Java Class Deployment Using Class Collections

Stefan Paal[1], Reiner Kammüller[2], Bernd Freisleben[3]

[1] Fraunhofer Institute for Media Communication
Schloss Birlinghoven, D-53754 St. Augustin, Germany
stefan.paal@imk.fraunhofer.de
[2] Department of Electrical Engineering and Computer Science, University of Siegen
Hölderlinstr. 3, D-57068 Siegen, Germany
kammueller@pd.et-inf.uni-siegen.de
[3] Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Strasse, D-35032 Marburg, Germany
freisleb@informatik.uni-marburg.de

Abstract. Java applications are composed of classes which are usually grouped and deployed using Java Archives. When an application is started, the hosting *Java Virtual Machine (JVM)* obtains the required classes one by one as they are needed from these archives. For this purpose, the JVM locates the related byte code by taking the names of the class and its package and evaluating the environment setting CLASSPATH. This works well as long as there is only one byte code matching the given class name, but it is not feasible when the byte code has to be selected among several classes with the same class name using properties such as version numbers or manufacturers. In this paper, we introduce so called *Java Class Collections* which enable the logical grouping of classes separately from their physical deployment within Java Archives and allow tagging them with supplementary properties used to select the right byte code. We illustrate the application of our approach for multi-application hosting and in remotely composable application systems.

1 Introduction

When Java source code is compiled, the Java compiler generates related byte code which is named after the corresponding class names and is placed into directories organized by the used package declarations. Consequently, the byte code of each class can be located using the class name and the package wherein the class has been declared. However, for deploying the application or class library, the byte code files are usually packaged into so called *Java Archives*. These organize the contained byte code in a similar manner but consist of a single file which can be deployed easily.

When a Java application is started by the *Java Virtual Machine (JVM)*, the related byte code of the application classes must be located, loaded and resolved. Resolution means that loaded classes may refer to other classes, which must also be loaded before the referring class can be used [1]. Hence, the native JVM delegates the class loading process 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 (composed of the package name and the class name itself) [2,3].

The *system class loader* is able to load classes deployed into directories of the local file system or stored in Java Archives. In addition, there may be other class loaders, which are user-defined and are free in the way the byte code is located and loaded [4].

Obviously, this kind of application deployment eases class localization and loading, but on the other hand, this feature makes it nearly impossible to select byte code depending on other properties than the class name, such as implementation version, multi-threaded or debug variant of the class etc. Of course, there are applications which instantiate a user-defined class loader and introduce a new loading mechanism, but they usually change the way classes are *loaded* and sometimes also how they are *located*, but not how they are *selected*. They still use only the package and class name. As a result, it is not possible to direct which classes should actually be used when composing a certain application. Instead, the JVM requests the needed classes using their class name and package, while it is assumed that the actually used class loader finds the appropriate byte code [2]. However, this is not always feasible, e.g. when an application is composed of classes from various Java Archives which can not be modified but may contain several classes having the same class name. In this case, the ordinary class loader will always select the classes in the sequence as they are found, but this will usually not fit to the requirements of the application.

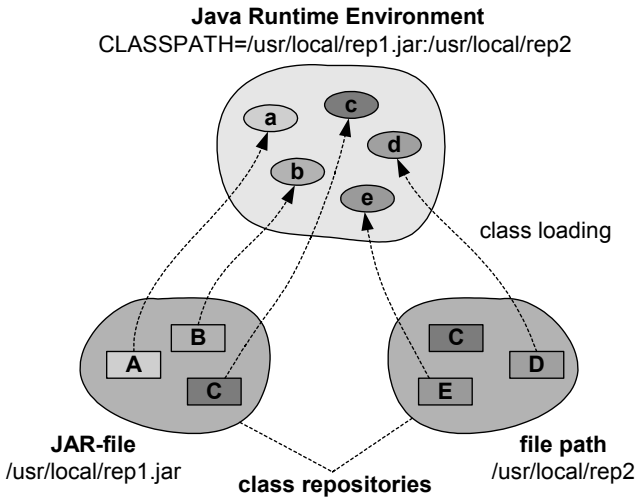


Fig. 1. Class loading with two conflicting class repositories

As shown in fig. 1, let us assume that the CLASSPATH contains two locations from where requested classes can be loaded. Apart from class C, only one byte code for each class can be found. However, class C is available in two different variants. Whenever the class loader is requested to load classes, it can only determine by the class name where to find the related byte code. Thus, it will load the byte code for C always from the JAR-file, since the JAR-file is specified in CLASSPATH before the file path. Consequently, the application is simply not able to instruct the JVM individually which variant should be taken.

Meanwhile, this scenario is not only limited to particular composed applications, but has been brought to standard Java applications by SUN itself. The latest release of their JDK 1.4.0 includes several third-party classes from the *Apache Software Foundation*, which are related to XML programming like parser and transformer classes. But these classes are still evolving, used and required in different variants, for example, in Apache Cocoon [5]. Consequently, as the system class loader usually looks for requested classes in the Java runtime library first, and only after that in user-related Java archives defined within CLASSPATH, it is difficult to load byte code for these classes from other archives instead. For this reason, developers have proposed special workarounds, e.g. using certain interpreter options to change the search sequence or modifying even the original runtime library and removing the clashing class files. As a result, the major problem in class deployment still remains, and even sophisticated Java solutions like Sun Enterprise Java Beans (EJB) [6], Sun ONE [7], FlexiNet [8] or Harness [9] do not address the following question: How can the Java Virtual Machine be configured, not only statically but even dynamically during runtime, which of the deployed classes are needed and should be selected based on using additional properties other than the class name?

In this paper, we present a new approach to Java class deployment to solve the illustrated problems. For this purpose, we propose *Java Class Collections* which allow to group classes individually on a logical level and decorate them with additional properties. Additionally, we introduce a customizable way of how classes are selected, namely also with respect to certain properties like version information or application demands. This decouples the mechanism of *application-based class selection* and *deployment-dependent class location*, supporting the development as well as the execution of applications. As a result, software engineers can focus on the programming of applications and do not have to deal with the configuration of the actually used installation environment and vice versa.

The paper is organized as follows. Section 2 reviews existing approaches of Java class deployment. In section 3, we present our approach and its realization with Java Class Collections in detail, and section 4 illustrates the application of our approach. The paper is concluded by section 5, where areas of future work are outlined.

2 Java Class Deployment

In this section, we present different approaches to Java class deployment. Our discussion does not consider approaches that require a proprietary *Java Development Kit (JDK)* or *Java Runtime Environment (JRE)*, since such approaches may not be deployed according to the license agreements [10].

2.1 File System

The simplest kind of Java class deployment is placing the Java byte code into the file system, where the directories follow the package structure. For example, fig. 2 shows the arrangement of the package *odin* and its descendant packages *system* and *service*,

which contain some interfaces and classes. To use these resources, the corresponding CLASSPATH has to be set to /usr/sdk.

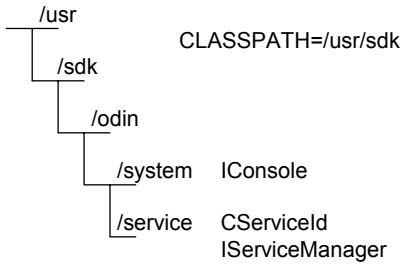


Fig. 2. Java Class Deployment using file system

In general, this approach is used during development time, when classes are often rebuilt and developers do not want to deploy the byte code yet. Hence, the main advantage of this deployment strategy is the simple exchange of single byte code files, but it is not well-suited for production environments where it is difficult to maintain a large number of files. This approach can often only be used when the byte code files are located on a mounted file system, since the native system class loader is not able to load classes from elsewhere.

2.2 Java Archives (JAR)

Another kind of deployment strategy uses *Java Archives* or so called *JAR files*. As described above, JAR files are used to assemble classes into one single file and organize them internally similar to the file system approach, as illustrated in fig. 3.

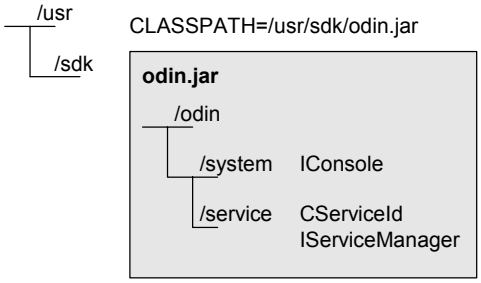


Fig. 3. Java Class Deployment using Java Archives

This approach is used widely for deploying an application or class library into a production environment. For this purpose, the developer generates an archive which contains the byte code of the application. Moreover, some developers also add third-

party classes which are required to run the application. The main advantage of this approach is its simplicity; the developer only has to deliver a single JAR file. In turn, the user/customer can include that single archive into the environment setting CLASSPATH and is able to start the application without coping with dozens of byte code files within the file system.

In addition, the packaged files within a Java Archive are compressed to allow the deployment of *Java Applets* which are started within a Java enabled Internet browser and are downloaded from a web server before they are executed [3]. Common to all application scenarios is that Java Archives are used as they are provided, they are not split nor repackaged. They are just placed somewhere, such that the class loader is able to access the archive.

2.3 Web Archives (WAR)

In contrast to the former approaches, which are used to run applications on the local machine of the user, the *Web Archives* approach is only used for services that are executed within a web server.

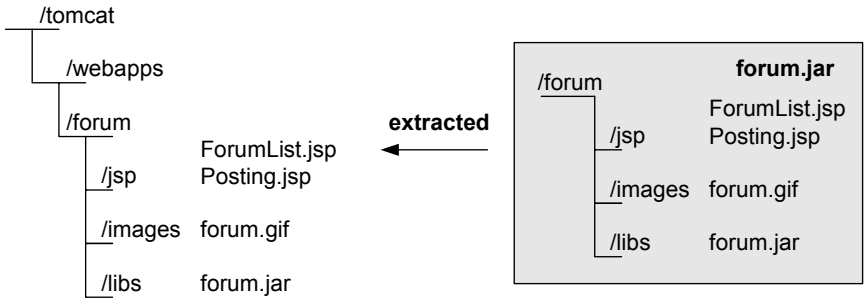


Fig. 4. Java Class Deployment using Web Archives

Similar to Java Archives, Web Archives are used to package byte code files, enabling the developer to deploy just a single file [11]. But in contrast to Java archives, they are not just used primarily for packaging byte code files, but also for grouping web resources like dynamic web pages or image files within a single file. In this context, they are used to deploy and install a web service by extracting all intrinsic files at the startup of the servlet engine (fig. 4), so that the web archive is not longer needed afterwards. However, although Web Archives are slightly different compared to the former approaches, the class selection procedure remains the same.

2.4 Java Web Start (JWS)

Recently, a new approach of application deployment has been introduced by *Java Web Start (JWS)* and the *Java Network Launching Protocol (JNLP)* in the latest

release of Sun JDK 1.4.0 [12]. This approach is mainly driven by the fact that Java Applets have to be downloaded from the web server each time when they should be executed. With *Java Web Start*, the applet code is cached on the client side and used to run the applet the next time it is started. Thus, applet startup time is decreased, and since modifications of the original applet code can be checked on the web server, the user is enabled to always get the most up-to-date version when it becomes available.

As shown in fig. 5, instead of requesting the applet code directly, the HTML browser retrieves the corresponding JNLP configuration file and launches the Java Web Start plugin. In turn, the latter checks for updates of the cached applet code by evaluating the JNLP file and starts the applet, eventually loading a newer version from the server before.

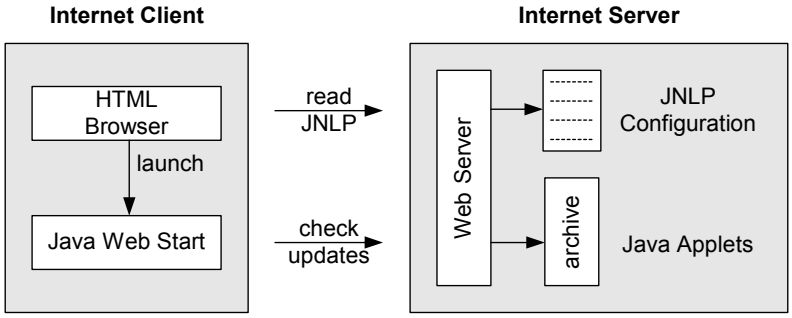


Fig. 5. Java Class Deployment with Java Web Start

JWS does not really introduce a new kind of class deployment strategy. It adds some caching and update checking functionality, but it is based on Java Archives and hence inherits similar features and limitations.

2.5 Custom Class Repository

In addition to these standardized strategies, which are more or less specified by SUN, there are other approaches for class deployment summarized under the term *Custom Class Repositories*. The classes, respectively their byte codes, are stored in a custom class repository and a corresponding, self-defined class loader is used to load the byte code from that repository. Certainly, there are different kinds of custom class repositories, but the principle of class loading is not changed. Each still relies on the introduced class loading strategy of SUN, customizing the way how classes are loaded and managed, but not how classes are selected.

An example is illustrated in fig. 6, where the servlet engine *Jakarta Tomcat* uses a customized class loader to retrieve the byte code of compiled *Java Server Pages (JSP)* and checks whether they have been modified since the last request. Thus, the servlet engine is able to manage a class repository that consists of a web server infrastructure where JSPs can be added and removed and consequently where byte code can appear and disappear dynamically. However, although handling this kind of

custom class repository is quite different from the original class loading approach, especially in the way how class updates are handled, it does not really change the way classes are selected.

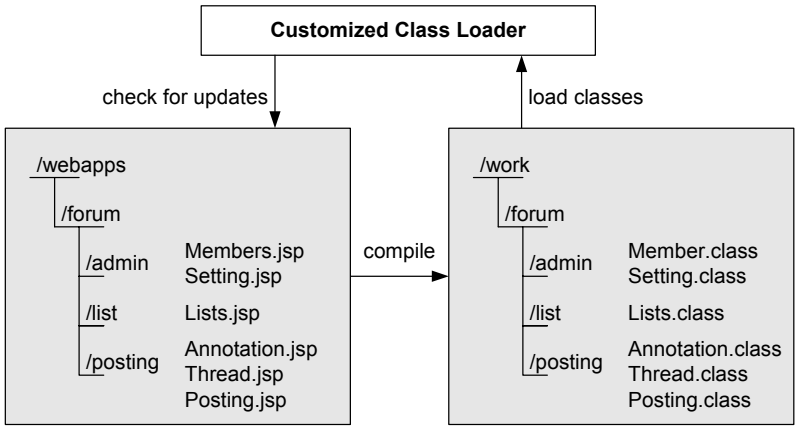


Fig. 6. Java Class Deployment with Java Servlet Engine

2.6 Summary

There are various approaches to Java class deployment which are more or less suitable in different application scenarios. Common to all approaches is that they are based on the original class loading approach proposed by SUN. Classes are selected by using their name and containing package, the associated byte code is located by evaluating these parameters to navigate through a file system directory or archive structure, and an appropriate class loader retrieves the byte code. To the best of our knowledge, there is no approach available which allows to change the class selection strategy.

3 Java Class Collections

In the following, we present our approach to Java class deployment based on managing class relationships and dependencies with so called *Java Class Collections (JCC)*. At first, we illustrate the objectives and basic ideas of JCC, then we describe our architectural approach, and finally, we describe its realization.

3.1 Objectives

Our proposed approach to Java class deployment is supposed to satisfy the following objectives:

Tagging and selecting classes with custom properties
As described above, ordinary Java class deployment approaches do not offer class selection using properties other than the class name. Thus, one objective is to tag classes with custom properties and to evaluate them for selecting a certain class.

Declaration of dependencies and compatibilities
Classes usually have dependencies, which must be satisfied before they can be loaded. In turn, a particular variant of a class might be compatible with other variants of the same class and can be used instead. Consequently, the declaration of dependencies and compatibilities among classes is another important objective.

Definition of class collections
Although it would be possible to tag and select every class individually, it is more useful to handle a collection of classes uniformly. These collections should not be limited to a Java Archive, but freely configurable with respect to which class should be included.

No modification of legacy applications
An important issue is the compatibility with existing applications. This should not only be valid for applications where the source code is available, but also for legacy applications that can not be re-compiled or even modified.

Transparent Handling of Existing Class Repositories
There are many Java archives and other class repositories which are heavily distributed and used. Apart from the suitability for legacy applications, any new approach must also support the usage of existing class repositories instead of forcing the developer to learn and apply a new strategy.

Although there might be further objectives, we think that these are the most important ones, since they change the style how classes are *deployed*, but without changing the way classes are *utilized*. This is important for standard Java programming, because developers do not want to change their technique of writing Java applications.

3.2 Concept

The basic idea of our proposal is to decouple the definition of physical deployment from the logical collection of classes. By introducing so called *Java Class Collections (JCC)*, we are able to associate additional features separately from existing deployment strategies and class repositories. Hence, the proposed approach works in conjunction with conventional Java deployment strategies, and no repackaging is needed. In addition, we introduce XML based configuration files [13] which define the composition of JCC and are used to tag them with custom properties, dependency and compatibility facets.

Finally, we present a custom class loader which is able to evaluate the configuration files and pick classes according to given properties. Therefore, it is possible to integrate legacy Java application transparently, simply by the fact that all class loading requests are handled through this customized loader without special knowledge or treatment by the application.

3.3 Architectural Approach

As shown in fig. 7, the proposed customized class loader uses an XML configuration file to determine which classes taken from different Java archives should be loaded when the application issues a class loading request.

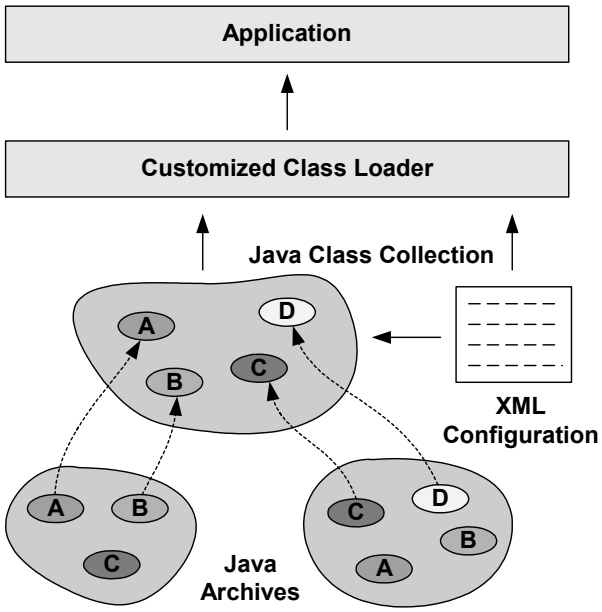


Fig. 7. Java Class Collections and Customized Class Loader

In the example, two archives used by the application contain partly identical classes A, B and C, which usually lead to conflicts when an ordinary class loader would be used. With the assistance of JCC, the deployer of the application can supply a configuration file that specifies a new class collection and the classes belonging to it. In this way, it is possible to direct class loading requests individually for each application by specifying an appropriate JCC definition, but still using conventional Java archives.

3.4 Realization

An excerpt of the configuration file is shown in fig. 8, which represents a sample configuration for the Apache-Xalan classes. The first part groups classes of release 1.2.2 together by specifying a property *version* besides the library name *apache-xalan*. In the second part, the same is defined for classes of release 2.2. Both parts declare JAR-files belonging to a certain variant of the collection and the classes that are contained within these files. For flexibility and efficiency, the declaration is based on regular expressions, thus it is easy to address various classes with few statements. In the case where no appropriate variant of a certain JCC can be found, the compatibility section allows to specifiy which other variant might be used instead. Similar to this definition, it is possible to declare dependencies, such that a JCC can only be used when another JCC is available.

```
<collection name="apache-xalan">
  <variant>
    <property name="version" value="1.2.2" />
    <dependency name="apache-xerces">
      <property name="version" value="1.4.1" />
    </dependency>
    <file location="/usr/sdk/xalan-1.2.2/xalan.jar">
      <resource name="org/apache/xalan/.*" />
    </file>
  </variant>
  <variant>
    <property name="version" value="2.2" />
    <compatible name="apache-xalan">
      <property name="version" value="1.2.2" />
    </compatible>
    <file location="/usr/sdk/xalan-2.2/xalan.jar">
       <resource name="org/apache/xalan/.*" />
       <resource name="org/apache/xpath/.*" />
    </file>
  </variant>
</collection>
```

Fig. 8. Definition of Java Class Collections

Although the example is quite short, it shows how simple ordinary class repositories can be disposed in class collections without changing any existing files. The configuration is extensible with additional properties and JAR-files, and it is even possible to group classes from different JAR-files within a single class collection. In this way, we are able to define *composition-based grouping* independently of the *deployment-dependent packaging* of JAR-files.

Since the basic approach for implementing a user-defined class loader is well-documented in the literature [2,3], we will focus on some special features. In this context, the major problems are the integration of legacy applications and locating the appropriate byte code of a requested class.

The first problem is solved by defining an *application loader*, which is started by the Java Virtual Machine instead of the application itself. It instantiates the customized class loader and directs this class loader to load the application classes. In

this way, all loaded application classes are associated with it and consequently, all class loading requests of the application are routed through that class loader. This is feasible for all kinds of legacy Java applications as long as they are not defining an own class loader which breaks the given class loading policy of Java. According to this policy, all class loaders have to delegate loading requests firstly to their parent, and only have to load the class itself if it can not be loaded by one of its parents.

When the class loader of the JCC is requested to load a certain class, it only gets the name and the package of the class, similar to all other conventional class loader implementations. But how can the JCC decide only with this information which particular class should be selected among several classes with the same name? This problem is solved by evaluating the configuration file illustrated above and looking for a matching entry within the variants of the JCC. For this purpose, it is important to define for each application which particular JCC should be used by the application, as illustrated in fig. 9.

```
<application name="example-app">
  <collection name="example">
    <property name="version" value="1.0" />
  </collection>
  <collection name="apache-xerces">
    <property name="version" value="2.0.0" />
  </collection>
  <collection name="apache-xalan">
    <property name="version" value="2.2" />
  </collection>
</application>
```

Fig. 9. JCC definition of an application

When the application is to be loaded, the system determines the required JCCs and the contained classes by *mapping* the given properties in the application configuration onto properties in the JCC configuration. The actually used algorithm is quite straightforward: it simply selects the first JCC meeting all given properties in the application configuration. Hence, these have to be refined in the case where there is more than one matching JCC.

Within this context, the system also checks whether there are two JCCs requested by the application that partly contain the same classes. If so, this would again lead to the problem which class to select among the given ones. Moreover, this could also be interpreted as a configuration error, since an ordinary application can typically use only a single variant of a class at a time. Therefore, the application and JCC configuration must ensure that there always is exactly one byte code variant available for each requested class. Upon a class loading request, the customized class loader has to look only for an entry *matching* the given class and package name. In our approach, this is achieved using regular expressions, but of course any other algorithm would be suitable as long as it can retrieve the right class by evaluating the given class and package name of the class loading request.

Finally, the presented approach does not incur much overhead on class loading compared to the original method. In fact, only when a class is requested the first time, the appropriate byte code has to be selected by checking the configured JCCs and

their regular expressions if they match the given class name. After loading the byte code, it is cached, and further class requests will retrieve it quite fastly from the cache. Thus, the performance of the approach primarily depends on the overall number of regular expressions used to describe the JCCs and not on the number of the JCCs itself. In this context, the usage of regular expressions not only offers a convenient way to configure JCCs, but it also allows to define them with only a few statements, hence decreasing lookup time.

As a result, applications relying on the standard class loading approach can be used transparently and additionally benefit from our approach. There is only a small impact on performance, and the definition of the JCCs and application configurations is both flexible and simple.

4 Application of the Approach

In the following, we demonstrate the suitability of our proposal in various application scenarios and highlight its benefits.

4.1 Multi-Application Hosting

One of the major advantages of Java (but actually also a disadvantage) is the *Java Virtual Machine (JVM)*. On the one hand, the JVM offers the ability to develop Java applications only once and run them on any platform for which a JVM exists (Write *Once, Run Anywhere* or briefly *WORA*) [10]. This clearly is essential for writing platform-independent applications like Java Applets [3], Java Servlets [22] and Enterprise Java Beans (EJB) [6]. On the other hand, an extra JVM has to be used every time when a Java application is to be started. Consequently, an application server which wants to host and compose several applications is usually forced to start numerous JVMs, each acquiring resources and classes partially used concurrently. Within this context, we have previously introduced a new approach using a single JVM to host several applications at the same time [14]. This approach not only saves resources, but also eases the interaction between applications which are not longer forced to communicate across JVM borders using RMI or CORBA [15,16,17]. However, this requires a new strategy for handling class requests, since class dependencies of different applications within a single JVM have to be harmonized. As a result, it is not longer feasible to rely exclusively on class and package name for this purpose, but it is necessary to include additional properties like versioning information, class compatibility and other application constraints.

Having this in mind, we have developed a new Java middleware kernel as part of ODIN [18], which is able to host several applications concurrently within a single JVM and use the presented approach to define JCCs and fulfill applications' class requirements. The latter is achieved by specifying the required JCC of each application in a configuration file and by using this information to manage so called *Java Class Spaces* [14] in which JCCs can be shared or used exclusively by applications as required.

As illustrated in fig. 10, 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*, which are independently configured by different JCCs. Furthermore, there are two class spaces, which contain class collections *Jcc1* and *Jcc2*, shared by the applications.

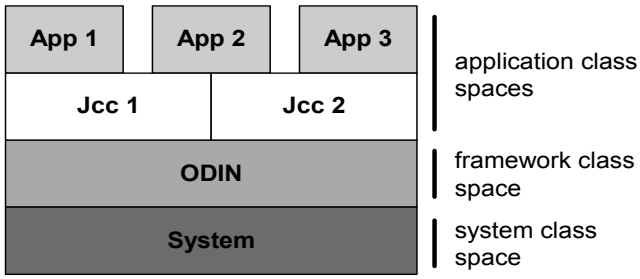


Fig. 10. Class Space Organization in ODIN

4.2 Composable Application Systems

In general, running Java applications on a single node is relatively simple when all required classes have been installed and configured properly. However, it is much more difficult to maintain plenty of network nodes, each hosting different applications, which in turn have to be composed of classes located in several archives, potentially spread over the network. As an example, application service providers offer their nodes for hosting customers' applications, but they are not willing and often also not able to reconfigure each host to run a new application. Instead, it would be highly desirable if applications on every node could be composed of appropriate classes dynamically, regardless of their deployment and location in the network. And since different, but also evolving applications request different variants of the same class with identical class names, each variant has to be installed and selectable by each application individually. As a result, a network of Java archives is created with numerous classes, which can not be distinguished only by their class name. Thus, a new strategy is needed to select the right classes for each application, crossing deployed Java archives. This is resolved by using the presented approach as already illustrated in fig. 7. Legacy Java archives are disposed into the network and particular defined JCCs are used to map the *logical* grouping of Java classes required by each application onto the *physical* grouping of Java classes, given by legacy Java archives.

Therefore, we are able to create network based class repositories, mediating required classes not only by their class and package names, but by evaluating further application constraints. Thus, we obtain a JCC network, similar to a file sharing system, but in contrast, we share JCCs according to the slogan *Write Once, Compose Elsewhere, Run Anywhere*. The presented approach has proven to close this gap within several ongoing research projects like *netzspannung.org* [19,20] and *Awake*

[21], which essentially benefit from the introduced flexibility of class deployment and transparent usage of legacy applications like Jakarta Tomcat [22] or Apache Cocoon [5].

5 Conclusions

In this paper, we have illustrated the lack of existing class deployment approaches to resolve classes by evaluating properties other than class and package name. We have shown that this feature is needed for application-based class selection and have presented a new approach for grouping Java classes within so called Java Class Collections (JCC). Its suitability to decouple *physical class deployment* given by legacy class repositories like JAR archives from *logical class constraints* was demonstrated by application examples from ongoing research projects. Although we have included JCC in our middleware kernel ODIN and hence we have used it in real production situations, there are still several topics for future work. First, the definition of JCC is currently done manually by defining regular expressions after analyzing the class repository. This should be done automatically and on the fly, therefore we are currently extending the implementation to inspect legacy archives and compile these configurations dynamically. Second, the definition of JCC is currently done by the class deployer when installing the archives, but this is a static task with respect to a running application. However, it would open additional, yet unknown customization options when the application is able to select classes not only by pre-defined constraints given by the application developer, but directed by runtime conditions and adjusting its behavior according to dynamically changing requirements.

Finally, while JCC opens a new kind of class deployment, we will continue to establish a network of class repositories which enables to *select* classes, rather than just to *resolve* them. Our vision is that this will support a different kind of deployment and configuration of distributed Java applications [23,24], namely that the related archives are not taken from a single source but obtained from various locations in the network. Thus, when an application should be run, it will be composed dynamically from several nodes of the network and automatically configure itself.

6 Acknowledgements

The presented approach has been evaluated and used in the implementation of the Internet platform *netzspannung.org* [19]. The related project CAT [25] is funded by the German Federal Ministry for Education and Research and is conducted by the research group MARS of the Fraunhofer Institute for Media Communication, St. Augustin in cooperation with the University of Siegen, Germany. Special thanks go to Monika Fleischmann, Wolfgang Strauss, Jasminko Novak, Gabriele Blome and Daniel Pfuhl.

References

1. Lindholm, T., Yellin, F. The Java Virtual Machine Specification. Addison-Wesley. 1999.
2. Venners, B. Inside The Java 2 Virtual Machine. McGraw-Hill. 1999.
3. Eckel, B. Thinking in Java. Prentice Hall. 2000.
4. 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). Canada. 1998. pp. 36-44.
5. Ziegler, C., Langham, M. Cocoon: Building XML Applications. New Riders. 2002.
6. Shannon, B., Hapner, M., Matena, V., Davidson, J., Pelegri-Llopart, E., Cable, L. Java 2 Platform Enterprise Edition: Platform and Component Specification. The Java Series, Addison Wesley, Reading, MA, USA, 2000.
7. Watson, M. Sun ONE Services (Professional Mindware). Hungry Minds. 2002.
8. Hayton, R., Herbert, A. FlexiNet: A Flexible, Component-Oriented Middleware System. Advances in Distributed Systems. LNCS 1752. Springer-Verlag 2000. pp. 497-508.
9. Kurzyniec, D., Sunderam, V. Flexible Class Loader Framework: Sharing Java Resources in Harness System. Proc. of International Conference on Computational Science (ICCS). LNCS 2073. Springer-Verlag 2001. pp. 375-384.
10. Sun Java Development Kit. http://java.sun.com/products/j2se/1.4/index.html
11. Rajagopalan, S., Rajamani, R., Krishnaswamy, R., Vijendran, S. Java Servlet Programming Bible. John Wiley & Sons. 2002.
12. Java Web Start. http://java.sun.com/products/javawebstart
13. Jung, E., Cioroianu, A., Writz, D., Akif, M., Brodhead, S., Hart, J. Java XML Programmer's Reference. Wrox Press Inc. 2001.
14. Paal, S., Kammüller, R., Freisleben, B. Multi-Application Hosting Using Class Spaces. In: Proceedings of the 2nd International Conference on Internet Computing (IC 2002). Las Vegas, USA. CSREA Press, 2002. pp. 259-266.
15. Maassen, J., Van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., Hofman, R. Efficient Java RMI for Parallel Programming. ACM Transactions on Programming Languages and Systems (TOPLAS). Vol. 23, Nr. 6. ACM 2001. pp. 747-775.
16. Orfali, R., Harkey, D. Client/Server Programming with Java and Corba. John Wiley & Sons, Inc. 1998.
17. Marvic, R., Merle, P., Geib, J.-M. Towards a Dynamic CORBA Component Platform. Proc. of 2nd International Symposium on Distributed Objects and Applications (DOA). Antwerpen, Belgium. IEEE 2000. pp. 305-314.
18. Open Distributed Network Environment. http://odin.informatik.uni-siegen.de
19. netzspannung.org, Communication Platform for Digital Art and Media Culture. http://netzspannung.org
20. Fleischmann, M., Strauss, W., Novak, J., Paal, S., Müller, B., Blome, G., Peranovic, P., Seibert, C. netzspannung.org - An Internet Media Lab for Knowledge Discovery in Mixed Realities. In Proc. of 1st Conference on Artistic, Cultural and Scientific Aspects of Experimental Media Spaces (CAST01). St. Augustin, Germany. 2001. pp. 121-129.
21. AWAKE - Networked Awareness for Knowledge Discovery. 2002. http://awake.imk.fraunhofer.de
22. Goodwill, J. Apache Jakarta Tomcat. APress. 2001.
23. Lewandowski, S. M. Frameworks for Component-based Client/Server Computing. ACM Computing Surveys. Vol. 30, Nr. 1. ACM 1998. pp. 3-27.
24. Little, M. C., Wheater, S. M. Building Configurable Applications in Java. Proc. of the 4 th International Conference on Configurable Distributed Systems. Annapolis, Maryland. 1998. pp. 172-179.
25. 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