PAAL, Stefan
KAMMÜLLER, Reiner
FREISLEBEN, Bernd

Customizable Deployment, Composition and Hosting of Distributed Java Applications

published on netzspannung.org:
http://netzspannung.org/about/mars/projects/pdf/awake-2002-1-en.pdf
14 March 2005

First published: Proceedings of the Conference on Distributed Objects and Applications (DOA 2002). LNCS 2519. Irvine, USA. Heidelberg: Springer, 2002, pp. 845-865.

Customizable Deployment, Composition and Hosting of Distributed Java Applications

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. Deploying and running Java applications on a single host is covered by standard approaches. However, when applications are dynamically deployed on distributed hosts, the situation is quite different. In this context, applications are likely to be composed of classes, located in remote repositories and possibly related to identical class names. Hence, the typical class loader approach is no longer feasible to resolve the right byte code. Moreover, the native *Java Runtime Environment (JRE)* has originally not been designed to host more than one application concurrently within a single *Java Virtual Machine (JVM)*. Thus, there are also unresolved issues concerning hosting distributed applications. In this paper, we present a new approach for a customizable Java application middleware with respect to the topics of *application deployment, composition and hosting*. Finally, the application of the approach within a distributed middleware platform is presented, wherein applications are *customizably* deployed, *dynamically* composed and *concurrently* hosted.

1 Introduction

In contrast to other, native programming languages like C++, Java does not only come with a programming environment but also with a runtime environment, the so called *Java Runtime Environment (JRE)*. It shields the application from the underlying system environment and provides access to its resources, independently of the current operating system or hardware. One part of the JRE is the *Java Virtual Machine (JVM)* which hosts the Java application and is responsible for locating and loading the related byte code and finally starting the execution of the application. In this sense, the JRE can be regarded as a quite simple application framework which is able to host a single application.

Before a Java application can be started by the JVM, the related byte code of the application classes must be located, loaded and resolved. The latter means that loaded classes may refer to other classes which must also be loaded before the referring class

can be used [1]. For this purpose, the native JVM delegates usually the class loading request to the so called *system class loader* that 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 other class loaders which are *user-defined* and are free in the way the byte code is located and loaded [4,5].

However, common to all class loaders is that each can load the same class only once during its lifetime, whereby two classes are assumed to be the same class when their FQCNs are equal. Due to this fact and the fact that Java applications are usually composed of many classes, the deployed classes are organized in tree-like organized packages, whose names are prepended to the class name, forming the FQCN.

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 or debug variant of the class. Of course, many application frameworks 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*. As a consequence, it is not possible to instruct existing application frameworks which classes should actually be used when composing a certain application. Instead, the application framework loads needed classes as they are requested, while it is assumed that these classes are found somehow during runtime [3].

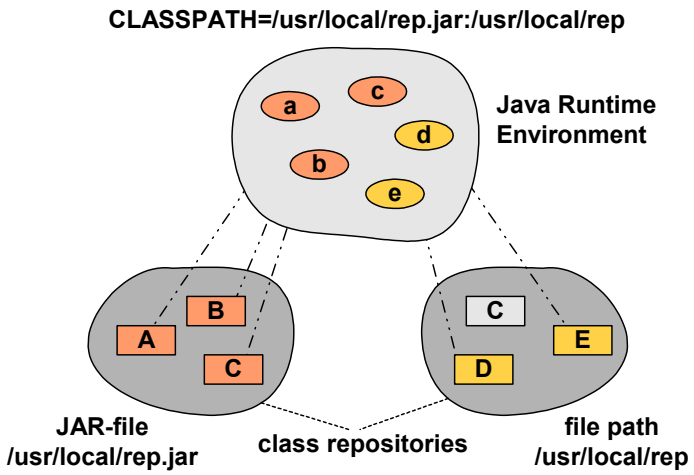

Fig. 1. Class loading with two conflicting class repositories

As shown in fig. 1, assuming the CLASSPATH contains two locations from where requested classes should be loaded. Beside of class C, only one byte code for each class can be found. But class C is available in two different variants. Thus, when the class loader is requested to load a class, it can only determine by its name where to find the related byte code. Hence, it will load the byte code for C always from the

JAR-file, since it is specified in CLASSPATH before the file path. As a result, the major problem in application composition and deployment still remains, namely how to build an application framework that can be customized, not only statically but even dynamically during runtime, which classes of the deployed ones are needed, should be selected and used to compose the application?

Apart from the class *selection* problem, there is another related problem concerning class *hosting*. The native JVM and its system class loader have been designed to host only a single application and are run by the underlying operating system within a single process. Therefore, further Java applications have to be started in additional JVMs. Although this is well-suited for stand-alone applications running on a single machine, it causes problems when a multi-application environment like a server platform should be configured to run more than one application, requiring the same classes in different variants concurrently and wanting to share data within the border of a JVM. As mentioned above, the problems are caused by the fact that class loaders can load a certain class only once during their lifetime. In detail, all loaded classes are put within so called *namespaces*, each associated with one class loader, wherein no two classes with the same FQCN can co-exist.

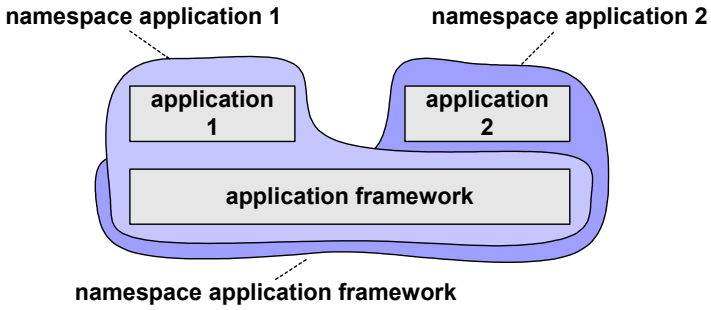

Fig. 2. Application hosting with separated application namespaces

There are implementations using more than one classloader, each managing its own namespace. That way, they can host the same application in different variants and are even able to reload dynamically the application code 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 application code completely and makes it difficult to share data directly among each other, as illustrated in fig. 2.

There are three class loaders and their associated namespaces, each for the applications and the framework itself. After starting the application framework, applications 1 and 2 are dynamically loaded and placed in two different namespaces, so that their classes are shielded against each other. Though this is necessary for concurrent application hosting when using classes with the same FQCN, the applications can only interact by using classes located in the framework namespace.

To understand the problem in more detail, it is useful to regard some class loading details, extracted from [2,3]. Beside of the provided class loaders like the *bootstrap class loader* and *system class loader*, application developers are able to define further

so called *user-defined* class loaders. Like any other class loader, they are ordinary Java objects, instantiated from a class that is inherited from *java.lang.ClassLoader*. In order to customize the class loading process, the user-defined class loader has to override the appropriate methods of *java.lang.ClassLoader*, typically *findClass* respectively *loadClass*. Then, after a related class loader object has been instantiated with *new*, it can be used to load a class by calling the method *loadClass* with the appropriate class name. Moreover, each loaded class is assigned to the namespace of the used class loader and subsequent class loading requests issued by its objects will be automatically passed to that class loader. Thus, a user-defined class loader is hooked into the system simply by using it to load an initial class with *loadClass*.

Furthermore, each class loader has exactly one parent class loader which is assigned upon its creation and that can not be changed afterwards. If no parent class loader is given, then the *system class loader* is automatically assigned as parent. This leads to a *parent-delegation model* and forms a class loader tree with the system class loader at the root. With that, the class loaders are usually implemented in such a manner that they delegate class loading requests at first to their parent and will not try to load the class, if the parent returns the requested class. But in the case the parent class loader is not able to load the requested class, for example it is not accordingly configured or an error occurs, the parent raises an exception and the request is returned to its child. Finally, if even the firstly called class loader at the end of the chain is also not able to load the class, a *ClassNotFoundException* is raised and it is up to the application to handle the exception.

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*. 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. An illustrating example is shown in fig. 3, where three class loaders CL1, CL2 and CL3 are shown.

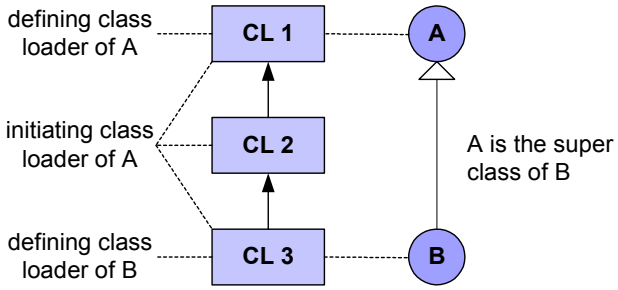

Fig. 3. Initiating and defining class loaders

Assuming 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. 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 can now be defined more precisely: it is simply a list of classes, for which the class loader has served as an initiating class loader. All classes within this list can *see* and use only the other classes in that list or loaded classes from their parent class loader, but not from any other class loader, located elsewhere.

Due to this fact, objects in two namespaces can only share data and collaborate, if they *see* the associated classes, or in other words, if the methods of the object stem from classes or interfaces, which are loaded from the same parent class loader. In fig. 4, B and C have been put in different, separated namespaces, since they are loaded and defined by CL2 and CL3, which are not chained. In turn, class A, the super class of B, is loaded and defined by CL1, which is the parent class loader of CL2 and CL3. Assuming that object o1 and o2 have been created and object o2 gets the reference of o1. Then, object o2 is able to downcast the reference of o1 to A, but not to B, since B is in another namespace and not visible to o2. Therefore, it can only call methods of o1, defined in A.

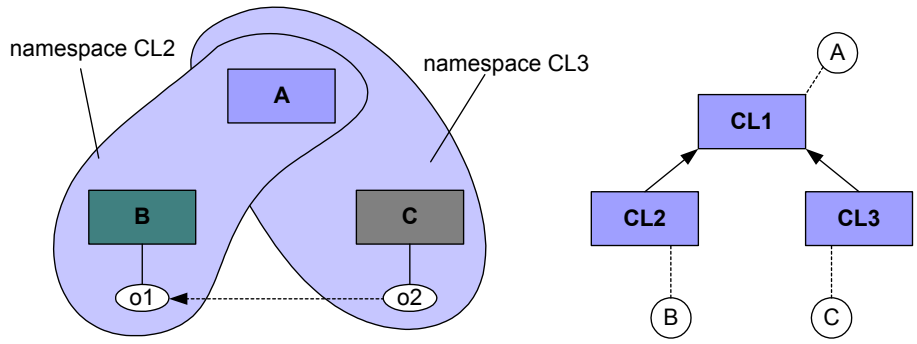

Fig. 4. Calling methods from objects in another namespace

Similar to the class loading problem mentioned above, the existing approaches do not offer the ability to *select* which classes should be put in separated namespaces and which can be kept in common namespaces nor how this task can be customized dynamically and individually for each application. This is especially important for a multi-application environment, when two unknown, dynamically loaded applications want to share common data with each other or with the application framework itself.

In this paper, we present a customizable Java application middleware to solve the illustrated problems. For this purpose, we introduce so called *Java Class Spaces*, which manage namespaces and the loaded classes in a customizable manner, enabling the application middleware to host different applications concurrently. Second, we propose so called *Java Class Collections*, which group classes individually and decorate them with additional properties. In contrast to the ordinary Java class loading approach, where classes are selected, located and organized within the JVM just by using their class names, we introduce a customizable way of how classes are selected

and organized dynamically, 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 customizable deployment, composition and hosting of distributed applications.

The paper is organized as follows. Section 2 discusses features of Java application frameworks with respect to *application development, composition, deployment and hosting*. In section 3, our approach to develop a customizable Java application middleware is presented. In section 4, the application of our approach is illustrated, and section 5 concludes the paper with an outline of future work.

2 Java Application Framework

There are various approaches to application frameworks running on top of a native JRE, which claim to ease the development and which support more or less *deployment*, *composition* and *hosting* of applications. In the following, we outline these tasks and how they are accomplished by existing approaches.

2.1 Application Development

The major task of an application developer is implementing the so called *business logic* of the application. But since a Java application can not run on its own, it has to follow the guidelines for writing an application itself that can be executed within the used runtime environment. Considering the native JRE, it has to provide a static method *main*, where the execution is started after creating the JVM. Conversely, the application can direct the JRE to load application classes by using the operator *new*. In addition, the application wants to access the underlying system and its resources as well as utilizing existing application code. For these purposes, the *Java Development Kit (JDK)* comes with a lot of classes that can be used to access the system resources and ease application development by providing ready-to-use classes for many purposes. In particular, the JDK offers methods how an application can be extended during runtime, requesting additional classes dynamically for example with the static method *forClass* or using the *reflection* API.

There are several specialized application frameworks that provide certain development support [6]. A widely used implementation for Internet applications is *Jakarta Tomcat*, which is used to host so called *Java servlets*, a specialized kind of application, and expects as the execution start point the method *init* within the servlet class [7]. It also comes with many classes, which assist the developer in implementing HTTP request handling and serving HTML pages.

2.2 Application Deployment

Besides application development as supported by the *programming environment*, the application must also be deployed in the actual *runtime environment* before it can be

executed. At first, the application framework should support the concurrent deployment of the same class in different implementation variants like debug or multi-threading releases. Though this is quite simple for stand-alone applications, it is getting rather complex when running a multi-application environment like *Jakarta Tomcat*, dealing with various installed class files.

Second, the original class loading approach 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 individually, e.g. over the network [8]. In this scenario, the user-defined class loader focuses usually on the task how the requested classes can be loaded. But some of them also deal with the question how to locate the requested class and use their own mechanism to resolve the location of the class. This allows to place the classes elsewhere, assuming there is a class loader to handle the related location and loading issues. An example of an Internet-based deployment approach is described in [9], where application code is spread over the Internet and dynamically loaded using particular class loaders.

2.3 Application Composition

A further important feature of Java application frameworks is the composition of applications from deployed classes, even dynamically configured during runtime. While the *application developer* defines within the application code *which* classes should be loaded, the *application composer* configures from *where* to load the requested classes. Within a native JRE the latter task is originally fulfilled by defining the environment setting CLASSPATH, specifying file paths or so called JAR files containing the needed classes. The *system class loader* provided by the JRE evaluates that setting to load classes, but it is only capable to access local file systems.

While this scenario is quite widely used, it inherits some essential weak points with respect to the origin idea of application composition. First, it is not possible to exclude single classes from a given path or JAR file, therefore classes with the same name located within two specified file paths are selected by the order in the CLASSPATH setting, leading to strange behaviors when not properly handled. An example are the Java classes of *Apache Xerces* [10], which accompany the latest SUN JDK 1.4.0 [11], but are also part of existing program installations like *Apache Cocoon* [12]. Due to evolving releases, there are conflicts which can not be always handled by appropriate orderings in CLASSPATH. This has even forced some installations to modify delivered JAR files for proper execution!

Second, in our opinion applications are not just composed by specifying the location of single classes or JAR files. Moreover, the latter are often packaged rather simply by the application deployer, which has grouped related and non-related classes within a single file for easy delivery. In contrast to that, applications are rather composed by the fact that groups of certain classes represent a particular needed functionality. These groups can span different JAR files or just be parts of them. As a result, application composition has to be separated from application deployment. Or in other words, *functionality driven class composition* should be independent of *installation based class deployment*. But this can not be achieved just by evaluating

the environment setting CLASSPATH, since it is used to define concurrently which concrete classes can be loaded and from where they are retrieved.

There are several application frameworks which extend the ordinary class loading approach and overcome some of the depicted limitations. They often use additional configuration files, enabling the specification of certain dynamically loaded classes independently of their location. Examples are plugins or handlers which provide certain exchangeable functionality like in *Java Database Connectivity (JDBC)*, whose database connectors can by selected transparently to the application, depending on the actually used database. Another example is *Jakarta Tomcat* that allows to define dynamically loaded protocol interceptors.

2.4 Application Hosting

As described above, the native JRE is not capable to host more than one application. However, there are approaches for multi-application environments which overcome this limitation and are able to host different applications concurrently within a single JVM. They share commonly used resources like code libraries, memory space, network or database connections, thus improving greatly the utilization of these resources in contrast to the usage of a single JVM per application [13,14]. This is facilitated in a particular manner, since the applications do not need to cross the borders of their JVM for that purpose and can collaborate directly without using RMI [15], CORBA [16] or similar solutions [17]. Particularly in multi-user environments like Internet-enabled application servers, when many applications share the same resources, these features may be quite important [18,19].

Another objective in this context is the dynamic loading and unloading of applications. For example, it is usually not allowed to shutdown a JVM which hosts several web services due to the reload of one single service. For this 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 to their parents. That way, parent class loaders get always the first chance to load the class before their childs, and all applications can share data using classes from the framework.

To summarize, there are various tasks and requirements an application framework has to deal with, namely it has to support the development of applications as well as their deployment, composition and hosting. Although there are many approaches which handle these tasks, they are mostly not customizable in the way they do that. They often introduce new features which are used to overcome existing limitations, but at the same time introduce other constraints, for example, to the structure of the Java application to be hosted, such as *Enterprise Java Beans (EJB)* [20], *Apache Avalon* [21], or *SUN ONE* [22]. Thus, they are not generally suitable for legacy Java applications. Moreover, most of them are not available stand-alone, but part of another framework like *Jakarta Tomcat* [7]. What is still missing is the seamless support of legacy applications, which benefit transparently from introduced features regarded above, not only in a predefined way, but in a customizable manner.

3 Customizable Java Application Middleware

In the following, we present a customizable *Java Application Middleware* which accomplishes the depicted tasks, not only by adding new features, but by customizing existing features concerning *deployment*, *composition* and *hosting* of Java applications. In contrast to conventional frameworks, our approach does neither force developers nor end-users to deal with a new development or runtime environment. In fact, the middleware mediates transparently between the underlying runtime environment and the application on top. Thus, it is also able to run on legacy Java runtime environments, originally delivered by SUN, and to host legacy applications, which transparently benefit from the customizable features.

3.1 Objectives

The objectives of a customizable Java application middleware are presented in the following:

Transparent Handling of Legacy Application Code and Class Repositories
An application framework which forces the developer or deployer 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 a customizable application framework is the transparent handling of new features, allowing the developer to focus on actual tasks and enabling the integration of legacy application code.

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 deployer, it limits the way how applications can be composed when new resources like classes or resource bundles are added dynamically. Thus, a customizable application framework should not only support the registration of classes by the deployer before starting the framework but also by the application during runtime.

Definition of Class Dependencies on the Composition and Deployment Level
Usually, applications are composed of dozens of classes, each having certain dependencies on other classes. They are resolved by loading the required classes in the case they are not already in memory. Though one could try to manage each class separately, this is not convenient. It would be better to group classes together with respect to their functionality and define dependencies between these groups, thus reducing the complexity of the configuration. The declaration of the groups can be used to specify how and from where to load the associated classes. As a result, the application developer and composer could specify *which* classes are needed, and the application deployer could configure *where* to find the appropriate classes and how to resolve their dependencies.

Dynamic Configuration of Shared and Separated Classes
Certain application developers want their applications 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 the resolution of occurring conflicts.

Decoration of Classes with Custom Properties for Conditional Class Selection
Ordinary class loaders *locate* and *load* the requested classes by using the given class name. This is sufficient as long as there is no need to *select* the correct class out of a bunch of classes with the same name, e.g. representing different implementation variants of the same class. For this purpose, additional properties of the requested class should be used to determine which class variant actually to load. However, existing application frameworks and their class loader do not support the specification of further parameters other than the class name.

Determining the Required Classes before an Application Starts
After a Java application has been started, the JRE loads missing classes into memory step by step as they are requested. In case a class can not be found, the JRE throws a *ClassNotFoundException* and the program is usually aborted. Due to this sequence, an application deployer is not able to track down a missing class before the exception occurs or even to configure the environment appropriately to avoid the exception at all. This is even getting worse when the deployer has to configure an application framework, wherein many applications should be hosted, increasing the complexity of class dependencies. Therefore, a framework should indicate missing classes before the application is started.

No Modification of Legacy Applications and Class Repositories
A further important issue is the compatibility with existing legacy 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. Furthermore, deployed class repositories like Java Archives should be usable without adaptation or repackaging.

3.2 Architectural Approaches

Regarding the defined features of a customizable Java application framework, the first task is to introduce a new *composition paradigm*. Although a Java application developer uses classes to implement an application and packages in which all classes are placed *logically*, they do not contain any *composition information*, neither which classes belong together nor which classes are required by the application during runtime. Therefore, our first architectural approach is the introduction of so called *class collections*, which are used to compose applications in a new way as follows.

In contrast to ordinary application frameworks, where application composition is quite weakly defined by naming required classes and packages and providing

locations where to find *potentially* needed classes, class collections refine this by defining explicitly which classes can be loaded by the application and which can not be loaded; even more, the application framework can determine *before* an application is started whether all classes are available. This is especially important in dynamically changing deployment scenarios like Internet environments or peer-to-peer architectures. Furthermore, they offer the decoration of certain classes with additional properties. Thus, an application framework can be customized dynamically not to load the debug variant of the class, but the release variant. This can be done without changing the deployment settings, just by defining the composition of the application while it is started.

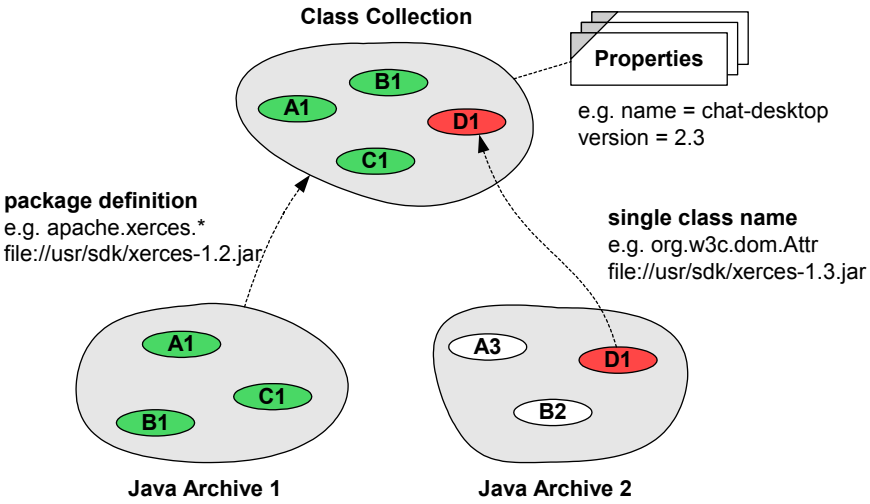

Fig. 5. Definition of a Class Collection, its properties and location of the related classes

Fig. 5 illustrates an example of a class collection. It contains properties, package and class names as well as the locations where the classes can be found, but not the byte code itself. All information is encapsulated within a configuration file, which can be browsed and searched by specifying the name and properties the searched collection should meet. This way, one could create a registry of class collection definitions, each associating classes and properties. An application framework can use this registry to request information where to find the appropriate classes and to compose applications with different class variants as it is needed by the application.

Even though class collections itself represent an essential improvement of how applications can be composed and deployed, there are still other features which are needed by a customizable application middleware. Based on class collections, we introduce another architectural approach, so called *class spaces*, which define the *layout of application code* within the application framework. Usually, 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, new namespaces are created in which the new classes can be placed. The old namespaces exist further, but are no longer used. This is done in a rather proprietary manner, and there is no way to define where the loaded classes have to be put, e.g. in the parent namespace or in the current namespace. However, this feature is required when two applications want to share data as illustrated above. Thus, we wrap *class spaces* around namespaces. They are able to monitor, which class should be loaded and can delegate customizably class loading requests to other class spaces.

In fig. 6, there is a class space SCS and two further class spaces UCS1 and UCS2 as children of the first. All objects, created by classes in UCS1 will have the class loader of UCS1 associated. In the case, one of these objects wants to create an object from a new class, the class loader is asked to load the byte code of the newly requested class. For this purpose, it determines wether 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. In this 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 with each other, using classes located in SCS.

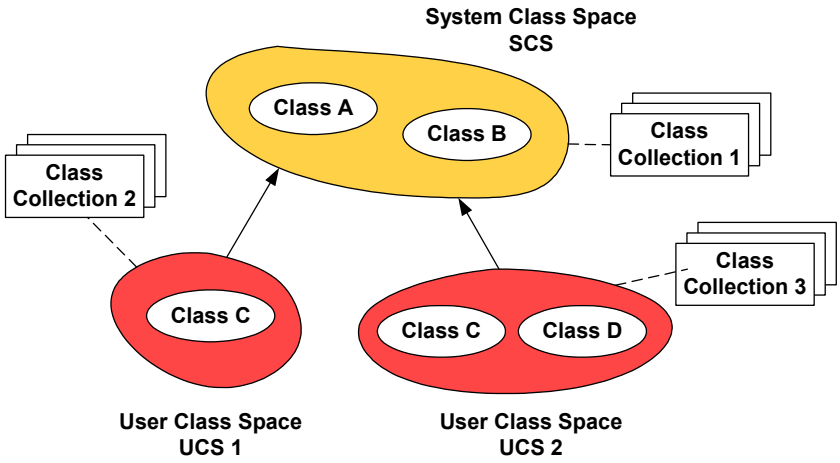

Fig. 6. Class Spaces

Another view on a more virtual level arises from the separation of objects due to the discrete, exclusive class layout. Chained class spaces, starting from a leaf class space following up the path to the system class space, load exclusively configured classes. In other words, there are no two class spaces in the chain, configured to load the same class. This is particularly important for casting objects, since an object can only be casted to a class or an interface which is loaded by a class loader on the path from its class loader up to the system class loader. As a result, though there are different class loaders loading classes from which objects are created, the objects can interact as if their classes are loaded by a single class loader. Thus, the usage of

different class loaders is virtually hidden and the collectivity of these objects is enclosed within a so called *object space*. Within this context, the smallest object space is rather trivial and is only associated with one class loader. In comparison to *namespaces*, which are associated with an *initiating class loader* and represents a lists of classes passed through the loader, *object spaces* are orthogonal to that. They do not represent a list of classes, but rather a list of objects which can interact.

In fig. 7, there are four class spaces, which are chained as illustrated. Furthermore, there are three non-trivial object spaces, containing several class spaces. The object space *A* contains class spaces *application1*, *root* and *system*, whereby object space *B* contains class spaces *application2*, *root* and *system*. The third object space *C* contains just *root* and *system*. Consequently, *application1* and *application2* can only interact using objects with related classes located in *root* and *system*.

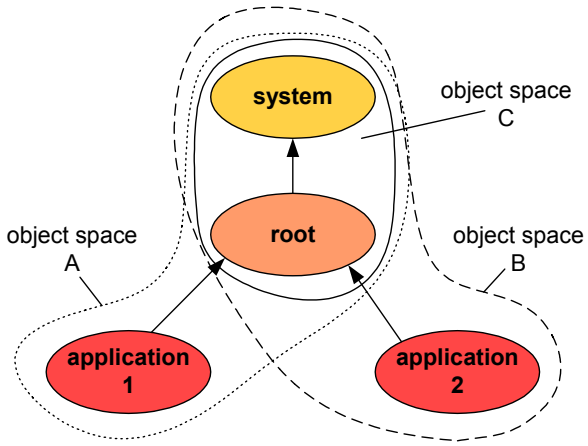

Fig. 7. Object Spaces

In addition, the configuration of a class space and its classes that can be loaded is combined with information where to find the related byte code. Although one is able to register separately each loadable class in a class space, it is much more convenient to use the introduced *class collections* for registration. With that, a class space can be configured to hold a whole bunch of classes in a particular variant at once.

A further special issue at this point is the question how the system should resolve registration conflicts between parent and child class space, if both want to register the same class. The answer is quite simple: all registrations must be checked against the registration in the chained class spaces, and if there is already a chained class space which handles the class, the registration is denied. Thus, concurrent classes could be managed up to a certain degree if the class spaces are configured properly by the application middleware.

Finally, the layout of application code, shared and separated classes is defined *without* modifying existing application code. The configuration process of class spaces and the arrangment of newly loaded classes are completely transparent to the application.

3.3 Realization

After presenting the basic ideas of our approach, its realization and its characteristics are illustrated in detail.

Class Collections. A class collection represents a virtual group of classes, which should be linked and associated with certain properties like version or debug information. The term *virtual* means, those classes are not really re-packaged and placed in a new JAR-file, but rather they remain completely in existing JAR-files or directory structures. The definition of class collections is done with an XML configuration file. An example of it can be seen in fig. 8, which shows a sample configuration for the *Apache-Xalan* classes. The first part groups classes of the release 1.2.2 together by specifying a property *version* beside the collection name *apache-xalan*. In the second part, the same is defined for classes of the release 2.2. For flexibility and efficiency, the declaration is done with regular expressions, thus it is easy to address various classes with few statements.

Though the example is quite short, it shows how simple ordinary class repositories can be disposed in class collections without changing any existing file. With it, an application framework can easily be configured where to find classes and which classes can be loaded at all. The configuration is arbitrarily extensible with additional properties and JAR-files; it is even possible to group classes from different JAR-files within a single class collection. That way, we are able to define *composition-based grouping* independently of *deployment-dependant packaging* of JAR-files.

```
<collection name="apache-xalan">
  <variant>
    <property name="version" value="1.2.2" />
    <file location="/sdk/apache-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="/sdk/apache-xalan-2.2/xalan.jar">
      <resource name="javax/xml/transform/.*" />
      <resource name="org/apache/xalan/.*" />
      <resource name="org/apache/xpath/.*" />
    </file>
    </variant>
</collection>
```

Fig. 8. Definition of class collections

Class Spaces. As introduced above, each class space encapsulates a class loader and its related namespace, whereby class spaces can be configured which classes they can load and which requests should be delegated to the parent class space. The configuration of class spaces can be done dynamically by the application itself as

shown in fig. 9. At first, a new class space *app* is created with the class space *system* as parent. Then, a new resource is registered in the class space, specifying that all classes, whose names start with *javax.xml.parsers* will be found in the given JAR-file and the class space should be able to load them. Another dynamic variant is shown after that using class collections with given properties. The class space is hereby configured to host classes defined in the collection *sun-javamail* with the version *1.2*. In contrast to the first variant, the usage of class collections defers the concrete configuration which classes can be loaded and where to find the byte code. The deployer is able to redefine that by modifying the configuration of the collections.

```
IClassSpace app = null;
app = getClassMgr().createClassSpace("app", "system");
app.registerResource("javax/xml/parsers/.*",
                             "/xerces-1.4.4/xerces.jar");
HashSet props = new HashSet();
props.add("version", "1.2");
app.registerCollection("sun-javamail", props);
Class parser = app.getClassLoader().loadClass
("org.apache.xerces.parsers.DOMParser ");
Class mail = app.getClassLoader().loadClass
                                  ("javax.mail.Message");
```

Fig. 9. Dynamic Class Space Configuration

Finally, at the end of the example, the class space is requested to load two classes; hence, the classes are injected into the class space. All subsequent class loading requests initiated by them will now be handled by the class space *app*. At this point, it should be stressed, that the class space does not load any class without a request.

An example for a static class space configuration is shown in fig. 10. A class space called *application* is defined with the class space *system* as parent. It is also configured to host classes from the collections *jakarta-tomcat* and *sun-javamail* with the given version numbers. The class space *application* is not automatically created, but whenever the application opens a class space with that name, the configuration files is read and the class space is configured respectively. In this context, the usage of class collections eases and enables the concrete composition by the deployer, who is able to redefine this by modifying the related configuration files.

```
<space name="application" parent="system">
  <collection id="jakarta-tomcat">
    <property name="version" value="3.2.3" />
  </collection>
  <collection id="sun-javamail">
    <property name="version" value="1.2" />
  </collection>
</space>
```

Fig. 10. Static Class Space Configuration

Object Spaces. As described above, *object spaces* are virtual entities, grouping objects together, which can interact as if they are loaded by one single class loader. This is quite important, in the case of casting or comparing objects, which might have

been instantiated by two classes, having the same FQCN, but are loaded within two class spaces. Therefore, the application framework offers methods to determine, whether two objects belong to one object space. Moreover, an object can belong to several object spaces, which can be also retrieved as shown in fig. 11.

```
Class c1 = system.getClassLoader().loadClass
                                    ("java.util.HashMap");
Class c2 = app.getClassLoader().loadClass
              ("org.apache.xerces.parsers.DOMParser");
Object o1 = c1.newInstance();
Object o2 = c2.newInstance();
boolean fSame = getClassMgr().sameObjectSpace(o1,o2);
HashSet objSpaces = getClassMgr().getObjectSpaces(o1);
```

Fig. 11. Using Object Spaces

3.4 Discussion

The presented approach is completely transparent to legacy application code and existing class repositories, no recompilation, rewriting or repackaging are needed. Instead, the concerns of *application composition, hosting* and *deployment* are separated using additional configuration files for *class spaces* and *class collections* as described. Certainly, they have to be set up appropriately according to the *composition* and *hosting* requirements of the applications and the class *deployment* on the platform. But with that, legacy applications are completely unaware to reside within *class spaces* or to be composed using *class collections*. The related application class spaces are created before the applications are started and the first class of the application is loaded into the related class space by the framework. In this context, the approach benefits from the automatic connection of loaded classes and the used class loader to hook itself transparently into the system for all subsequent class loading requests of the application. Indeed, this can only be guaranteed as long as the application does not create a user-defined class loader itself, which may break the configured class separation. Another topic is the overhead by using our approach in contrast to a legacy runtime environment. At this point, it should be stressed, that there is no overhead once the requested classes are loaded, since the classes are cached and no particular lookup into the introduced configuration files is needed. On the other hand, the first loading request of a class imposes some overhead regarding the lookup which class space may handle the class and which class collection contains the requested class. But this can be minimized using regular expressions to keep the configuration of class collections small. Furthermore, the hierarchy of class spaces will usually be rather tiny, typically less than five levels. Thus, only few delegations between childs and their parents occur. And in the case a parent can not load a class, it returns control to its childs using boolean return values instead of exceptions; hence speeding up the delegation. Compared to other middleware solutions, our approach does not force developers to use a new programming model or technique. It does not propose how to add new functionality or aspects to a Java program like *Aspect-Oriented Programming (AOP)* [23], but rather it presents a new way to *deploy*, *compose* and *host* Java applications transparently with regard to their implementation.

4 Application of the Approach

The presented approach has been developed as part of the *middleware platform ODIN* [24], which is currently being used in several ongoing research projects like *netzspannung.org* [25,26] and *Awake* [27]. Its major goal is the provision of an open, multi-platform programming and runtime environment for distributed applications. It supports multi-application hosting within a single JVM, which improves the utilization of resources and eases the inter-application communication. For that purpose, it has to shield each hosted application against each other, but also to enable collaboration between them. The Java implementation of ODIN uses the presented approach to achieve these goals and its application is now presented in the following.

4.1 Java Application Middleware

The JRE contains a JVM, which enables users to run Java Applications on each supported platform, independent of the operating system. In this context, the native JVM is usually only able to host one application at the same time. But in contrast to that, the presented approach is used by ODIN to form a *Java Application Middleware (JAM)* that enables multi-application hosting within a single JVM.

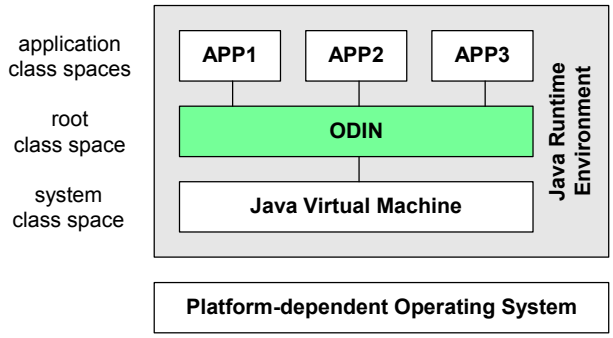

Fig. 12. Application Middleware

This is achieved by placing ODIN between the hosted applications and the JVM, as shown in fig. 12, and is also feasible for legacy applications without any modification of the related byte code. For this purpose, ODIN manages loaded classes using *Java Class Spaces (JCS)* as described in the presented approach above and intercepts class loading requests as follows. For running an application, the JAM has to be started first. It establishes a *system layer*, wherein all basic classes like *java.lang.** will be loaded into the related *system class space*. On top of this layer, the *ODIN framework layer* resides with its *root class space* that is created as a child class space of the *system class space*. It handles all framework related classes of ODIN. Afterwards, an *application class space* is created for each hosted application, in the example for *app1*, *app2*, and *app3* with the *root class space* as parent. Finally, the classes containing the method *main* of the applications are loaded using the user-defined class

loader associated with the corresponding application class space. Thus, the application class space represents the *initiating class loader* of the application classes; hence all subsequent class loading requests of the application are transparently handled by the class space as described in the presented approach. This can be done dynamically by requesting applications on the fly or statically as depicted in fig. 13 for *app1*, *app2* and *app3*. The parameters *classReg* and *classSpaceReg* define where to find the configurations files for *class collections* and *class spaces*.

```
java MainStarter -classReg=file:///etc/collections.xml\
-classSpaceReg=file:///etc/spaces.xml app1 app2 app3
```

Fig. 13. Starting the Application Middleware ODIN and the hosted applications

4.2 Distributed Application Systems

Generally, the origin JRE has not been designed for distributed scenarios, where applications are dynamically composed of classes, retrieved from remote repositories.

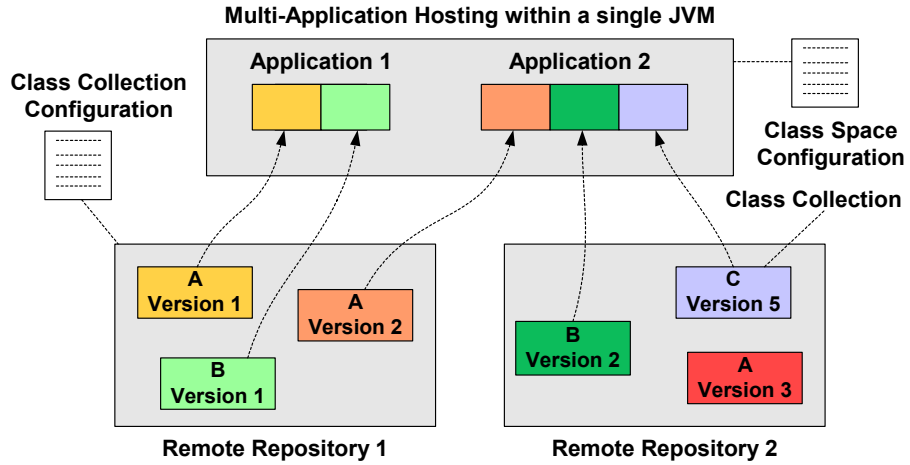

Fig. 14. Remote Repositories and Application Hosting

But this is quite important when distributed applications like *agents* are dynamically deployed on several computers, which are neither previously configured to host the application nor contain the appropriate classes to run the application. But since different applications request possibly different variants of the same class, each variant has to be installed and selectable by each application individually. Therefore, we use the presented approach of *Java Class Collections (JCC)* in ODIN to define the *logical* grouping of Java classes required by each application, independently of the *physical* location of Java classes as shown in fig. 14. The *Java Class Spaces (JCS)*, wherein the applications should be hosted, are configured with the appropriate JCC, containing the correct variants of the required classes. With that, applications could request classes without any knowledge from which remote repository the classes are

actually retrieved. Thus, the *application configuration* is completely decoupled from the *platform configuration*. This is especially useful for deploying distributed applications, since one can specify the required classes of the application independent of the deployment scenario of the host, where it should run.

5 Conclusions

In this paper, we have presented a customizable application middleware, which enables the separate configuration of *application composition* and *application deployment*. Along with that, we have showed the customization of *application hosting*, managing different implementations of the same class concurrently and how to shield and share these classes for hosting multi-applications within a single JVM. In this context, we have introduced so called *class collections*, *class spaces* and *object spaces* which are heavily based on the decoration of Java classes with additional properties, but also on the managed separation of loaded classes and objects within the JVM. Finally, after the illustration of the implementation issues, we depicted the application of the approach in the middleware platform ODIN.

Based upon the presented approach, there are various issues for future work. Actually, we use static configuration files to configure the class spaces and the required class collections of an application. But this could be done dynamically, evaluating the related class files and considering the current configuration of the framework. Though, it is not difficult to handle that manually, it would help to integrate dynamically legacy applications coming from elsewhere, e.g. dynamically loaded from the Internet. Even more complex is the dynamic reconfiguration of the application environment, if new classes have to be loaded. As long as there are only few class spaces, one could easily track down the relationship between them and provide an appropriate configuration. However, an Internet platform hosting many dynamic services reaches very fast dozens of class spaces. This complicates their proper arrangement, which is important for shielding and sharing loaded classes and applications.

Another interesting issue is the *dynamic resolving* of missing classes. Beside of the *static prevention* used in an ordinary JVM, the class space could catch the *ClassNotFoundException* and try to locate the requested class *on the fly*, pretending the class has been found. This would open a new way how classes are resolved and loaded, e.g. one could interpose a *class loading interceptor* and defer the decision *which* implementation of the class is used until running the application.

6 Acknowledgements

The presented approach has been evaluated and used in the implementation of the Internet platform *netzspannung.org* [25]. The related project CAT [28] is funded by the German Federal Ministry for Education and Research and is conducted by the research group MARS from the Fraunhofer Institute for Media Communication, St. Augustin in cooperation with the University of Siegen, Germany.

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. Gong, L. Secure Java Class Loading. IEEE Internet Computing, Vol. 2, Nr. 6, pp. 56-61. 1998.
6. Expresso - Application Development Framework. http://www.jcorporate.com/html/products/expresso.html. 2002.
7. Jakarta Tomcat - Servlet Engine. http://jakarta.apache.org/tomcat/index.html
8. Java Web Start. http://java.sun.com/products/javawebstart. 2002.
9. Paal, S., Kammüller, R., Freisleben, B. Distributed Extension of Internet Information Systems. In Proc. of the 13th International Conference on Parallel and Distributed Computing and Systems (PDCS 2001). Anaheim, USA. IASTED 2001. pp. 38-43.
10. Apache Xerces - Java XML Parser. http://xml.apache.org/xerces2-j. 2002
11. SUN Java Development Kit. http://java.sun.com/products/j2se/1.4/index.html. 2002.
12. The Apache Software Foundation. Apache XML Cocoon. 2001. http://xml.apache.org/cocoon/xsp.html
13. Fayad, M. E., Schmidt, D. C., Johnson, R. E. Implementing Application Frameworks: Object-Oriented Frameworks at Work. John Wiley & Sons. 1999.
14. Lewis, T. Object Oriented Application Frameworks. Manning Publications Co. 1995.
15. Grosso, W. Java RMI. O'Reilly & Associates. 2001.
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. Latteier, A. Bobo and Principia: An Object-Based Web Application Platform. WebTechniques, February 1999.
19. Little, M. C., Wheater, S. M. Building Configurable Applications in Java. Proc. of the 4th International Conference on Configurable Distributed Systems. Annapolis, Maryland. 1998. pp. 172-179.
20. Monson-Haefel, R. Enterprise Java Beans. O'Reilly & Associates. 2000.
21. Apache Server Framework Avalon. http://jakarta.apache.org/avalon/framework/index.html
22. Sun Open Network Environment (ONE). http://www.sun.com/sunone. 2002
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. Proc. of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
24. Open Distributed Network Environment. http://odin.informatik.uni-siegen.de
25. netzspannung.org, Communication Platform for Digital Art and Media Culture. http://netzspannung.org
26. 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.
27. AWAKE - Networked Awareness for Knowledge Discovery. 2002. http://awake.imk.fraunhofer.de
28. 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