Loosely-Separated "Sister" Namespaces in Java

Yoshiki Sato* and Shigeru Chiba

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

Abstract. Most modern programming systems such as Java allow us to link independently developed components together dynamically. This makes it possible to develop and deploy software on a per component basis. However, a number of Java developers have reported a problem, ironically called the *version barrier*, imposed by the strict separation of namespaces. The version barrier prohibits one component from passing an instance to another component if each component contains that class type. This paper introduces a novel concept for Java namespaces, called *sister namespaces*, to address this problem. Sister namespaces can relax the version barrier between components. The main purpose of this paper is to provide a mechanism for relaxing the version barrier, while still allowing type-safe instance accesses between components with negligible performance penalties in regular execution.

1 Introduction

Practically all modern programming environments allow developers to utilize some kind of component system (e.g., JavaBeans [9], EJB [22], CORBA [25], .NET/DCOM/ActiveX [24], Eclipse plug-ins [29]). A component system allows programmers to develop a component-based application, which can be developed and then deployed per component. Most of the component systems for Java adopt a single class loader per component, and thereby create a unique namespace for each application component. A namespace is a map from the class names to the class definitions. A set of classes included in the same component *joins* [1] its own namespace and thus naming conflicts between components can be avoided. Moreover, a component can be dynamically and individually updated without restarting the whole execution environment.

One significant drawback of such component systems for Java is the difficulty for components to communicate across class loader boundaries in the Java Virtual Machine (JVM) [13, 20, 8, 14]. In fact, such communication is well known to frequently cause a cast error `ClassCastException` or a link error `LinkageError`. Most of the link errors are just bugs; an error is caused when a class is wrongly

* Currently, Mitsubishi Research Institute, Inc., Japan.

[1] A class joining a namespace means it is being loaded by the class loader that creates the namespace.

A.P. Black (Ed.): ECOOP 2005, LNCS 3586, pp. 49–70, 2005.

loaded by both parent and child loaders [16]. These bugs can be easily avoided if developers are careful. However, cast errors are extremely difficult to avoid since this problem is caused by the strict separation of namespaces, ironically called the *version barrier*. The version barrier is a mechanism that prevents a version of a class type from being converted to another version of that specific class type. For instance, it restricts an instance of the former type to be assigned to the variable of the latter type.

In Java, a class type is uniquely identified at runtime by the combination of a class loader and a fully qualified class name. If two class definitions with the same class name are loaded by different loaders, two versions of that class type are created and they can co-exists, although they are regarded as distinct types. The version barrier is a mechanism for guaranteeing that different versions of a class are regarded as different types. This guarantee is significant for performance reasons. If different versions of a class were not regarded as different types, the advantages of being a statically typed language would be lost. Moreover, if the same class definition (i.e., class file) is loaded by different class loaders, different versions of that class are created and regarded as distinct types. Therefore, if two components load the same class file individually, one component cannot pass an instance of that class type to the other.

This paper presents our novel concept of namespaces in Java, which we call *sister namespaces*, and the design of that mechanism. Sister namespaces can relax the version barrier between application components. An instance can be carried beyond the version barrier between sister namespaces if the type of that instance is compatible between these namespaces. The main purpose of this paper is to provide a mechanism for relaxing the version barrier while keeping type-safe instance accesses with negligible performance penalties in regular execution. The mechanism of sister namespaces is implemented by extending the type checker and the class loader of the JVM.

The rest of this paper is organized as follows. Section 2 describes two problems that cause trouble for component-based application developers. Section 3 presents the design and implementation of the sister namespace. Section 4 discusses a few implementation issues. Section 5 presents the results of our experiments. Section 6 compares the sister namespace mechanism to other related work. Section 7 concludes this paper.

2 Problems of the Version Barrier

This section presents two problems that developers often encounter when developing a component-based application in Java. These problems are actually caused by the version barrier between namespaces.

2.1 J2EE Components

Most J2EE platforms, either commercial (e.g., Websphere, Weblogic) or open-source (e.g., JBoss, Tomcat), support both the development and the deployment

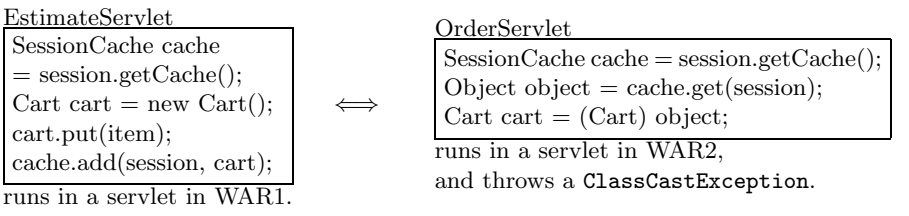

Fig. 1. Passing the session cache from one WAR component to another

of pluggable component archives (EJB-JARs, WARs, and EARs). A Web Application Archive (WAR file) is used to deploy a web-based application. This file can contain servlets, HTML files, Java Server Pages (JSPs), and all associated images and resource files. An Enterprise Application Archive (EAR file) may contain one or more Enterprise JavaBeans (EJBs) and WARs. The functionality of the so-called hot deployment enables such J2EE components to be plugged and unplugged at runtime without restarting the application servers. Thus, a J2EE application can be dynamically customized on a per-component basis. This dramatically improves the productivity of software development. For enabling hot deployment, each component joins a distinct namespace, loaded by a distinct class loader.

However, the version barrier makes it impossible to pass instances of each version of a class across the boundary of J2EE components, or namespaces. Such instances are typically caches, cookies, or session objects or beans. For example, consider the following scenario. An instance of the `Cart` class must be passed between servlets included in different web application archives (that is, from the `EstimateServlet` included in one web archive, WAR1, to the `OrderServlet` in another web archive, WAR2). The class file of the `Cart` class is packaged into a Java Archive (JAR) file, and identical copies of that JAR file reside in the `WEB-INF/lib` directories in each web archive. Thus, each class loader loads the `Cart` class separately. Figure 1 illustrates the implementation of these servlets: `EstimateServlet` puts an instance of `Cart` into the session cache and `OrderServlet` pulls that instance out of the cache. When casting it from `Object` to `Cart`, the JVM will throw a `ClassCastException`. Since the `Cart` class referenced by the `EstimateServlet` class is a distinct type from the type referenced by the `OrderServlet` class, the version barrier prevents assignment of that instance to the variable `cart` in the `OrderServlet` class by throwing a cast error in advance.

Some readers might think the delegation model of class loaders in Java is a solution to the problem above. These WAR components can share the same version of a class if they delegate the loading of that class to their common parent, such as the `EAR` class loader (Figure 2). In fact, the typical J2EE platform has such a common parent loader. Child class loaders can have their parent loader load a class if they want to share the same version of that class. In the case of J2EE, the `SystemClassLoader` is the parent of all EAR class loaders and an EAR class loader is, in turn, the parent of all WAR class loaders included

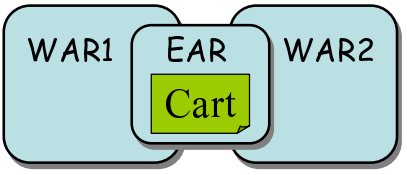

Fig. 2. A parent EAR class loader is used for sharing class types between WAR1 and WAR2. The rounded box represents a namespace for the J2EE component. The overlapping part means the overlapped namespace

Fig. 3. The JBoss application server based on the unified class loader architecture makes a parent-child relationship between the communicating components

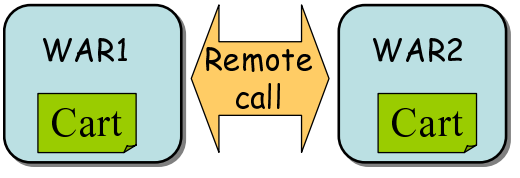

Fig. 4. All inter-component communications are realized by a remote call

in that EAR. However, the solution using a parent class loader tightly couples several irrelevant J2EE components together. Such coarse-grained composition decreases the maintainability and availability of all related software components. For example, consider the two components `DVDStore` and `Pizzeria`: the former models an online 24-hour DVD store and the latter models an online home delivery pizzeria available from hours 10 to 21. If both of these components share the abovementioned application component including `Cart` and if this component is packaged into `Pizzeria`, then undeploying `Pizzeria` for maintenance stops the service by `DVDStore`. Since `DVDStore` must run 24 hours a day, it is almost impossible to decide the maintenance schedule of `Pizzeria`.

To solve this problem, the JBoss application server provides the unified class loader (UCL) architecture [21] for sharing across components across the J2EE components. A collection of UCLs acts as a single class loader, which places into a single namespace all the classes to be loaded. All classes are loaded into the shared repository and managed by this repository. However, this architecture disables different J2EE components with the same name (Figure 3).

Another technique, considered a last resort, is using the Java Serialization API to exchange objects between different J2EE components through a byte stream, which is the referred to as *Call-by-Value* (Figure 4). Typical J2EE platforms adopt this approach for inter-EAR communications. However, even if an EAR wants to transfer an object to another EAR deployed in the same container (or JVM), it must execute a remote call. This remote call is a waste of I/O re-

sources and it decreases the overall performance. Although the Local Interface mechanism introduced in EJB2.0 allows communications between components without remote calls (*Call-by-Reference*), these components must be packaged together in the same archive.

2.2 Eclipse Plug-in Framework

The Eclipse platform [37], an integrated development environment for Java, can be considered as a component system due to its advanced plug-in framework. A plug-in module can contain all sorts of resources, including code and documentation. A plug-in module must also contain sufficient information for the platform to incorporate the code or documentation into itself. The plug-in framework allows us to easily add, update and remove discrete portions of the contents. In addition, since a separate class loader (called a plug-in class loader) is created for each plug-in module, each plug-in module has its unique namespace and is dynamically deployable.

However, the Eclipse plug-in framework has a structural problem due to the version barrier. For example, consider the Eclipse help system plug-in module [12]. It is a useful plug-in module that allows users to develop and deploy professional-quality, easy-to-use, and searchable online documentation. The Eclipse help system can be used as an infocenter, which is an application implemented as a web component and accessible from a web browser. However, to be used as an infocenter, the current Eclipse help system needs to run on a separate process from the process of the web server (Figure 5). The web server must make new processes for the help system and the minimum Eclipse system, and then the web server must dispatch all requests to the help system. Thus, every communication for dispatching requests from the web server to the help system is a remote call, which involves marshalling all passed instances.

A real problem of the example above is that, no matter which namespace the help system joins, all instances must be marshaled and unmarshaled with performance penalties to avoid trouble due to the version barrier when they are passed between the web server and the help system. This is true even if the help system is run on the same process as the web server. Suppose that the

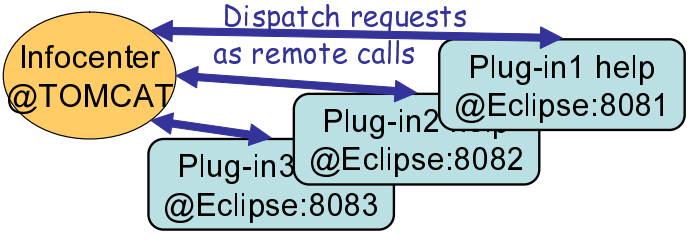

Fig. 5. The Eclipse help system must run as a separate process

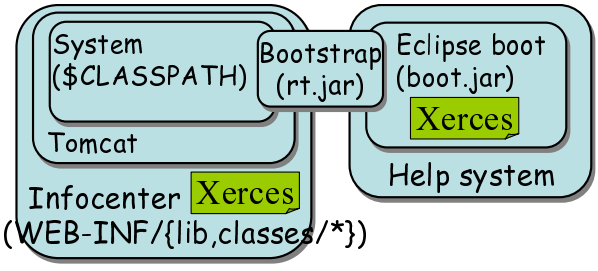

Fig. 6. The Xerces archives are loaded in duplicate for the Eclipse help system and the infocenter

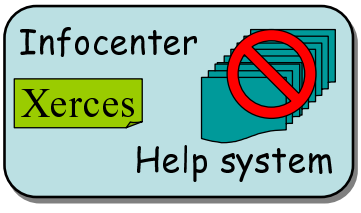

Fig. 7. Loading all components by a class loader breaks the isolation of each namespace

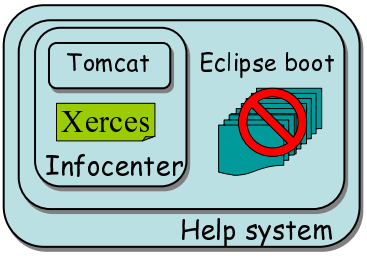

Fig. 8. Delegating the Xerces archives to the web component class loader breaks the isolation of the help system

help system runs on the same JVM as the infocenter, and both the help system and the infocenter use the Apache Xerces [36] archive, which contains an XML parser in the `WEB-INF/lib` directory. If the help system joins a namespace independent of the namespace of the infocenter (Figure 6), the version barrier does not allow the instances of an XML parse tree to be exchanged between the help system and the infocenter, since the copies of the Xerces archive are loaded in duplicate and then different versions of the tree-node class types are created for each archive. If the help system joins the same namespace as the infocenter by deploying as a WAR file into the `WEB-INF/lib` directory (Figure 7), the XML parse tree can be exchanged between the two components. However, this obviously breaks the isolation of the help system from the infocenter. For example, several core components of the Eclipse platform must also be loaded together with the help system, and these core components cause naming conflicts with the infocenter. Furthermore, all the components must be redeployed together when some of the components are redeployed for maintenance. Finally, if the help system joins a descendant namespace of the infocenter (Figure 8), delegating the Xerces archives to the parent class loader also allows sharing the Xerces archives. However, it ends up breaking separated namespaces, too.

2.3 Extending Assignment Compatibility

The problems illustrated above can be solved if the algorithm for computing *assignment compatibility* in the Java programming language is extended to include version conversions between different versions of a class type. Here, the version conversion means a conversion from a version of a class type to any other version of that class type. If this conversion is chosen in the context of assignment, casting, and method invocation conversions such as widening and narrowing conversions, instances could be easily passed across the version barrier [2]. For example, this extension of assignment compatibility would allow assignments between different versions of a class type. Thus, a component would be able to pass instances into and from another component, even if both components load and define that class type separately. The `OrderServlet` class in Figure 1 would not throw a cast error. Moreover, the Eclipse platform would not need to care about where and how many Xerces libraries are available in the current execution environment.

However, naively relaxing the version barrier by extending the assignment compatibility causes a serious security problem. For example, a program may access a non-existing field or method and then crash the JVM. In fact, the version barrier of Sun JDK 1.1 was wrongly relaxed, and thus it had a security hole known as the type-spoofing problem, first reported by Saraswat [26]. This security hole had been solved by the loader constraint scheme [18], which rather strengthens the version barrier. To avoid this security problem while relaxing the version barrier, it would be necessary to have runtime type checking, as is found in dynamically typed languages such as CLOS, Self, and Smalltalk. In such languages, since a variable is not statically typed, any type of instance can be assigned to it. For security, several interpreters for dynamically typed languages perform runtime type checks, called guard tests, so that an exception can be thrown at runtime if a non-existing method or field is accessed. A drawback of this approach is that it requires frequent runtime type checks, which implies non-negligible performance degradation, whereas the JVM performs these runtime type checks. Another technique is to perform runtime type checks at every assignment operation, such as the `aastore` Java bytecode instruction, which is used for storing an object reference in an array object. This operation verifies that the stored object is type-safe. However, this approach also causes performance degradation, since the JVM must perform a type-check for not only `aastore` but also for a large number of other assignment instructions.

3 Sister Namespaces

We propose *sister namespaces*, which can relax the version barrier between namespaces. Different versions of a class type that join sister namespaces can be

[2] If two class types have assignment compatibility with each other, one type can be converted to the other type in the context of not only assignment conversions but also casting and method invocation conversions.

assignment compatible with each other if these versions have differences while still preserving the *version compatibility*. Our challenge is to relax the version barrier while keeping type-safe instance accesses efficient. In this section, we first define extended assignment compatibility, which is based on Java binary compatibility [7] (Section 3.1). Next, we show the sister-supported type checker, which takes the central role in relaxing the version barrier for sister namespaces (Section 3.3). The type checker blocks illegal objects when they move across the version barrier, and thus no subsequent extra check is needed for these objects. This is enabled because it is prohibited for a namespace to become a sister of its parent or child namespace. In addition, we present the sister loader constraint (Section 3.4) and then the schema class loading scheme (Section 3.5). They prevent eager class loading and type inconsistencies, respectively.

We implemented the sister namespaces on the IBM Jikes Research Virtual Machine (JRVM) [1]. The extensions to the JRVM are only the sister-namespace API, a sister-supported class loader, and a sister-supported type checker. The API is provided as an extension to the existing `java.lang.ClassLoader` in the GNU Classpath libraries. These extensions consist of several core classes of the JRVM such as class and object representations.

3.1 Version Compatibility

This section provides the definition of version compatibility, which securely extends the assignment compatibility between different versions of a class type. We define two versions of a class type, C_{ver1} and C_{ver2}, as assignment compatible with each other if C_{ver1} is version compatible with C_{ver2} and vice versa. A class type C_{ver2} is version compatible with C_{ver1} if all the class types that could previously link with C_{ver1} and work with an instance of C_{ver1} without errors are able to also correctly work with instances of C_{ver2} without other assignment compatibility rules such as a subtyping relation. Thus, if C_{ver1} and C_{ver2} are version compatible, then an instance of C_{ver1} can be securely converted to the type C_{ver2} when it is assigned to a variable of C_{ver2} and vice versa. Here, being secure means that every operation on C_{ver2} is applicable to the instance of C_{ver1} without errors; any method call, field access, or type casting applied to the variable does not fail.

The following are the differences that programmers are permitted to make between two versions of a class while preserving version compatibility between the two versions:

- Differences of declared static members such as a static field, a static method, a constructor, or an initializer.
- Differences of the implementation of instance members, such as an instance method.

The differences are derived from the study of the binary compatible changes mentioned in the Java language specification [11].

Version compatibility is based on the idea of binary compatibility; it means that an instance rather than a class can work with the binary of another ver-

sion of the class type. Java binary compatibility defines a set of changes that developers are permitted to make to a package or to a class or interface type while preserving the compatibility with the preexisting binaries. A change to a class type is binary compatible with preexisting binaries if preexisting binaries that previously linked without errors will continue to link without recompiling. Version compatibility defines differences between two versions of a class type that preserve the binary compatible property between an instance of one version and the binary of the other version. Unlike the original binary compatibility, the version compatibility allows any change to static members since static member accesses are irrelevant to instances. Version compatibility deals with the compatibility between an instance and the binary of another version of that class type. Therefore, to be version compatible, two versions of a class type must have the same set of private members, although the implementations of those members may differ. This is a difference from the binary compatibility, which allows the two versions to have a different set of private members. Since a private member can be accessed from not only `this` instance but also from other instances of another version of that class type, version compatibility requires that the two versions have the same set of private members.

3.2 Creating Sister Namespaces

A sister namespace is a first-class entity, but it is created implicitly when a class loader is instantiated with a class loader given as a parameter. The `ClassLoader` class provides the new constructor as follows:

```
protected ClassLoader(ClassLoader parent, ClassLoader sister)
```

The class loader obtained from this constructor becomes a *sister class loader* of the class loader specified by the parameter `sister`. The latter class loader also becomes a sister of the former one. These two sister class loaders construct their own sister namespaces; the version barrier between them is relaxed if the version compatibility is satisfied. The sister class loaders must not have a parent-child relationship. This rule is significant for the efficient type checking we describe later. If the sister class loaders have such a relationship, the construction of the sister namespaces fails. In this paper, if a version of a class type is loaded earlier (or later) than other versions, it is called a *younger* (or *older*) sister version of that class type. This young-old relationship is independent of the creation order of the sister class loaders.

In the case described in Section 2.1, the application programmers can exchange instances between two namespaces for WAR1 and WAR2 if they are sister namespaces. Each namespace can contain a different version of the type of the exchanged instance. WAR1 and WAR2 must be loaded by the class loaders created as follows:

```
ClassLoader ear = new EARClassLoader();
ClassLoader war1 = new WARClassLoader(ear);
ClassLoader war2 = new WARClassLoader(ear, war1);
```

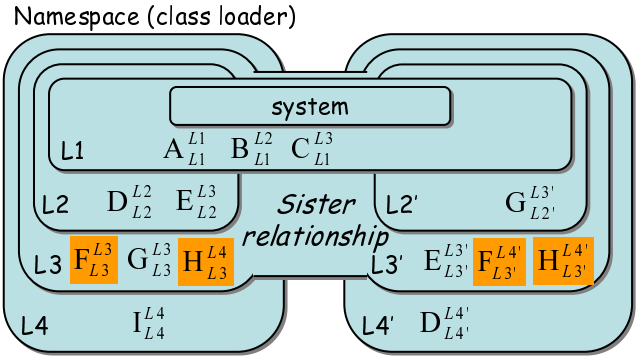

Fig. 9. The notation $C^{L_i}_{L_d}$ represents a class type, where C denotes the name of the class, L_d denotes the class's defining loader, and L_i denotes the loader initiated class loading. An inclusion relation represents a parent-child relationship. For example, the class loader L1 is a parent of both L2 and L2'. And the classes A, B, C, and the system classes are visible in the namespaces L2, L2', L3, L3', L4, and L4'. In this figure, the sister namespace L3 and L3' have a sister relationship

The `ear`, `war1`, and `war2` are instances of the `ClassLoader` class. The third `new` operation creates sister namespaces for WAR1 and WAR2. Both `war1` and `war2` have the same parent class loader `ear`. In general, application programmers of components, such as Applets, Servlets, Eclipse plug-ins, and EJB, do not have to be aware of namespaces or class loaders. These are implicitly managed by the application middleware. Creating sister namespaces by using the `ClassLoader` constructor above is the work of middleware developers. A sister namespace can make another plain namespace its sister on demand. Since the sister relationship is transitive, if a namespace becomes a sister of a namespace and then it becomes a sister of another namespace, all three namespaces become sisters of each other. Programmers can incrementally create a new namespace and make it another sister of the other sister namespaces. This feature would be useful in cases of incremental development processes and routine maintenance work.

Note that all class types *defined* by a sister class loader can be version compatible with the corresponding sister version of that class type, even if the loading of these class types are *initiated* by the child class loaders. An *initiating* class loader, which initiates the loading of a class type, does not have to actually load a class file. Instead, it can delegate to the parent class loader. The class loader that actually loads a class file and defines that type is called a *defining* class loader of that type. This delegation mechanism is used for sharing the same version of class type between the initiating and defining class loaders. In Figure 9, if two sister namespaces are created between class loaders L3 and L3', the classes F and H can be version compatible with F' and H', respectively. The pairs E and E' or G and G' are not compatible with each other since they are defined by other class loaders.

3.3 Sister-Supported Type Checking

The version barrier is relaxed by a type checker that considers the sister namespaces. In Java programs, most bytecode instructions such as the method invocation instructions `invokevirtual` and `invokenonvirtual`, and field access instructions such as `getfield` and `putfield` are statically typed. These instructions do not perform dynamic type checking. Therefore, these instructions *as they are* can work correctly with any version of class type if they are version compatible. On the other hand, several instructions such as `instanceof`, `checkcast`, `invokeinterface`, `athrow`, and `aastore` entail dynamic type checking. The type checking by those instructions must be enhanced if the version barrier is relaxed so that version compatible instances can be passed between sister namespaces. The algorithm of enhanced type checking for sister namespaces is shown in Figure 10. After the regular type checks are performed, and if they fail (line 2), the extra checks are executed (lines 3–6). First, a sister relationship is examined (line 3). If the left-hand side class type (`LHS`) and the right-hand side class type (`RHS`) have a sister relationship, then the type checker determines whether one class type has undergone the schema compatible loading process against the other type (line 4). Schema compatible loading is introduced later.

Note that these extra checks for sister namespaces are executed only after the regular type checks fail. Since typical programs do not frequently cause type errors, this enhancement for the built-in type checker implies no performance penalties as long as instances are not passed between sister namespaces.

The sister-supported type checker only prohibits a version incompatible instance from being passed between sister namespaces. A version incompatible class can join each of the sister namespaces if an instance of that version stays within the namespace. To avoid the security problem described in Section 2.3 (naively relaxing the assignment compatibility), sister namespaces must detect a version incompatible instance being passed between sister namespaces. This detection is executed by only the `checkcast` instruction. In other words, the detection is not executed by other instructions for method invocation, field access,

```
1: if LHS is a subtype of RHS then true
2: else if LHS is not a subtype of RHS then
3:     if LHS is a sister type of RHS &&
4:         LHS is version compatible with RHS then true
5:     else false
6:     end
7: end
```

Fig. 10. Pseudo code for enhanced type checking for sister namespaces. A type check is the determination of whether a value of one type, hereafter the right-hand side (RHS) type , can legally be converted to a variable of a second type, hereafter the left-hand side (LHS) type. If so, the RHS type is said to be a subtype of the LHS type and the LHS type is said to be a supertype of the RHS type

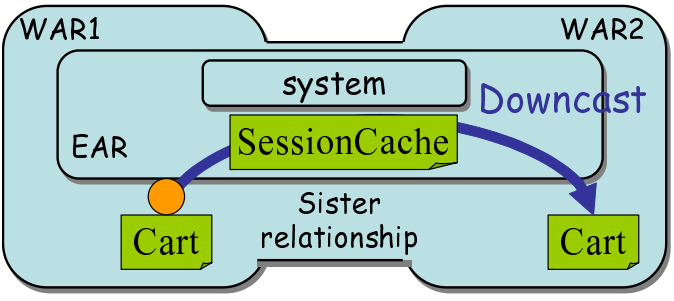

Fig. 11. Downcast enforced by the bridge-safety property satisfied between namespaces

and assignment. This is mainly due to the design of sister namespaces, which must not have a parent-child relationship between them. This rule brings the *bridge-safety* [26] property to all classes included in the sister namespaces. This property guarantees that an instance of a class type is always examined by the `checkcast` instruction when it is passed between sister namespaces. It must be first upcast to a type loaded by the common parent class loader of the two sister class loaders, and then it must be downcast before it is assigned to the class type loaded by the sister class loader at the destination. For example, when an instance of `Cart` is passed, it will be first upcast to a super class of `Cart`, such as the `Object` class, and then downcast to another version of the `Cart` class (Figure 11). Therefore, the `checkcast` instruction is always executed when the instance is downcast to `Cart`.

To implement the sister-supported type check, we modified the `VM_DynamicTypeCheck` class in the JRVM. We extended that class and the TIB (Type Information Block) for fast type checking to consider sister relationships. The original TIB holds several arrays of type identifiers. For example, the arrays of extended superclass types and of implemented interface types are stored in the TIB for fast type checking without looking up the whole type hierarchy [3][2]. Similarly, the extended TIB holds two arrays of `sid`s. The `sid` is the identifier of a sister relationship. The two arrays are of the `sid`s of the extended superclasses and the `sid`s of the implemented interfaces. The `sid` of a class can be obtained from a `VM_Class` object representing that class. We extended the `VM_Class` class to hold the `sid` of the class.

3.4 Sister Loader Constraint

A straightforward implementation of the sister-supported type checker requires eager class loading. Even if the sister-supported type checker verifies that the type of an instance is version compatible, that instance cannot be fully *trusted*. The instance may contain a version incompatible instance as a field value or return it as a result of a method execution. That is, the *untrusted* instance may *relay* an incompatible instance. Since an instance is type checked only when it is downcast, the types of the instance that may be relayed must also be type

checked at the same time. Therefore, the type checker verifies all the class types occurring in the class definition of that instance, such as parameter types[3], return types, and field types. It also recursively verifies the class types occurring in the definitions of those types. However, if this recursive type check is naively implemented, all the related classes would have to be eagerly loaded. This eager loading is practically unacceptable, since the advantages of the dynamic features of Java would be lost. The sister-supported type checker must be able to work with the scheme of lazy class loading. Note that the original class loading mechanism of Java is based on lazy class loading.

To examine version compatibility while enabling lazy class loading, the JVM maintains a set of *sister loader constraints*, which are dynamically updated when the sister-supported type checker works. If the type checker finds a class type that must be verified but has not been loaded yet, the JVM does not eagerly load that class; instead, it records a sister loader constraint. For example, if the type checker attempts to verify that a version of class C is version compatible with another version C', but C or C' has not been loaded yet, the JVM records as a constraint that C must be version compatible with C'. This constraint is later verified when C or C' is loaded. If the type checker detects that this constraint is not satisfied, it throws a `LinkageError`. While the type checker is verifying that constraint, if it finds another class type that must be verified but is not loaded, a new sister loader constraint is recorded. If the type checker finds a class type that must be verified and has been already loaded, it recursively verifies that class type at the same time. Note that every constraint is verified only once. The result of the verification is recorded to avoid further verification.

In summary, the JVM needs to maintain the invariant: *Each class type coexisting in the namespace satisfies all the sister loader constraints.* The invariant is maintained as follows:

Every time a new class joins a sister namespace, the JVM verifies whether that class type will violate an existing sister loader constraint.

If the class type being loaded violates an existing sister loader constraint, loading that class type fails since that class type is untrusted in the namespace. If there is no constraint referring to that class type, the JVM loads that class type, although that class type might be version incompatible. It is verified later when a new constraint referring to that class type is recorded.

Every time a new sister loader constraint is recorded, the JVM verifies whether that constraint is satisfied with the class types that have been already loaded.

If a class type that has already been loaded does not satisfy a newly recorded constraint, loading the class type that starts the type checking process producing the new constraint is untrusted and hence the loading is aborted. If any class types needed for verifying that constraint have not been loaded, the verification

[3] If a parameter type is not version compatible, an incompatible instance may be sent back without type checking to the namespace that has sent the instance of the class type including that parameter type.

is postponed until those classes are loaded. Otherwise, if all the class types needed for the verification have been loaded and the constraint is successfully verified, the constraint is removed from the record.

For efficient verification of constraints, we added an array of flags to `VM_Class`. Each flag indicates whether the version of the class type represented by a `VM_Class` object has been recursively type checked with another sister version. The flag is true only if the two versions of the class type are version compatible and if the type checker has verified that those two versions never *relay* a version incompatible instance. Since there might be multiple sisters, the `VM_Class` object holds an array of the flags, each of which indicates the result of the type check with each sister version. The JVM uses these flags for executing a recursive type check only once.

3.5 Schema Compatible Loading

Even if two versions of a class type satisfy the version compatibility, these instances may have schema incompatibility. This means that the layout of the internal type information blocks (TIBs) may not be identical between the two versions of the class type. The TIB holds fields and function pointers to a corresponding method body. The order of the TIB entries depends on the JVM or compilers; it does not depend on the order of the member declarations in a source file or a class file. Thus, even if two versions of a class type have version compatibility, the layout of the TIBs may not be identical.

The sister-supported class loader guarantees that layouts of the TIBs are identical between two versions of a class type if the class types are version compatible. Since the JVM uses a constant index into the TIB when it accesses a field or a method, the JVM cannot correctly execute the bytecode if the layouts of the TIBs are not identical between compatible versions of the class type. Therefore, when the class loader loads a younger version of a class type, the JVM constructs the TIB of that version of the class so that the layout of the TIB is identical to that of the TIB of an older version of the class. This loading process is called *schema compatible loading*. Note that this process is given up against the incompatible class type that has no binary compatibility with the older sister version of that class type. This result is employed by the JVM to quickly examine whether a class type is trusted or not.

In the JRVM, a TIB is constructed during the execution of the `resolve` method in `VM_Class`. The `resolve` method is invoked during the class resolution process by the `VM_ClassLoader` class, an instance of which represents a class loader. The `resolve` method has been extended to perform the schema compatible loading.

4 Discussion

4.1 Canceling JIT Compilations

Just-In-Time (JIT) compiled code sometimes needs to be canceled since a devirtualized method call does not correctly refer to a method declared in a sis-

ter version of the class type. Recent optimizing JIT compilers [15, 32] perform the devirtualization optimization that transforms not only a final and a static method but also a virtual method call to a static method. For a given virtual call, the compiler determines whether or not the call can be devirtualized by analyzing the current class hierarchy. If the method can be devirtualized and its code size is small enough, the compiler inlines the method. Therefore, if the type of an instance is converted to a sister version of that class type, the JVM would continue to invoke the original inlined code instead of the real method of that instance. This is because the JIT compiler does not consider sister namespaces; method bodies might be different among sister versions of the same class type.

To avoid this problem, the JIT compiler must cancel devirtualization when a new sister version of a class type is loaded. Fortunately, most optimizing JIT compilers have an efficient cancellation mechanism for dynamic class loading. Since a whole class hierarchy cannot be statically determined in Java, JIT compilers can dynamically replace [32] or rewrite [15] inlined code. This is performed when a new subclass is loaded and the subclass overrides a method that has not been overridden by the other subclasses. The JIT compiler that supports sister namespaces also cancels devirtualization when version compatibility is verified and a new sister version of a class type is available.

4.2 Eager Notifications of Version Incompatibility

Version incompatibility checks between sister versions of a class type may eagerly throw a cast error before any incompatible class types actually co-exist in one namespace. For example, if the type checker detects that a class type may relay a version incompatible instance, it throws a cast error. This eager notification strategy is similar to the loader constraint scheme [18]. The JVM prohibits different versions of a class type from even being loaded if the JVM encounters an operation that relays instances from one namespace to the other. If the JVM has already loaded these versions of the class, the JVM throws a link error. However, except for the loader constraint scheme, verification of compatibility and error notification are not performed at loading time but are done later by the linker, while the linker resolves the constant pool items (e.g., `NoSuchMethodError`, `IllegalAccessError`, `IncompatibleClasChangeError`). If a Java program includes binary incompatibility, it continues to run until it actually executes an illegal operation caused by that incompatibility. This lazy verification and notification are useful in practice.

However, we have adopted the eager strategy for avoiding performance penalties due to version compatibility checks. To delay notifications of incompatibility as long as the program continues to run without errors, a number of guard tests must be embedded into the incompatible class. In Java, once the JVM executes a method invocation or a field access, the operation is linked with the call site and replaced with efficient code that does not perform type checking anymore. Therefore, the guard tests must be embedded for verifying version compatibility after the version incompatibility is found. It requires the JIT compiler to recompile the code that refers the incompatible class type. Moreover, the guard tests

imply the non-negligible performance overhead mentioned in Section 2.3; thus, we do not delay the notifications of incompatibility.

5 Experimental Results

This section reports the results of our performance measurements. We performed all the experiments on the IBM Jikes Research Virtual Machine 2.3.2 with Linux kernel 2.4.25, which were running on a Pentium4 1.9GHz processor with 1GB memory. Both the Jikes RVM and our modified RVM were compiled to use the baseline compiler for building the boot image with the semi-space garbage collector.

Baseline Performance. To measure the baseline performance, we ran the SPECjvm98 [31] benchmarks on both our JVM and the unmodified JVM. The problem sizes of all the benchmarks were 100 (maximum). Table 1 lists the results. The numbers are the average execution time for 20 repetitions. The baseline overhead due to the sister namespace was negligible.

Cost of Loading Classes Into Sister Namespaces. We measured the time for loading classes with a plain class loader or a sister class loader. This experiment shows the performance penalty incurred by the sister class loader, which executes schema compatible loading and verifies the version compatibility between classes. We took nine application programs, listed in Table 2, to measure the total loading time. The loading process includes delegating to the parent class loader, searching for a class file in a specified classpath, and resolving, initializing, and instantiating that class type in the JVM. All loading processes are iterated 20 times. The results show that the performance penalty varied among those applications from around 14% to 67%. The penalties mostly depended on the number of declared methods and fields. Thus, the largest application showed the largest overhead.

Table 1. SPECjvm98 benchmark results on both our JVM and the unmodified JVM

Benchmark Program	*Jikes RVM (JRVM)*	*Sister-supported JRVM (SVM)*	*SVM /JRVM*
_201_compress	47.293 ms	46.218 ms	97.7%
_202_jess	40.258 ms	38.726 ms	96.2%
_205_raytrace	22.704 ms	23.404 ms	103.1%
_209_db	65.628 ms	67.075 ms	102.2%
_213_javac	54.698 ms	57.759 ms	105.6%
_222_mpegaudio	29.344 ms	29.210 ms	99.5%
_227_mtrt	25.812 ms	24.563 ms	95.2%
_228_jack	28.372 ms	28.047 ms	98.9%
Total	314.109 ms	315.002 ms	100.3%

Table 2. Total loading time using an ordinary class loader and a sister class loader. All classes are sequentially loaded by the `loadClass()` method

Program *(No. of classes)*	*Total loading time* *plain namespace*	*sister namespace*	*sister /plain*
JDOM (72 classes)	328 ms	382 ms	116.5%
Crimson (144 classes)	569 ms	696 ms	122.3%
jaxen (191 classes)	802 ms	919 ms	114.6%
dom4j (195 classes)	1,308 ms	1,487 ms	113.7%
SAXON (351 classes)	1,749 ms	2,113 ms	120.8%
XT (466 classes)	1,223 ms	1,422 ms	116.3%
XercesJ 1 (579 classes)	2,495 ms	3,046 ms	122.1%
XercesJ 2 (991 classes)	4,144 ms	6,177 ms	149.1%
XalanJ 2 (1,548 classes)	12,884 ms	15,290 ms	166.6%

JDOM [38] : A simple Java representation of an XML document, version 1.0
Crimson [34] : A Java XML parser included with JDK 1.4 and greater, version 1.1.3
jaxen [39] : An XPath engine, version 1.0.
dom4j [23] : The flexible xml framework for Java, version 1.5.2
SAXON [17] : An XSLT and XQuery processor, version 6.5.3
XT [19] : A fast, free implementation of XSLT in java, version 20020426a
XercesJ 1 : The Xerces Java Parser 1.4.4.
XercesJ 2 : The Xerces2 Java Parser 2.6.2.
XalanJ 2 [35] : An XSLT processor for transforming XML documents, version 2.6.0.

Cost of the `Checkcast` Instruction. Finally, we measured the execution time for type checking. The sister-supported type checking includes not only the ordinary `checkcast` operation but also the checking of trusted instances. We ran a program that executes the `checkcast` instruction for every class included in a given application, and then we measured the total execution time of all the `checkcast` instructions. Both experiment programs ran after all the classes had been loaded and then the version compatibility of all the classes was verified. We successively ran the program twice; the execution time of the second run indicates the execution time of `checkcast` after the version compatibility of all the possibly relayed classes is verified during the first run. Some of the first checks also make use of the results of previous verifications.

Table 3 lists the results. The results are the average of 10,000 iterations. The total execution time of the first checks was from about 10 to 40 times slower than the ordinary `checkcast` operation. This is because the sister-supported type checker traverses all possibly relayed class types. Since each application has a different number of possibly relayed classes, the relative performance varies for each application. On the other hand, the second checks included only around 160% overhead compared to the ordinary `checkcast` operation. Note that this overhead is incurred only when `checkcast` examines the type of an instance coming from another sister namespace. The overhead is negligible in regular cases.

We also compared the execution time of the type check with the time for marshalling and unmarshalling several XML data objects. Remember that the most harmless practice for the inter-component communication described in Section 2

Table 3. Total execution time of the type check by `checkcast`

Program *(No. of classes)*		`checkcast`	*Sister namespaces*		*Relative performance*	
			first	*second*	*first*	*second*
JDOM	(72 classes)	33.3 us	1,205.7 us	53.7 us	3,721%	261.3%
Crimson	(144 classes)	69.0 us	1,659.7 us	112.6 us	2,505%	263.1%
jaxen	(191 classes)	89.2 us	1,573.1 us	139.8 us	1,864%	256.7%
dom4j	(195 classes)	109.7 us	4,371.7 us	185.8 us	4,085%	269.3%
SAXON	(351 classes)	295.7 us	5,141.3 us	499.8 us	1,839%	269.0%
XT	(466 classes)	381.5 us	4,505.8 us	698.7 us	1,281%	283.1%
XercesJ 1	(579 classes)	644.4 us	7,824.3 us	1,041.3 us	1,314%	261.5%
XercesJ 2	(991 classes)	1,158.6 us	11,534.6 us	1,798.0 us	1,096%	255.1%
XalanJ 2	(1,548 classes)	1,650.6 us	22,627.8 us	2,696.2 us	1,471%	263.3%

is using a remote call, which passes an object by means of the call-by-value. This practice lets us avoid the problem of the version barrier, but it implies overhead due to the marshalling and unmarshalling for parameter passing. On the other hand, sister namespaces also let us avoid the problem, and it implies extra overhead only due to the type check. We measured the execution time for marshalling and unmarshalling DOM objects created by XercesJ 2.6.2 from 33 XML files taken from the Eclipse help system, which includes the Platform, Workbench, JDT, Plug-in and PDE document plug-ins. The overall file size was about 400KB. The measured execution time was 3 million times larger than the execution time of the ordinary `checkcast` operation. Of course, actual inter-component communication using a remote call would spend much more time for the network data transportations. Therefore, this result shows the sister namespace is a significantly faster solution compared to the solution of passing objects by a remote call.

6 Related Work

In the object database community, several schema evolution techniques such as schema or class versioning [5, 30] have been studied. These techniques allow multiple co-existing versions of a schema or a class. Instances are evolved when passing through the version barrier into the modified application or other applications. Using such evolvable object databases is a workable alternative for component-based applications. However, our work concentrates on programming environments, especially where runtime overheads due to schema evolution must be severely minimized.

There have been other research activities tackling the version barrier problem in the programming language and environment community. Most of the previous research regarded the version barrier as a temporal boundary between *old* and *new* components and thus focused on dynamic software updates or evolution. That is, multiple versions of the same class type could not simultaneously co-exist in the running program. Therefore, our problems were not directly ad-

dressed. In this research area, the main topic is which existing object should be adapted to an updated version of the class type and how and when. For example, the work on the hotswapping of classes falls into this category. Malabarba *et al.* [20] modified the JVM to make a class reloadable at runtime so that all existing objects can be updated incrementally. The JPDA (Java Platform Debugger Architecture) [33] and the `java.lang.instrument` package of the Java2 SDK5.0 provide the restricted hotswap functionality, whereby existing instances can be considered as the new version of the class type without being updated. Hjalmtysson and Gray [13] implemented dynamic classes in C++ by using templates. Users can selectively update some but not all objects with the help of wrapper (or proxy) classes and methods. Hnětynka *et al.* [14] proposed the renaming approach using a bytecode manipulation tool. A class loader using this renaming approach allows the reloading of a class, although it renames that class.

Our work mainly focuses on the spatial version barriers among multiple components. An older version of a class type remains after a new version is loaded. Dynamic type changes such as predicate classes [4], reclassifying objects [6], and wide classes [28] may allow relaxing of the spatial version barrier, since multiple class members can be implicitly merged by the explicit composition operation. Type-based hotswapping proposed by Duggan *et al.* [8] is similar to our work but classified into the same category as the above systems. The .NET counterparts of the Java class loaders are *application domains*, which are used to load and execute assemblies and can run in a single process. However, they adopt the call-by-value semantics on inter-component communication between application domains using the .NET Remoting Framework. Of course, dynamic typing languages, such as CLOS, Self, and Smalltalk, provide more flexible mechanisms for allowing types to be changed at runtime. However, our challenge is relaxing only the version barrier in the strictly typing object-oriented world with negligible performance penalties. Our contribution is that we have provided a simple mechanism for relaxing the version barrier, which has been confusing Java programmers because of the complicated semantics.

7 Conclusion

This paper presented the design and implementation of loosely-separated sister namespaces in Java. Combining multiple namespaces as sister namespaces can relax the version barrier between them. It thereby allows an instance to be assigned to a different version of that class type in that sister namespace. This mechanism was implemented on the IBM Jikes RVM for evaluation of the performance overhead. Our experiment showed that, once an instance passes into the sister namespace across the version barrier, all instances of that class type can go back and forth between the sister namespaces with significantly low performance overhead. Our experiment also demonstrated that the execution performance has only negligible overhead unless an instance is passed across the version barrier.

We plan to develop a dynamic AOP (Aspect-Oriented Programming) system based on sister namespaces. We have developed a Java-based dynamic AOP sys-

tem called Wool [27]. It allows weaving aspects with a program at runtime by using the hotswap mechanism of the standard debugger interface called JPDA (Java Platform Debugger Architecture) [33]. The version compatible changes shown in this paper are almost the same as that supported by the JPDA. Therefore, using sister namespaces will make our dynamic AOP system simpler and more efficient while keeping the equivalent flexibility.

Currently, our primary focus for future work is the formal proof of the type safety on the sister namespaces. We will also examine this issue with respect to the Java security architecture [10].

Acknowledgement

We would like to express our deep gratitude to the anonymous reviewers. Hidehiko Masuhara gave helpful comments on an early draft of this paper. We also thank Romain Lenglet for his great efforts to fix numerous English problem in this paper. This research was partly supported by the CREST program of Japan Science and Technology Corp.

References

1. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeno virtual machine. IBM System Journal **39** (2000) 211–238
2. Alpern, B., Cocchi, A., Fink, S.J., Grove, D., Lieber, D.: Efficient implementation of java interfaces: Invokeinterface considered harmless. In: Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001). Number 11 in SIGPLAN Notices, vol.36, Tampa, Florida, USA, ACM (2001) 108–124
3. Alpern, B., Cocchi, A., Grove, D.: Dynamic type checking in jalapeño. In: Java Virtual Machine Research and Technology Symposium. (2001)
4. Chambers, C.: Predicate classes. In: ECOOP'93 - Object-Oriented Programming, 7th European Conference. Volume 707 of Lecture Notes in Computer Science., Kaiserslautern, Germany, Springer-Verlag (1993) 268–296
5. Clamen, S.M.: Type evolution and instance adaptation. Technical Report CMU-CS-92–133, Carnegie Mellon University School of Computer Science, Pittsburgh, PA (1992)
6. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: Fickle : Dynamic object re-classification. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference. Volume 2072 of Lecture Notes in Computer Science., Budapest, Hungary, Springer (2001) 130–149
7. Drossopoulou, S., Wragg, D., Eisenbach, S.: What is java binary compatibility? In: Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada (1998) 341–361

8. Duggan, D.: Type-based hot swapping of running modules. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01). Volume 10 of SIGPLAN Notices 36., Florence, Italy, ACM (2001) 62–73
9. Englander, R.: Developing Java Bean. O'Reilly and Associates, Inc. (1997)
10. Gong, L., Ellison, G., Dageforde, M.: Inside Java2^{TM} Platform Security: Architecture, API Design, and Implementation 2nd Edition. Addison-Wesley, Boston, Mass. (2003)
11. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass. (2000)
12. Halsted, K.L., Roberts, J.H.J.: Eclipse help system: an open source user assistance offering. In: Proceedings of the 20st annual international conference on Documentation, SIGDOC 2002, Toronto, Ontario, Canada, ACM (2002) 49–59
13. Hjálmtyśson, G., Gray, R.: Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In: Proceedings of the USENIX Annual Technical Conference, New Orleans, Louisiana, USENIX (1998)
14. Hnětynka, P., Tůma, P.: Fighting class name clashes in java component systems. In: Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003. Volume 2789 of Lecture Notes in Computer Science., Klagenfurt, Austria, Springer (2003) 106–109
15. Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., Nakatani, T.: A study of devirtualization techniques for a java$^{\mathrm{tm}}$ just-in-time compiler. In: Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000). Number 10 in SIGPLAN Notices, vol.35, Minneapolis, Minnesota, USA, ACM (2001) 294–310
16. JUnit FAQ: Why do I get an error (ClassCastException or LinkageError) using the GUI TestRunners?, available at: http://junit.sourceforge.net/doc/faq/faq.htm. (2002)
17. Kay, M.: SAXON The XSLT and XQuery Processor, available at: http://saxon.sourceforge.net/. (2001)
18. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Proceedings of OOPSLA'98, Proceedings of the 1998 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
19. Lindsey, B.: XT, available at: http://www.blnz.com/xt/. (2002)
20. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime Support for Type-Safe Dynamic Java Classes. In: Proceedings of ECOOP 2000 - Object-Oriented Programming, 14th European Conference. Volume 1850 of Lecture Notes in Computer Science., Springer-Verlag (2000) 337–361
21. 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
22. Matena, V., Stearns, B.: Applying Enterprise JavaBeans^{TM}: Component-Based Development for the J2EE^{TM} Platform. Pearson Education (2001)
23. Metastaff, Ltd.: dom4j: the flexible xml framework for Java, available at: http://www.dom4j.org/. (2001)
24. Nathan, A.: .NET and COM: The Complete Interoperability Guide. Sams (2002)
25. OMG: The Common Object Request Broker: Architecture and Specification. Revision 2.0. OMG Document (1995)

26. Saraswat, V.: Java is not type-safe. (1997)
27. 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
28. Serrano, M.: Wide classes. In: ECCOP'99 - Object-Oriented Programming, 13th European Conference. Volume 1628 of Lecture Notes in Computer Science., Lisbon, Portugal, Springer-Verlag (1999) 391–415
29. Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P.: The Java Developer's Guide to Eclipse. Addison-Wesley (2003)
30. Skarra, A.H., Zdonik, S.B.: The management of changing types in an object-oriented database. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86). Volume 11 of SIGPLAN Notices 21., Portland, Oregon (1986) 483–495
31. Spec - The Standard Performance Evaluation Corporation: SPECjvm98. (1998)
32. Sun Microsystems: The Java HotSpot Performance Engine Architecture, available at: http://java.sun.com/products/hotspot/whitepaper.html. (1999)
33. Sun Microsystems: Java^{TM} Platform Debugger Architectuer, available at: http://java.sun.com/j2se/1.4/docs/guide/jpda. (2001)
34. The Apache XML Project: Crimson Java Parser, available at: http://xml.apache.org/crimson. (2000)
35. The Apache XML Project: Xalan Java XSLT Processor, available at: http://xml.apache.org/xalan-j. (2002)
36. The Apache XML Project: Xerces2 Java Parser, available at: http://xml.apache.org/xerces2-j. (2002)
37. The Eclipse Foundation: Eclipse.org, homepage : http://www.eclipse.org/. (2001)
38. The JDOM^{TM} Projec: JDOM, available at: http://www.jdom.org/. (2000)
39. The Werken Company: jaxen: universal java xpath engine, available at: http://jaxen.org/. (2001)