    service = (Service)c.newInstance();
  }
}
```

Once the Server class resolves the symbolic reference to a Service class, it will contain a hard link to that class type. An already-resolved reference cannot be changed. The type conversion in the last line of the Server.updateService method will fail for new versions of Service returned from the class loader.

Reflection allows the Server class to use the Service class without a direct reference. Alternatively, Server and Service classes can share a common interface or superclass:

```
class Server {
  private ServiceInterface service; // use an interface
  public void updateService(String location) {
    MyClassLoader cl = new MyClassLoader(location);
    Class c = cl.loadClass("Service");
    service = (ServiceInterface)c.newInstance();
  }
  public void processRequest (...) {
    service.run(...);
  }
}
```

Dispatching through an interface is typically more efficient than reflection. The interface type itself must not be reloaded, because the Server class can refer to only one ServiceInterface type. The getServiceClass method must return a class that implements the same ServiceInterface every time.

After we call the updateService method, all future requests will be processed by the new Service class. The old Service class, however, may not have finished processing some of the earlier requests. Thus two Service classes may coexist for a while, until all uses of the old class are complete, all references to the old class are dropped, and the old class is unloaded.

3.2 Instrumenting Class Files

A class loader can instrument the class file before making the defineClass call. For example, in the MyClassLoader example, we can insert a call to change the contents of the class file:

```
class MyClassLoader extends ClassLoader {
   public synchronized Class loadClass(String name) {
      ...
      byte[] data = getClassData(directory, name);
      byte[] newdata = instrumentClassFile(data);
      return defineClass(name, newdata, 0,
                         newdata.length());
      ...
   }
}
```

An instrumented class file must be valid according to the Java virtual machine specification [15]. The virtual machine will apply all the usual checks (such as running the byte code verifier) to the instrumented class file. As long as the class file format is obeyed, the programmer has a great deal of freedom in modifying the class file. For example, the instrumented class file may contain new byte code instructions in existing methods, new fields, or new methods. It is also possible to delete existing methods, but the resulting class file might not link with other classes.

The instrumented class file must define a class of the same name as the original class file. The loadClass method should return a class object whose name matches the name passed in as the argument. (Section 4.1 explains how this rule is enforced by the virtual machine.)

A class loader can only instrument the classes it defines, not the classes delegated to other loaders. All user-defined class loaders should first delegate to the system class loader, thus system classes cannot be instrumented through class loaders. User-defined class loaders cannot bypass this restriction by trying to define system classes themselves. If, for example, a class loader defines its own String class, it cannot pass an object of that class to a Java API that expects a standard String object. The virtual machine will catch and report these type errors (see section 4 for details).

Class file instrumentation is useful in many circumstances. For example, an instrumented class file may contain profiling hooks that count how many times a certain method is executed. Resource allocation may be monitored and controlled by substituting references to certain classes with references to resource-conscious versions of those classes [19]. A class loader may be used to implement parameterized classes, expanding and tailoring the code in a class file for each distinct invocation of a parametric type [1].

4 Maintaining Type-safe Linkage

The examples presented so far have demonstrated the usefulness of multiple delegating class loaders. As we will see, however, ensuring type-safe linkage in the presence of class loaders requires special care. The Java programming language relies on name-based static typing. At compile time, each static class type corresponds to a class name. At runtime, class loaders introduce multiple namespaces. A *run-time* class type is determined not by its name alone, but by a pair: its class name and its defining class loader. Hence, namespaces introduced by user-defined class loaders may be inconsistent with the namespace managed by the Java compiler, jeopardizing type safety.

4.1 Temporal Namespace Consistency

The loadClass method may return different class types for a given name at different times. To maintain type safety, the virtual machine must be able to consistently obtain the same class type for a given class name and loader. Consider, for example, the two references to class X in the following code:

```
class C {
   void f(X x) { ... }
   ...
   void g() { f(new X()); }
}
```

If C's class loader were to map the two occurrences of X into different class types, the type safety of the method call to f inside g would be compromised.

The virtual machine cannot trust any user-defined loadClass method to consistently return the same type for a given name. Instead, it internally maintains a *loaded class cache*. The loaded class cache maps class names and initiating loaders to class types. After the virtual machine obtains a class from the loadClass method, it performs the following operations:

- The real name of the class is checked against the name passed to the loadClass method. An error is raised if loadClass returns a class that does not have the requested name.
- If the name matches, the resulting class is cached in the loaded class cache. The virtual machine never invokes the loadClass method with the same name on the same class loader more than once.

The ClassLoader.findLoadedClass method introduced in section 2 performs a lookup in the loaded class cache.

4.2 Namespace Consistency among Delegating Loaders

We now describe the type safety problems that can arise with delegating class loaders. The problem has been known for some time. The first published account was given by Vijay Saraswat [20].

Notation 4.1 *We will represent a class type using the notation* $\langle \mathsf{C}, L_d \rangle^{L_i}$*, where* C *denotes the name of the class,* L_d *denotes the* *class's defining loader, and L_i denotes the loader that initiated class loading. When we do not care about the defining loader, we use a simplified notation C^{L_i} to denote that L_i is the initiating loader of* C*. When we do not care about the initiating loader, we use the simplified notation $\langle \mathsf{C}, L_d \rangle$ to denote that* C *is defined by L_d.*

Note that if L_1 delegates C to L_2, then $\mathsf{C}^{L_1} = \mathsf{C}^{L_2}$.

We will now give an example that demonstrates the type safety problem. In order to make clear which class loaders are involved, we use the above notation where class names would ordinarily appear.

```
class ⟨C, L1⟩ {
   void f() {
      ⟨Spoofed, L1⟩^L1 x = ⟨Delegated, L2⟩^L1.g();
   }
}
class ⟨Delegated, L2⟩ {
   ⟨Spoofed, L2⟩^L2 g() { ... }
}
```

C is defined by L_1. As a result, L_1 is used to initiate the loading of the classes Spoofed and Delegated referenced inside C.f. L_1 defines Spoofed. However, L_1 delegates the loading of Delegated to L_2, which then defines Delegated. Because Delegated is defined by L_2, Delegated.g will use L_2 to initiate the loading of Spoofed. As it happens, L_2 defines a different type Spoofed. C expects an instance of $\langle \mathsf{Spoofed}, L_1 \rangle$ to be returned by Delegated.g. However, Delegated.g actually returns an instance of $\langle \mathsf{Spoofed}, L_2 \rangle$, which is a completely different class.

This is an inconsistency between the namespaces of L_1 and L_2. If this inconsistency goes undetected, it allows one type to be forged as another type using delegating class loaders. To see a how this type safety problem can lead to undesirable behaviors, suppose the two versions of Spoofed are defined as follows:

```
class ⟨Spoofed, L1⟩ {
   public int secret_value;
   public int[] forged_pointer;
}
class ⟨Spoofed, L2⟩ {
   private int secret_value;
   private int forged_pointer;
}
```

Class $\langle \mathsf{C}, L_1 \rangle$ is now able to reveal a private field of an instance of $\langle \mathsf{Spoofed}, L_2 \rangle$ and forge a pointer from an integer value:

```
class ⟨C, L1⟩ {
   void f() {
      ⟨Spoofed, L1⟩^L1 x = ⟨Delegated, L2⟩^L1.g();
      System.out.println("secret value = " +
                              x.secret_value);
      System.out.println("stolen content = " +
                              x.forged_pointer[0]);
   }
}
```

We can access the private field secret_value in a $\langle \mathsf{Spoofed}, L_2 \rangle$ instance because the field is declared to be public in $\langle \mathsf{Spoofed}, L_1 \rangle$. We are also able to forge an integer field in the $\langle \mathsf{Spoofed}, L_2 \rangle$ instance as an integer array, and dereference a pointer that is forged from the integer.

The underlying cause of the type-safety problem was the virtual machine's failure to take into account that a class type is determined by both the class name and the defining loader. Instead, the virtual machine relied on the Java programming language notion of using class names alone as types during type checking. The problem has since been corrected, as described below.

4.2.1 Solution

A straightforward solution to the type-safety problem is to uniformly use both the class's name and its defining loader to represent a class type in the Java virtual machine. The only way to determine the defining loader, however, is to actually load the class through the initiating loader. In the example in the previous section, before we can determine whether C.f's call to Delegated.g is type-safe, we must first load Spoofed in both L_1 and L_2, and see whether we obtain the same defining loader. The shortcoming of this approach is that it sacrifices lazy class loading.

Our solution preserves the type safety of the straightforward approach, but avoids eager class loading. The key idea is to maintain a set of loader constraints that are dynamically updated as class loading takes place. In the above example, instead of loading Spoofed in L_1 and L_2, we simply record a constraint that $\mathsf{Spoofed}^{L_1} = \mathsf{Spoofed}^{L_2}$. If Spoofed is later loaded by L_1 or L_2, we will need to verify that the existing set of loader constraints will not be violated.

What if the constraint $\mathsf{Spoofed}^{L_1} = \mathsf{Spoofed}^{L_2}$ is introduced after Spoofed is loaded by both L_1 and L_2? It is too late to impose the constraint and undo previous class loading.

We must therefore take both the loaded class cache and loader constraint set into account at the same time. We need to maintain the invariant: *Each entry in the loaded class cache satisfies all the loader constraints.* The invariant is maintained as follows:

- Every time a new entry is about to be added to the loaded class cache, we verify that none of the existing loader constraints will be violated. If the new entry cannot be added to the loaded class cache without violating one of the existing loader constraints, class loading fails.

- Every time a new loader constraint is added, we verify that all loaded classes in the cache satisfy the new constraint. If a new loader constraint cannot be satisfied by all loaded classes, the operation that triggered the addition of the new loader constraint fails.

Let us see how these checks can be applied to the previous example. The first line of the C.f method causes the virtual machine to generate the constraint $\mathsf{Spoofed}^{L_1} = \mathsf{Spoofed}^{L_2}$.

If L_1 and L_2 have already loaded the Spoofed class when we generate this constraint, an exception will immediately be raised in the program. Otherwise, the constraint will be successfully recorded. Assuming Delegated.g loads $\mathsf{Spoofed}^{L_2}$ first, an exception will be raised when C.f tries to load $\mathsf{Spoofed}^{L_1}$ later on.

4.2.2 Constraint Rules

We now state the rules for generating constraints. These correspond to situations when one class type may be referred to by another class. When two such classes are defined in different loaders, there are opportunities for inconsistencies across namespaces.

- If $\langle \mathsf{C}, L_1 \rangle$ references a field:

  $$\mathsf{T}\ \textit{fieldname};$$

  declared in class $\langle \mathsf{D}, L_2 \rangle$, then we generate the constraint:
  $$\mathsf{T}^{L_1} = \mathsf{T}^{L_2}.$$

- If $\langle \mathsf{C}, L_1 \rangle$ references a method:

  $$\mathsf{T}_0\ \textit{methodname}\ (\mathsf{T}_1, \ldots, \mathsf{T}_n);$$

  declared in class $\langle \mathsf{D}, L_2 \rangle$, then we generate the constraints:
  $$\mathsf{T}_0^{L_1} = \mathsf{T}_0^{L_2}, \ldots, \mathsf{T}_n^{L_1} = \mathsf{T}_n^{L_2}.$$

- If $\langle \mathsf{C}, L_1 \rangle$ overrides a method:

  $$\mathsf{T}_0\ \textit{methodname}\ (\mathsf{T}_1, \ldots, \mathsf{T}_n);$$

  declared in class $\langle \mathsf{D}, L_2 \rangle$, then we generate the constraints:
  $$\mathsf{T}_0^{L_1} = \mathsf{T}_0^{L_2}, \ldots, \mathsf{T}_n^{L_1} = \mathsf{T}_n^{L_2}.$$

The constraint set $\{\mathsf{T}^{L_1} = \mathsf{T}^{L_2}, \mathsf{T}^{L_2} = \mathsf{T}^{L_3}\}$ indicates that T must be loaded as the same class type in L_1 and L_2, and in L_2 and L_3. Even if, during the execution of the program, T is never loaded by L_2, distinct versions of T could not be loaded by L_1 and L_3.

If the loader constraints are violated, a java.lang.LinkageError exception will be thrown. Loader constraints are removed from the constraint set when the corresponding class loader is garbage-collected.

4.2.3 Alternate Solutions

Saraswat [20] has suggested another approach to maintaining type safety in the presence of delegating class loaders. That proposal differs from ours in that it suggests that method overriding should also be based upon dynamic types rather than static (name-based) types. Saraswat's idea is appealing, in that it uses the dynamic concept of type uniformly from link time onwards.

The following code illustrates the differences between his model and ours:

```
class ⟨Super, L1⟩ {
  void f(Spoofed x) {...code1...}
}
class ⟨Sub, L2⟩ extends ⟨Super, L1⟩ L2 {
  void f(Spoofed x) {...code2...}
}
class Main {
  public static void main(String[] args) {
    Spoofed s1 = new Spoofed();
    Sub sub = new Sub();
    Super duper = sub;
    duper.f(s1);
  }
}
```

Assume that L_1 and L_2 define different versions of Spoofed. Saraswat considers the f methods in Super and Sub to have different type signatures: Super.f takes an argument of type $\langle \mathsf{Spoofed}, L_1 \rangle$ whereas Sub.f takes an argument of type $\langle \mathsf{Spoofed}, L_2 \rangle$. As a consequence, Sub.f is not considered to override Super.f in this model.

In our model, if Main is loaded by L_2, a linkage error results at the point where f is called. The behavior in Saraswat's model is very similar: a NoSuchMethodError results.

The difference in approach becomes apparent when Main is loaded by L_1. In our model, when Main is loaded by L_1, the call to f would invoke code2. A linkage error would be raised when code2 attempted to access any fields or methods of Spoofed. In Saraswat's model the call to f executes code1 (that is, code2 does not override code1).

We believe it is better to fail in this case than to silently run code that was not meant to be executed. A programmer's expectation when writing the classes Super and Sub above is that Sub.f does override Super.f, in accordance with the semantics of the Java programming language. These expectations are violated in Saraswat's proposal.

Saraswat also suggests a modification to the class loader API that would allow the virtual machine to determine the run-time type of a symbolic reference without actually loading it. This is necessary in order to implement his proposal without the penalty of excessive class loading. We believe it would be worth exploring this idea independently of the other aspects of Saraswat's proposal.

Other proposals have also focused on changing the protocol of the ClassLoader class, or subdividing its functionality among several classes. Such changes typically reduce the expressive power of class loaders.

5 Related Work

Class loaders can be thought of as a reflective hook into the system's loading mechanism. Reflective systems in other object-oriented languages [6, 14] have provided users the opportunity to modify various aspects of system behavior. One could use such mechanisms to provide user-extensible class loading; however, we are not aware of any such experiments.

Some Lisp dialects [17] and some functional languages [2] have a notion of first-class environments, which support multiple namespaces similar to those discussed in this paper.

Dean [5] [4] has discussed the problem of type safety in class loaders from a theoretical perspective. He suggests a deep link between class loading and dynamic scoping.

Jensen et al. [12] recently proposed a formalization of dynamic class loading in the Java virtual machine. Among other findings, the formal approach confirmed the type safety problem with class loaders.

Roskind [18] has put in place link-time checks to ensure class loader type safety in Netscape's Java virtual machine implementation. The checks he implemented are more eager and strict than ours.

The Oberon/F system [16] (now renamed Component Pascal) allows dynamic loading and type-safe linkage of modules. However, the dynamic loading mechanism is not under user control, nor does it provide multiple namespaces.

Dynamically linked libraries have been supported by many operating systems. These mechanisms typically do not provide type-safe linkage.

6 Conclusions

We have presented the notion of class loaders in the Java platform. Class loaders combine four desirable features: lazy loading, type-safe linkage, multiple namespaces, and user extensibility. Type safety, in particular, requires special attention. We have shown how to preserve type safety without restricting the power of class loaders.

Class loaders are a simple yet powerful mechanism that has proven to be extremely valuable in managing software components.

Acknowledgements

The authors wish to thank Drew Dean, Jim Roskind, and Vijay Saraswat for focusing our attention on the type safety problem, and for many valuable exchanges.

We owe a debt to David Connelly, Li Gong, Benjamin Renaud, Roland Schemers, Bill Shannon, and many of our other colleagues at Sun Java Software for countless discussions on security and class loaders. Arthur van Hoff first conceived of class loaders.

Bill Maddox, Marianne Mueller, Nicholas Sterling, David Stoutamire, and the anonymous reviewers for OOPSLA'98 suggested numerous improvements to this paper.

Finally, we thank James Gosling for creating the Java programming language.

References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 49–65, October 1997.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag, August 1991. Lecture Notes in Computer Science 528.

[3] Gilles Barbedette. Schema modifications in the LISP O_2 persistent object-oriented language. In *European Conference on Object-Oriented Programming*, pages 77–96, July 1991.

[4] Drew Dean, 1997. Private communication.

[5] Drew Dean. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security*, pages 18–27, April 1997.

[6] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[8] JavaSoft, Sun Microsystems, Inc. *JavaBeans Components API for Java*, 1997. JDK 1.1 documentation, available at http://java.sun.com/products/jdk/1.1/docs/guide/beans.

[9] JavaSoft, Sun Microsystems, Inc. *Reflection*, 1997. JDK 1.1 documentation, available at http://java.sun.com/products/jdk/1.1/docs/guide/reflection.

[10] JavaSoft, Sun Microsystems, Inc. *The Java Extensions Framework*, 1998. JDK 1.2 documentation, available at http://java.sun.com/products/jdk/1.2/docs/guide/extensions.

[11] JavaSoft, Sun Microsystems, Inc. *Servlet*, 1998. JDK 1.2 documentation, available at http://java.sun.com/products/jdk/1.2/docs/ext/servlet.

[12] Thomas Jensen, Daniel Le Metayer, and Tommy Thorn. Security and dynamic class loading in Java: A formalisation. In *Proceedings of IEEE International Conference on Computer Languages, Chicago, Illinois*, pages 4–15, May 1998.

[13] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[14] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[16] Oberon Microsystems, Inc. *Component Pascal Language Report*, 1997. Available at http:/www.oberon.ch/docu/language_report.html.

[17] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition*. Department of Computer Science, Yale University, January 1984.

[18] Jim Roskind, 1997. Private communication.

[19] Vijay Saraswat. Matrix design notes. http://www.research.att.com/~vj/matrix.html.

[20] Vijay Saraswat. Java is not type-safe. available at http://www.research.att.com/~vj/bug.html, 1997.

[21] David Ungar and Randall Smith. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1987.