Negligent Class Loaders for Software Evolution

Yoshiki Sato and Shigeru Chiba

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
and Japan Science and Technology Corp
{yoshiki,chiba}@csg.is.titech.ac.jp

1 Introduction

Modern software should be dynamically evolvable. The onward sweep of technology, introducing new features, and fixing immortal security holes, drive this evolution. Moreover, practical demands of on-the-fly changes of running applications eagerly encourage the evolution since the downtime of commercial software directly causes big losses. Enabling the evolution of software components is a major focus of efforts of the designers of imperative, static typing, and high-performance languages, such as C++ and Java.

Recently, the ability of Java's class loaders, which play a central role in runtime flexibility of Java, has been considered to be insufficient for software evolution. Java class loaders do not allow reloading classes that have been loaded since to allow reloading a class tends to make several crucial runtime optimizations difficult [6]. Instead, Java class loaders provide a mechanism for using different versions of a class at the same time although the use of this mechanism is restricted. This restriction is called the version barrier.

This paper presents a negligent class loader, which can relax the version barrier between class loaders for software evolution. The version barrier is a mechanism that prevents an object of a version of a class from being assigned to a variable of another version of that class. In Java, if a class definition (i.e. class file) is loaded by different class loaders, different versions of the class are created and regarded as distinct types. If two class definitions with the same class name are loaded by different loaders, two versions of the class are created and they can coexist while they are regarded as distinct types. The version barrier is a mechanism for guaranteeing that different versions of a class are different types. Regarding two versions as distinct types is significant for performance reasons. If not, advantages of being a statically typed language would be lost.

2 Motivations

First, this section shows some practical examples that bring to us inconvenience of the current Java class loaders.

2.1 Dynamic aspect weaver

Dynamic AOP (Aspect-Oriented Programming) is receiving interests growing in both the academia and the industry. Unlike static AOP systems, dynamic AOP systems allow dynamically weaving and unweaving an aspect into/from a program. Moreover, advice and pointcuts can be changed during runtime. These dynamic features extend the application domain of AOP. Dynamic AOP can shorten the lead time of the edit-deploy-run cycle of software development. It can also allow using aspects for making the behavior of application software adaptable to changes of the runtime environment and requirements.

The version barrier makes it difficult to implement an instance-based dynamic AOP system. Such a dynamic AOP system allows weaving a different aspect with a particular instance of a class. A simple implementation of such an AOP system would use multiple class loaders, each of which loads a different version of a class woven with a different aspect. However, this implementation approach does not work because instances of those versions are not compatible because of the version barrier (Figure 2.1). Therefore, most of dynamic AOP systems adopt complicated implementation techniques such as static code translation [8] [1], just-in-time hook insertion [11], and modified JVM [9] [2] although these techniques imply certain performance penalties.

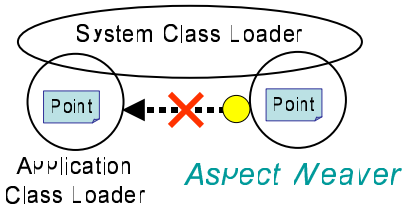

Fig. 1. Aspects can not be woven into loaded classes.

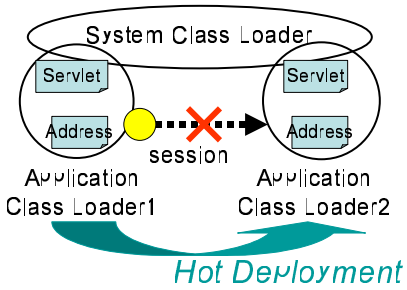

Fig. 2. Hot deployment discards all internal states.

2.2 Hot deployment for application servers

Most of application servers including both commercial (Websphere, Weblogic, etc.) and open-source (JBoss, Tomcat, etc.) provide the hot deployment functionality, which enables software components (EJB-JAR, EAR, or WAR package) to be plugged and unplugged without restarting application servers. This dramatically improves the productivity of software development. To be separately deployed and undeployed at runtime, each component is loaded by a distinct class loader. Thus, a J2EE application can be dynamically customized per component.

However, if a new version of a component is deployed for software evolution, instances of the new version cannot be exchanged for instances of the old version of that

component because of the version barrier (Figure 2.1). Such instances are caches, cookies, and session objects and session beans. These instances should be passed to the new component through the shared container of the application server. This is remarkably inconvenient in practice. That is to say, J2EE application servers provide hot deployment but not “hot evolution”.

The version barrier makes it difficult to include a copy of a shared class in every component. If two components share a class, for example, for exchanging data, each component should be able to include a copy of that shared class since an ideal component should contain all the class files that are necessary for running an application. However, the version barrier disables exchanging an instance of that shared class between the two components. To avoid this problem, JBoss provides the UCL (Unified Class Loader) architecture [7] but this architecture prevents different components from including different classes with the same name.

3 Negligent class loaders

We propose a *negligent class loader (NCL)*. The NCL can relax the version barrier if an incoming object is an instance of a class loaded by a sibling class loader specified by the programmer in advance. Here, a sibling means a class loader sharing the same parent loader. For example, a NCL allows assigning an instance of class Customer that has been loaded by a sibling to a variable of Customer loaded by the NCL. To keep consistency, differences between the two versions of Customer must satisfy the rules described below.

Naively relaxing the version barrier may cause serious security problems. For example, a program may access a non-existing method and then crash the JVM. In fact, the version barrier of Sun JDK 1.1 was accidentally relaxed and thus it had a security hole known as the type-spoofing problem first reported by Saraswat [10]. This security hole was solved by the loader constraint scheme [5], which strengthens the version barrier. To avoid this security problem, runtime type checking is necessary. For example, dynamically typed languages such as CLOS and Smalltalk do not provide the version barrier. Since a variable is not statically typed, any type of instance can be assigned to a variable. For security, these languages perform runtime type checking so that, if a non-existing method or field is accessed, an exception will be thrown at runtime. A drawback of this approach is that it requires frequent runtime type checking, which implies non-negligible performance degradation.

The NCL restricts the relaxation of the version barrier only to the classes loaded by a sibling class loader so that runtime overheads due to the relaxation will be reduced. Suppose that a sibling class loader loads a class S and then the NCL loads a class N. The NCL allows an instance of S to be assigned to a variable of the type N if the following condition is satisfied:

1. The class S includes all the members such as methods and fields of the class N. The method bodies can be different between N and S.
2. Or, the class N includes all the members of the class S. In this case, a new class S’ is dynamically generated and a reference in instances of S is changed to indicate

the internal type information block representing S' instead of S. S' is a copy of S but it also includes the methods and constructors included in N but not in S. These methods in S' are called secure handling functions. If they are accessed, they throw a runtime exception representing illegal access. This is for avoiding a serious security hole that can be used for buffer overflow attacks. Thereby the secure execution is guaranteed with respect to N and S as in dynamically typed languages. Without that substitution of S' for S, a program written for N may violate the boundary of the method table in S.

In both the cases, the class N must have the same super class as the class S and it must not weaken access restriction against the class S. For example, if a method is public in S, then the method in N must be also public.

The type information block (TIB) referred to by an instance contains a method table that holds function pointers to a corresponding method body. For secure execution, the NCL creates the TIB of the class N to be compatible as much as possible with that of the class S when the class N is loaded. In particular, the order of the method table entries in the TIB of the class N must be the same as the order in the TIB of the class S with respect to the methods that both the classes S and N include. Otherwise, for example, a program written for N might invoke a wrong method body if the targt instance is of S. The algorithm for constructing the TIB is shown in Figure 3.

In our architecuture, runtime type checks are performed only when the `checkcast` bytecode instruction is executed. The `checkcast` instruction examines the version of an instance and, if it does not satisfy the condition above, it throws the `ClassCastException`. To do this, we modify the JVM. Other instructions such as `invokevirtual`, `getfield` and assignment instructions like `astore`, do not have to examine the condition. This is because the NCL relaxes the version barrier only for sibling class loaders, which satisfy the bridge-safety property [10]. If this property is satisfied between two class loaders, instances of a class `C` loaded by one class loader are always upcast to a class loaded by their parent class loader before the instances are passed to a class loaded by the other class loader. For example, the instances will be upcast to a super class or an interface of the class C, such as the `Object` class, which is loaded by the parent class loader. Thus explicit downcast must be executed before those instances are assigned to a variable of a type loaded by the NCL (Figure 3).

4 Related Work

There are a number of research activities for software evolution. Liang and Bracha [5] described a programming technique of using an interface type as the type of a variable that can refer to instances of multiple versions of a class. However, this technique requires programmers to define an interface type for every multi-versioned class and access instances of the class through the interface type. Hjalmtysson and Gray [3] implemented dynamic classes in C++ by using templates. Their system uses wrapper (or proxy) classes and methods. This approach does not require runtime system support or language extensions. However, it implies performance penalties. Malabarba *et al.* [6] modified the JVM to make a class reloadable at runtime. However, it also implies runtime penalties because of difficulties in performing runtime optimization techniques,

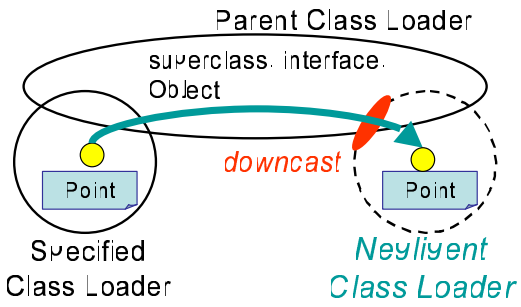

Fig. 3. Downcast enforced by the bridge safety between the NCL and a sibling class loader.

```
if S has been already loaded && S's loader is a sibling of N's one then
   for each member m in S {
      if N includes the member m then
         Push the member m into the type information block of N
      else
         Push a secure handling function into the type information block of N
      end
   }
end
```

Fig. 4. Pseudo code for constructing the internal type information block.

such as method inlining and quick bytecode instructions. The Java2 SDK1.5 will support the hot swap mechanism with the `java.lang.instrument` package. Although it enables reloading a class to a certain degree, a new version of a class does not include a method or field that was not included in the previous version.

5 Current State

We are currently implementing a negligible class loader on the IBM Jikes Research Virtual Machine [4]. We will use this implementation to evaluate the performance overheads of our approach. In addition, we will study the proof of the type safety of our system, and reason about our approach with respect to the Java security architecture. Furthermore, we have not considered how to correctly handle arrays of multi-versioned class types.

References

1. Baker, J., Hsieh, W.: Runtime Aspect Weaving Through Metaprogramming. In: 1st International Conference on Aspect-Oriented Software Development. (2002) 86–95

2. Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual Machine Support for Dynamic Join Points. In: International Conference on Aspect-Oriented Software Development. (2004)
3. Hjálmtyśson, G., Gray, R.: Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In: In Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, USENIX (1998)
4. IBM: Jikes Research Virtual Machine. (2001)
5. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Proceedings of OOPSLA'98, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications. Number 10 in SIGPLAN Notices, vol.33, Vancouver, British Columbia, Canada, ACM (1998) 36–44
6. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime Support for Type-Safe Dynamic Java Classes. In: ECOOP 2000 - Object-Oriented Programming, 14th European Conference. Volume 1850 of Lecture Notes in Computer Science., Springer (2000) 337–361
7. Marc Fleury, F.R.: The JBoss Extensible Server. In: ACM/IFIP/USENIX International Middleware Conference. Volume 2672 of Lecture Notes in Computer Science., Rio de Janeiro, Brazil, Springer (2003) 344–373
8. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A flexible framework for AOP in Java. In: Reflection 2001. (2001) 1–24
9. Popovici, A., Alonso, G., Gross, T.: Just in Time Aspects: Efficient Dynamic Weaving for Java. In: 2nd International Conference on Aspect-Oriented Software Development. (2003) 100–109
10. Saraswat, V.: Java is not type-safe. (1997)
11. Sato, Y., Chiba, S., Tatsubori, M.: A Selective, Just-In-Time Aspect Weaver. In: Second International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt Germany (2003) 189–208