PAAL, Stefan
KAMMÜLLER, Reiner
FREISLEBEN, Bernd

Java Class Separation for Multi-Application Hosting

published on netzspannung.org:
http://netzspannung.org/about/mars/projects/pdf/awake-2002-2-en.pdf
14 March 2005

First published: Proceedings of the 3rd Conference on Internet Computing (IC 2002). Las Vegas, USA. CSREA 2002. pp. 259-266.

Java Class Separation for Multi-Application Hosting

Stefan Paal
Fraunhofer Institute for Media Communication
Schloss Birlinghoven
D-53754 St. Augustin, Germany

Reiner Kammüller, Bernd Freisleben
Department of Electrical Engineering
and Computer Science
University of Siegen
Hölderlinstr. 3, D-57068 Siegen, Germany

Abstract. *Java applications are usually executed within a Java Virtual Machine (JVM), which is part of the Java Runtime Environment (JRE). In this scenario, hosting more than one application at the same time within a JVM is not originally supported, and customizable mechanisms to manage different application classes and related byte codes with the same class name concurrently are still lacking. In this paper, we present a novel approach to Java class separation used to transform the native JVM into a multi-application environment. Our proposal is based on introducing so called class spaces, which enable class selection using properties other than the class name and which arrange also loaded classes within a JVM in a customizable manner. The feasibility of the approach is demonstrated by presenting a middleware framework in which applications needing the same classes in different versions are configured dynamically and hosted concurrently.*

Keywords. Java Class Separation, Java Virtual Machine, Java Class Loading, Application Hosting, Internet Middleware

1. Introduction

The execution of a Java application requires the so called *Java Runtime Environment (JRE)* and its *Java Virtual Machine (JVM)*. However, before a Java application can be started by the JVM, the related byte code for the application classes must be located, loaded and resolved using the *dynamic class loading* approach of Java [1]. For this purpose, the native JVM delegates the class loading to the so called *system class loader*, which locates the byte code using an environment variable CLASSPATH and the *fully-qualified class name (FQCN)* of the required classes, which is composed of the package name and the class name itself [2,3]. In addition, there may be further class loaders, which are user-defined and make their own arrangements of how the byte code is located and loaded. But common to all class loaders is the fact that each can load the same class only once during its lifetime, whereby two classes are assumed to be equal when their FQCN are equal [4].

This is usually suitable since the native JVM and its system class loader have been designed to host and manage only a single application at the same time and which is executed by the underlying operating system within a single process. For this reason, further Java applications have to be started in additional JVM. Though this is well-suited for stand-alone applications running on a local machine with one user interface, it causes problems when a multi-application environment like an application server should be configured to run more than one application, possibly requiring the same classes in different variants concurrently and requesting to share data across applications within a single JVM. As mentioned above, the problems are caused by the fact that a class loader can load a class only once during its lifetime, since all loaded classes are put within so called *namespaces*, each associated with a class loader, where no two classes with the same FQCN can co-exist.

Due to this fact, there are implementations using more than one class loader, each managing its own namespace and able to load the same class concurrently to another class loader. That way, they avoid conflicts with classes having the same FQCN and can host applications in different variants concurrently. They are even able to reload the application code in a new namespace dynamically when it has been changed in the byte code repository. Although this resolves the multi-application problem, it raises another problem, since it separates all applications completely and makes it difficult to share data among them, as explained in the following.

Each class loader has usually one parent class loader, defining a *parent-delegation model* and forming a class loader tree with the system class loader at the root. The class loaders are configured in such a manner that they delegate class loading requests at first to their parent, and only if the parent could not load the requested class, the current class loader tries to load the class. That way, a class is loaded only once in the chain from the current class loader up to the root, usually the system class loader. All classes and objects in a certain class loader can "see" and use only the loaded classes from the parent class loader, but not from the child class loader or other class loaders, located elsewhere. Due to this fact, objects in two class loaders can only share data and collaborate if they "see" the associated class, or in other words, if their class loaders have the same parents.

Within this context, there are two kinds of class loader associated with a loaded class instance, so called *initiating class loaders* and so called *defining class loaders* [2]. While each loaded class instance has only one defining class loader, that is actually loading the related byte code and finally defining the class instance. There might be several associated initiating class loaders, simply by the fact, that they are in the chain between the requesting class loader and the defining class loader.

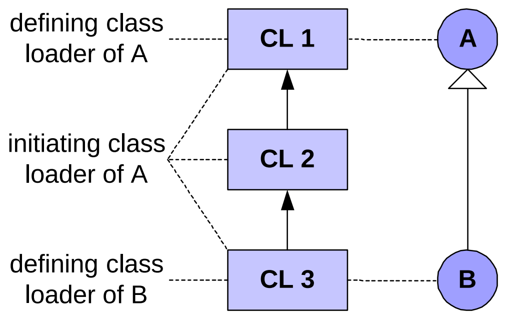
defining class loader of A
initiating class loader of A
defining class loader of B
CL 1
CL 2
CL 3
A
B

Figure 1: Initiating and defining class loaders

An illustrating example is shown in fig. 1, there are three class loader CL1, CL2 and CL3. The class loader CL1 is able to load class A and CL3 can load class B, which is inherited from A. When CL3 is requested to load B, it must first load its super class A. And since it is not able to load A, it delegates the request up to its parent CL2, which in turn delegates the request to CL1. Finally, CL1 loads A and becomes the defining class loader of A. The other participating class loaders become initiating class loaders of A. The namespace of a class loader, already mentioned above, can now be defined more exactly, it is simply a list of classes, for which the class loader has served as an initiating class loader [2].

In this paper, we present a novel approach to configure the class layout within a JVM and show how this could be used to manage the separation of Java classes and hence transform the native JVM into a multi-application environment. For this purpose, we introduce so called *class spaces*, which manage namespaces and the loaded classes in a customizable manner.

The paper is organized as follows. Section 2 briefly describes the features of existing Java application environments and its class loading approaches with respect to application hosting and class separation. In section 3, we present the features of a multi-application framework and our approach in detail. In section 4, the use of our approach within a middleware framework is illustrated. Section 5 concludes the paper and outlines areas of future work.

2. Java Application Environment

There are various approaches towards application environments running on top of a native JRE, which claim to ease and customize the deployment, loading and running of applications and the related byte code like *Java Web Start* [5]. Most of them are not available stand-alone, but part of another framework like *Jakarta Tomcat* [6]. In addition, some approaches put various constraints on the environment and the Java application to be hosted like *Enterprise Java Beans (EJB)* [7], thus they are not generally suitable for legacy Java applications. Nevertheless, each approach represents a certain kind of application framework with particular features, especially application hosting and class loading, which are discussed in the following.

2.1 Application Hosting

Multi-application environments provide an environment for hosting different applications concurrently, sharing commonly used resources like code libraries, memory space, network connections or access to databases within a single JVM, improving greatly the utilization of these resources [8,9] in contrast to the usage of a JVM per application. This is facilitated in a particular manner, since applications do not need to cross the borders of their JVM for that purpose and can collaborate directly without using RMI, CORBA or similar solutions [10,11]. Particularly in multi-user environments like Internet-enabled application servers [12,13,14], when many applications share the same resources, this feature may be quite important.

Another objective in this context is the dynamic loading and unloading of applications. For example, it is usually not possible to shutdown a JVM, which hosts several web services due to the reload of one single service.

For that purpose and to ensure the proper execution of applications in the presence of other hosted applications, each application is placed in a new namespace in which an application can load its own classes separately from classes loaded by other applications, except for the classes loaded by the framework. This is ensured by the fact that each class loader delegates class loading requests at first up to their parents. That way, parent class loaders get always the first chance to load the class before its child and all applications can share data using classes from the application framework.

In summary, the major tasks of existing multi-applications environments are the concurrent ability of shielding resources of applications from each other but also to enable applications to be reloaded and share common code and data. This is more than ever important for web application environments.

2.2 Class Loading

An important feature of a Java application environment is the ability to load classes dynamically during runtime and extend the application with additional classes. In this scenario, there are several, slightly different approaches based on the common class loader approach, which are detailed in the following.

Customized Class Location

The origin class loader of Java has a quite limited way to locate and load classes dynamically from sources other than the file system. The obvious way to extend this capability is the provision of user-defined class loaders which can access class files over the network, or access repositories in which the byte code is stored. In this scenario, the user-defined class loader usually focuses on the task of how the requested classes can be *loaded*. But some of them also handle the problem of *locating* the requested class, process the loaded byte code and use its own mechanism to resolve the location of the class [14].

Resolving Naming Conflicts

A Java naming space can not contain two different classes with the same FQCN. Therefore, some application frameworks use a separate namespace to hold the classes of each application, whereby each namespace is managed by its own classloader, having a parent class loader and building a tree-like structure, with the system class loader at the root. And since class loaders always delegate the class loading requests at first to their parent, commonly used classes are placed by the parent class loader in their namespaces. That way, each application can load its own class, independently of other hosted applications, but nevertheless it is able to share data with other applications, using the classes loaded by the parent class loaders [3].

Managing Life-Cycle

An important issue of application frameworks is the loading and executing of applications. For this purpose, the framework must be able to load and unload the byte code of a class. But the native JRE does not support unloading an already loaded class. Thus, application frameworks use different namespaces to reload a class; as a result, they resign to reload the byte code, but instantiate a new namespace, where the class is "reloaded" [2].

In summary, there are different approaches of how applications can be hosted and classes are loaded. Although this is often done in a pre-defined way, and the application is neither able to configure the organization of the namespaces nor to define where the classes are placed among the namespaces after they are loaded. However, common to all approaches is that they do not deal with how the byte code of the requested classes are *selected* and *organized* in namespaces with respect to the problems mentioned at the beginning of this paper. In the following, we will show how these problems can be addressed and solved using our approach of customizable Java class separation.

3. Customizable Java Class Separation

Application frameworks use the class loading approach of Java with different goals, and many implementations are heavily based on using namespaces. What is still missing is the possibility of defining from which classes an application is composed, where they can be found and how they are arranged in namespaces, so that applications using the same classes in different variants can coexist and in addition are able to collaborate. Given an application framework, this is essential for building an environment where applications can load classes which are not shared with anyone else nor can be shared due to name conflicts, but also can share classes with other applications or the framework itself. These issues are the tasks of a multi-application environment. The required features are presented in detail in the following and realized by the proposed approach, presented afterwards.

3.1 Features

There are mainly two parties which have to deal with the features of an application framework. The first one are *application developers*, which want to use the framework to implement new applications on top of it, and the second one are *administrators*, who install and configure the framework and related applications on a certain computer.

Transparent Handling With Respect To Existing Application Code

An application framework which forces the developer to change existing and moreover running application code to be executable within the framework, will of course not be accepted widely. Therefore, a keystone of an application framework is the transparent handling of new features, leaving the application developers to deal with their actual tasks. For example, approaches which introduce some kind of class name mangling for hosting variants of the same class would break legacy application code.

Dynamic Configuration Of Shared And Separated Classes By Classspace Aware Applications

Another kind of application developers wants their application to use special features of the framework. Particularly with regard to loading concurrent classes, the configuration of shared and separated classes should be customizable by the application. This enables the dynamic extension with additional functionalities and plugins during runtime, and eases resolving occurring conflicts.

Static Configuration of Class Layout Depending On The Application Requirements

When a Java application should be started, the JVM has to load the required classes before. But actually, the JVM loads classes only on demand, and when a class is not found later, the JVM is usually terminated, since there is nothing to solve the problem. In contrast, a multi-application environment should not be terminated due to a single wrongly configured application. Instead, the application environment should determine the required classes before the application starts and deny the execution if classes are missing.

Customizable Resource Registration During Runtime

The native class loader concept of Java expects the specification of class repositories like JAR-files and file path in the environment variable CLASSPATH. Though this is fairly well-suited for standalone applications which are managed by a single developer and administrator, it limits the way of how applications can be composed when adding dynamically new resources like classes or resource bundles. Thus, the application environment should not only support the registration of classes by the administrator before starting the framework, but also by the application afterwards during runtime.

3.2 Architectural Approach

In the following, we introduce our architectural approach of so called *class spaces*, that specifies where to place loaded classes and hence defines the *layout of application code* within an application framework.

An application framework does not only have the task to load and start the application, but also to host the application and its classes in memory.

Generally, all classes of an application are placed within a single namespace. But even if there is no conflict with classes using the same class name, some applications ask the framework to reload or exchange some portion of the application code like plugins. For this purpose, additional namespaces are created in which the new classes can be placed. The former namespaces exists further, but are no longer used. This is done rather proprietary, and there is no way to define where loaded classes have to be put, in the parent namespace or in the current namespace. However, particularly this feature is required when two applications want to share data as illustrated above. Thus, we wrap so called *class spaces* around namespaces. They are able to monitor which class should be loaded and can delegate the class loading, customizable through the application framework, to other class spaces.

In fig. 2, there is a class space SCS and two further class spaces UCS1 and UCS2 as children of SCS. All objects created by classes in UCS1 will be associated with the class loader of UCS1. In the case one of these objects creates another object from a new, yet not loaded class, the class loader is asked to load the byte code of the newly requested class. For this purpose, it determines whether the related class space, in this case UCS1, is configured to load the class. If not, it delegates the request up to its parent class space SCS and its class loader. This is directly opposed to the original behavior of chained class loaders, where all class loading requests are at first directed to the parent class loader. But with this reversal, the class spaces can be configured to hold shared classes in the parent class spaces and unshared classes in the child class spaces.

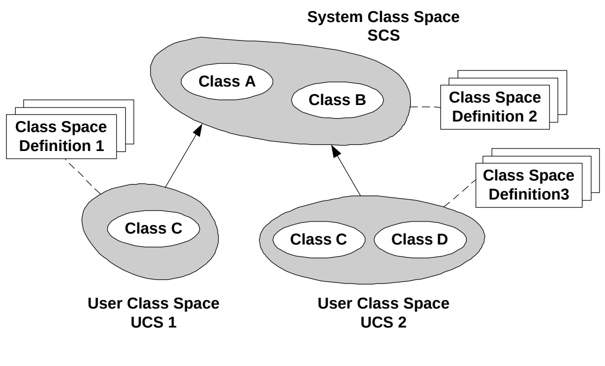

Figure 2: Class Spaces

In the example, objects in UCS1 and UCS2 can be built from classes with the same name C, but could rely on different class implementations. On the other hand, they can share data and collaborate using classes and objects located in SCS.

In addition, the configuration of a class space and the information which classes can be loaded is combined with the information where to find the related byte code. A special issue at this point is the question of how the system should resolve registration conflicts between parent and child class space, if both want to register the same class for loading. The answer is quite simple: all registrations must be checked against the configuration of the chained class spaces, and if there is already a chained class space which handles the class, the registration is denied. This is manageable up to a certain degree if the class spaces are configured properly by the application framework. But certainly, there are constellations, which can not always be resolved, i.e. applications, which want to collaborate using the same class in different variants. In this case, however, some kind of adaptation is needed anyway.

Finally, as a result, the layout of application code, shared and separated classes is defined without modifying existing application code. The configuration process of class spaces and the distribution of newly loaded classes are completely transparent to the application. Of course, this is only guaranteed as long as the application does not use a self-defined class loader. In this case, it would be possible, that the foreign class loader retrieves classes without knowledge of the parent class loaders and their associated class spaces. Thus, they would not be able to check future resource registrations whether the related classes are already handled by one of the child class loader.

3.3 Realization

After presenting the basic idea of our approach, the implementation and its characteristics are illustrated in this subsection.

As introduced above, each class space encapsulates a class loader and its related namespace. They can be configured, which classes they are allowed to load and which requests should be delegated to the parent class space. The configuration of class spaces can be done statically by using an XML configuration file given by an administrator, or dynamically by the application implemented by a developer. Of course, the configuration of the class space can be modified during runtime as long as no conflicts occur as mentioned above.

An example of a dynamic configuration is shown in fig. 3. At first, a new class space *app* is created with the class space *system* as its parent. Then, a new resource is registered in the class space, specifying that all classes whose names begin with *org.apache.xerces* will be found in the given JAR-file, and the class space should be able to load this class. By the way, the dots in the class name are masked with slashes, since a dot is a special letter in regular expressions, which are used to specify the matching pattern.

```
ClassSpace app = null;

app = ClassManager.getClassManager().
createClassSpace("application",
"system");

app.registerResource
("org/apache/xerces/.*",
"/usr/lib/xerces-1.4.4/xerces.jar");

Class parser = app.getClassLoader().
loadClass("org.apache.xerces.parsers.
DOMParser");
```

Figure 3: Using class spaces

At the end of the example, the class space is requested to load a class, or with other words, the class is injected into the class space. All subsequently class loading requests initiated by this class, respectively its objects, will also be handled by the class loader of the associated class space *application*.

At this point, it should be stressed that the class space does not load any class without a request. The configuration does only specify which classes can be loaded by the class space and where to find the related byte code.

Besides the illustrated dynamic configuration of class spaces, which is primarily used by developers, there is also a static configuration file. It is read when the system is started and enables administrators to specify the class space of each application separately. An excerpt of an example is shown in fig. 4. It defines a class space called *application*, with the class space *system* as its parent. The class space *application* is configured to host classes from the JAR-files specified in *path* and the following *resource* entries.

```
<space name="application"
parent="system">

 <jar path="/usr/sdk/servlet.jar">
  <resource name="javax/servlet/.*" />
 </jar>

 <jar path="/usr/sdk/xalan.jar
  <resource name="apache/xalan/.*" />
 </jar>

</space>
```

Figure 4: Example of a class space definition file

The class space *application* is not automatically created, but whenever the system opens a class space named *application*, the configuration file is read, and the class space is configured respectively. Afterwards, additional resources can be registered dynamically provided that they do not cause any conflicts with existing registrations.

4. Application of the Approach

The presented approach has been developed as part of the middleware framework *ODIN* and has proven its suitability and correctness in several ongoing research projects, which are based upon this framework like *CAT* and its associated Internet platform *netzspannung.org* [16,17,18]. In this context, the major goal of ODIN is the provision of an open programming and runtime environment for hosting dynamically composed and concurrently executed applications like Web Services or Web Agents. In addition, applications often collaborate and access commonly shared resources like databases, network connections or session informations. Consequently, ODIN uses a single JVM to host multi applications and eases the collaboration within the same node by enabling object interconnection across applications without using CORBA [11] or RMI [19]. Though this reduces the usage of runtime resources, ODIN has also to ensure, that applications, which do not want to interact can be shielded against each other. For achieving these goals, the Java implementation of ODIN uses class spaces invented by the presented approach. Their application is briefly presented in the following.

As illustrated in fig. 5, ODIN creates a *system class space* and a *framework class space*, which contain commonly shared byte code like core Java classes and classes from ODIN itself. In this example, there are three concurrently hosted applications *App1-3*, each put in a separate *application class space*. Furthermore, there are two libraries *Lib1* and *Lib2*, shared by the applications. They can easily interact by using classes from *Lib1* and *Lib 2*, and ODIN or Java core classes, of course. In turn, application related classes placed in *App1-3* are shielded and do not influence each other.

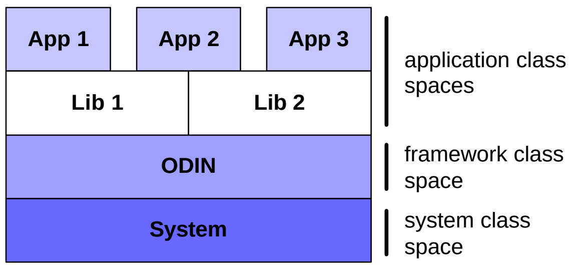

Figure 5: Class Space Organization in ODIN

While this example is quite simple, it might get much more complex when there are more class space levels and libraries, which depend themselves again on other libraries. An actual example is the latest JRE 1.4.0 from Sun, which comes with several classes from the *Apache Software Foundation* for handling XML files. Some applications like *Jakarta Tomcat* [6] or *Apache Cocoon* [20] need exactly these classes, however in different variants. Unfortunately, the JRE bundles them with other system and core Java classes within the same JAR file, thus they can not be simply excluded or replaced. As a result, there are some workarounds available to solve this problem, e.g. by arranging and installing the JAR files in a particular way or even by extracting the classes from the runtime library of the JRE. Though this is feasible for that certain scenario, it remains a workaround, which have to be checked and adapted for each application. In contrast to that, using the presented approach, this problem has been solved quickly by ODIN, configuring the *system class space* not to load the XML related classes from the JDK, but from another class space, configured to load the right classes. Moreover, this can be done separately for each hosted application and does not require any modification of the standard Java runtime installation.

Another guiding principle in ODIN is the separation of concerns, resulting in a runtime system, where our approach is used to encapsulate roles within class spaces like in the example above. The *system class space* is formed by the classes of the JRE and the *framework class space* groups classes of ODIN. Both together contain the *middleware kernel* of ODIN, which is able to host dynamically composed applications and to control their class loading. That way, ODIN uses the presented approach to control which applications are allowed to use certain classes, dependent on their current role in the system.

5. Conclusions

In this paper, we have outlined the request for multi-application hosting and discussed related problems of existing solutions regarding application class layout and class selection. We have presented a new customizable approach to host different implementations of the same Java class concurrently and how to manage, shield and share these classes within a single JVM, solving the regarded problems. In this context, we have introduced so called class spaces, which are based on the managed separation of loaded classes and objects within the JVM. After illustrating the implementation issues, we have presented the application and suitability of the approach in the middleware framework ODIN and its capabilities for hosting dynamically composed and concurrently executed Java applications.

There are several issues for future work. For example, the resource configuration of class spaces is rather simple and tedious. We are currently investigating how this could be done automatically and more comfortably, e.g. by analyzing package files. Another topic arises from the fact that the layout of the class space has to be defined by the application installer. It would be much more efficient to have a system that adjusts the class spaces according to the requirements of recently requested application while considering already hosted applications. Finally, the presented approach has been only implemented and evaluated for the *Java Virtual Machine*. But it would be interesting, whether it is adaptable to other virtual-machine based environments like Microsoft .NET [21], since the covered problems seem to be similar.

6. Acknowledgements

The presented approach has been evaluated and used in the implementation of the Internet platform *netzspannung.org* [17]. The related project CAT [16] is funded by the German Federal Ministry for Education and Research and is conducted by the research group MARS from Fraunhofer Institute for Media Communication, St. Augustin in cooperation with the University of Siegen, Germany. Special thanks go to Monika Fleischmann, Wolfgang Strauss and Jasminko Novak.

References

[1] Liang, S., Bracha, G. Dynamic Class Loading In The Java Virtual Machine. Proc. of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM 1998. pp. 36-44.

[2] Venners, B. Inside The Java 2 Virtual Machine. McGraw-Hill. 1999.

[3] Eckel, B. Thinking in Java. Prentice Hall. 2000.

[4] Lindholm, T., Yellin, F. The Java Virtual Machine Specification. Addison-Wesley. 1999.

[5] Java Web Start http://java.sun.com/products/javawebstart

[6] Jakarta Tomcat - Servlet Engine http://jakarta.apache.org/tomcat/index.html

[7] Monson-Haefel, R. Enterprise Java Beans. O'Reilly & Associates. 2000.

[8] Fayad, M. E., Schmidt, D. C., Johnson, R. E. Implementing Application Frameworks: Object-Oriented Frameworks at Work. John Wiley & Sons. 1999.

[9] Lewis, T. Object Oriented Application Frameworks. Manning Publications Co. 1995.

[10] Orfali, R., Harkey, D. Client/Server Programming with Java and Corba. John Wiley & Sons, Inc. 1998.

[11] Marvic, R., Merle, P., Geib, J.-M. Towards a Dynamic CORBA Component Platform. Proc. of International Symposium on Distributed Objects and Applications. 2000. pp. 305-314.

[12] Latteier, A. Bobo and Principia: An Object-Based Web Application Platform. WebTechniques. February 1999.

[13] Sun Open Network Environment (ONE) http://www.sun.com/sunone

[14] Apache Server Framework Avalon http://jakarta.apache.org/avalon/framework/index.html

[15] Gong, L. Secure Java Class Loading. IEEE Internet Computing, Vol. 2, Nr. 6, pp. 56-61. 1998.

[16] Open Distributed Network Environment http://odin.informatik.uni-siegen.de

[17] Fleischmann, M., Strauss, W. Communication of Art and Technology (CAT). IMK/MARS, GMD St. Augustin. http://imk.gmd.de/images/mars/files/Band_1_download.pdf

[18] netzspannung.org, Communication Platform for Digital Art and Media Culture. http://netzspannung.org

[19] Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi

[20] Apache Software Foundation. Apache Cocoon. 2001. http://xml.apache.org/cocoon

[21] Farley, J. Microsoft .NET vs J2EE: How do they stack up. O'Reilly. 2001.