A method is disclosed for reducing memory consumption in a run time environment by reducing copies of code in memory, wherein the method limits, based on the delegation, identical resources in memory and reduces memory consumption by not creating class loaders to load resources that are already loaded by existing class loaders.
Construction of Delegation Classloaders

302 Initiate classloader creation

304 Delegation required?

308 Yes

310 Create Delegation Classloader (DCL)

312 Identify required libraries and generate list of delegate loaders

314 For every delegate

314 Determine whether existing loader available for each required library

316 Return reference to loader for required library

316 Delegate to referenced loader

FIG. 3
FIG. 4

Store

Loader object

DCL_a

CL_b

GCL_c

SCL_d

Key

a.jar; 3241.1.3

b.jar; 2442.0

-a.jar, b.jar, c.jar

a.jar, WEB-INF/classes
Construction of Delegation Classloaders

1. Initiate classloader creation
2. Check for delegation required?
   - Yes: Create Delegation Classloader (DCL)
   - No: Create non-delegating loader
3. Identify required libraries and generate list of delegate loaders
4. Load library using non-delegating classloader
5. Check for circular reference
   - Yes: Return reference to loader for required library
   - No: Delegate to referenced loader

FIG. 5
FIG. 6

602
BPDelegationClassLoader

1
604
BytesProvider

1

1...

Classloader

606
MEMORY EFFICIENT CLASSLOADING TO REDUCE THE NUMBER OF CLASS COPIES

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a continuation of pending U.S. application Ser. No. 10/319,875 filed Dec. 13, 2002. This application additionally claims priority from U.S. Provisional Application No. 60/341,089, filed Dec. 13, 2001, which is hereby incorporated by reference in its entirety.

FIELD OF THE INVENTION

[0004] This invention relates to the field of computer programming.

BACKGROUND OF THE INVENTION

[0005] Object-oriented programs, such as those written in Java, often incorporate classes or other resources that reside in existing libraries. The Java environment provides a means for loading such libraries on demand at runtime.

[0006] Runtime loading of library code in Java is achieved using software objects known as "classloaders." Classloaders are adapted to obtain referenced library code, organize the code into classes (if necessary), and make the code available to software that references the library.

[0007] When library code is needed multiple times by a Java program, its classloader loads the library only once and caches it for future requests. A single Java runtime environment, however, often runs multiple applications at the same time and may therefore need to load the same library several times. This can lead to unnecessary duplication of the library code.

COMPUTER PROGRAM LISTING

[0002] Applicants submit herewith a software code on compact disc and in hard copy which is hereby incorporated by reference for each of its teachings and embodiments.

Name | Size (bytes) | Created
--- | --- | ---
AbstractBytesProvider.java | 8,644 | Dec. 12, 2002
AbstractBytesProviderDecorator.java | 2,528 | Dec. 12, 2002
AbstractBytesProviderKey.java | 782 | Dec. 12, 2002
BytesProvider.java | 1,649 | Dec. 12, 2002
BytesProviderClassLoader.java | 2,191 | Dec. 12, 2002
BytesProviderDelegationClassLoader.java | 7,503 | Dec. 12, 2002
BytesProviderFactory.java | 27,206 | Dec. 12, 2002
BytesProviderKey.java | 4,226 | Dec. 12, 2002
BytesProviderPreferenceDelegationClassLoader.java | 4,243 | Dec. 12, 2002
BytesProviderStore.java | 5,176 | Dec. 12, 2002
BytesProviderUtil.java | 1,245 | Dec. 12, 2002
CachedJarKey.java | 5,018 | Dec. 12, 2002
ClassLoaderFactory.java | 5,018 | Dec. 12, 2002
ClassLoaderMessages.java | 8,076 | Dec. 12, 2002
ClassLoaderMessages.properties | 6,858 | Dec. 12, 2002
Configuration.java | 9,659 | Dec. 12, 2002
DorkCacheJarBytesProvider.java | 6,030 | Dec. 12, 2002
ExplodedLocalStorage.java | 3,427 | Dec. 12, 2002
ExtraJarDependencies.java | 3,829 | Dec. 12, 2002
FileTreeList.java | 1,279 | Dec. 12, 2002
GroupedWrapperBytesProvider.java | 4,319 | Dec. 12, 2002
GroupKey.java | 3,706 | Dec. 12, 2002
Handler.java | 1,828 | Dec. 12, 2002
JarCacheTypeSelector.java | 695 | Dec. 12, 2002
JarKey.java | 1,371 | Dec. 12, 2002
JarProcessor.java | 643 | Dec. 12, 2002
JarProcessorCallback.java | 317 | Dec. 12, 2002
JarProcessorMessages.java | 33 | Dec. 12, 2002
JarProcessorMessages.properties | 7,935 | Dec. 12, 2002
JarInputStreamHelper.java | 3,824 | Dec. 12, 2002
JaxURLConnection.java | 9,313 | Dec. 12, 2002
LocalStorage.java | 644 | Dec. 12, 2002
ManagerClassLoader.java | 686 | Dec. 12, 2002
MemoryCacheJarBytesProvider.java | 1,540 | Dec. 12, 2002
NullJarProcessorCallBack.java | 1,995 | Dec. 12, 2002
ParentClassLoaderFactory.java | 2,180 | Dec. 12, 2002
PlainLocalStorage.java | 2,217 | Dec. 12, 2002
PreferenceDelegationClassLoader.java | 4,189 | Dec. 12, 2002
SubdirectoryWrapperBytesProvider.java | 643 | Dec. 12, 2002
StubJarKey.java | 713 | Dec. 12, 2002
URLBytesProvider.java | 340 | Dec. 12, 2002
URLClassLoaderCE.java | 2,711 | Dec. 12, 2002
URLKey.java | 488 | Dec. 12, 2002
URLs.java | 1,289 | Dec. 12, 2002

COPYRIGHT NOTICE

[0003] A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.
time, each of which is typically provided with its own class loader. Moreover, larger applications may instantiate multiple class loaders for different program components. When two applications or components require the same library code, their class loaders each load a copy of it into memory. This duplication requires additional memory and may significantly affect system performance.

A typical prior art scenario illustrating this problem is depicted in FIG. 1. Shown in FIG. 1 is a Java runtime environment 100 that contains an application server 102 built on top of an object request broker (ORB) 104. Several applications 106 are deployed in application server 102. One or more of applications 106, such as application 106c in the particular example of FIG. 1, may also be containers for other applications 108. Also shown in FIG. 1 is a security provider 110 running in the same Java runtime environment 100, and built on top of an encryption engine 112.

As is further shown in FIG. 1, each application 102-112 in environment 100 is provided with its own class loader 114 adapted to serve resources required by its respective application. Moreover, in the exemplary scenario of FIG. 1, each application 102-112 references the same XML parser library, xmlparser.jar. Consequently, at run time, multiple copies of this XML parser library are loaded into memory, one by each class loader 114. As noted, this duplication consumes memory and negatively affects performance.

One possible approach for solving this problem might be to provide a single class loader for two or more applications. The shared class loader could examine the source code of both applications and identify all libraries referenced by both. It could then load a single copy of such common libraries for use by both applications.

This approach, however, suffers from several drawbacks. First, if the first and second applications require different versions of the same library, the class loader may load a version of the library not compatible with one of the applications.

Second, sharing a class loader in this way permits each application to load all classes and resources of the other application. Consequently, if one of the applications is malicious, it may harm the second application by, for example, creating instances, making invocations, or examining data of the second application.

SUMMARY OF THE INVENTION

The present invention provides for a computer-based system for reducing memory consumption in a run time environment by reducing copies of code in memory, wherein the system comprises: a processor; and computer storage storing computer readable program code that is executable by the processor to reduce memory consumption in a run time environment by reducing copies of code in memory, and wherein the computer storage comprises computer readable program code for: (a) loading an application in the run time environment; (b) implementing a plurality of class loaders in memory, each class loader in the plurality of class loaders being associated with a resource; (c) implementing a store in memory comprising a plurality of references to existing class loaders stored in the memory, (d) implementing a first object adapted to identify at least one resource required by the application, the first object creating a second object adapted to retrieve the at least one resource, the second object determining if an existing class loader is available for loading the at least one resource, and the second object determining if a reference exists to the existing class loader for the at least one resource, and the second object delegating task of loading the at least one resource to the existing class loader based on the determined reference; and (e) limiting, based on the delegation, identical resources in memory and reducing memory consumption by not creating class loaders to load resources that are already loaded by existing class loaders.

The present invention also provides a computer-based method for reducing memory consumption in a run time environment by reducing copies of code in memory, wherein the method comprises: (a) loading an application in the run time environment; (b) implementing a plurality of class loaders in memory, each class loader in the plurality of class loaders being associated with a resource; (c) implementing a store in memory comprising a plurality of references to existing class loaders stored in the memory, (d) implementing a first object adapted to identify at least one resource required by the application, the first object creating a second object adapted to retrieve the at least one resource, the second object determining if an existing class loader is available for loading the at least one resource, the second object determining if a reference exists to the existing class loader for the at least one resource, and the second object delegating task of loading the at least one resource to the existing class loader based on the determined reference; and (e) limiting, based on the delegation, identical resources in memory and reducing memory consumption by not creating class loaders to load resources that are already loaded by existing class loaders.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a runtime environment using prior art class loaders;

FIG. 2 illustrates a preferred embodiment of the present system and method;

FIG. 3 illustrates a preferred embodiment of the construction of a delegation class loader in the present system and method;

FIG. 4 illustrates a preferred embodiment for implementing a store in the present system and method;

FIG. 5 illustrates a preferred embodiment of the construction of a delegation class loader in the present system and method;

FIG. 6 illustrates a preferred embodiment of the relationship between a bytes provider delegation class loader and a bytes provider in the present system and method;

FIG. 7 illustrates a preferred embodiment of a bytes provider hierarchy; and

FIG. 8 illustrates a preferred embodiment for implementing a store in the present system and method.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

One preferred embodiment of the present invention is described in connection with an exemplary Java runtime environment such as the environment shown in FIG. 2. As shown in FIG. 2, a Java runtime environment 200 preferably contains an application server 202 built on top of an object request broker (ORB) 204. Several applications 206 are deployed in application server 202. One or more of applications 206, such as application 206c in the particular example of FIG. 2, may also be containers for other applications 208.
Also shown in FIG. 2 is a security provider 210 running in a Java runtime environment 200, and built on top of an encryption engine 212.

[0024] Each application 202-212 is preferably provided with a classloader 214. In a preferred embodiment, classloaders 214 are of a construction that differs from the construction of prior art classloaders, such as classloaders 114 in FIG. 1. More specifically, classloaders 214 preferably belong to a novel class of classloaders referred to herein as delegation classloaders.

[0025] As described in more detail below, a delegation classloader is adapted to identify libraries referenced in a piece of code and delegate the task of loading those libraries to a second object. The second object may be a delegation classloader or other loader, as described below. Illustratively, as shown in FIG. 2, each classloader 214 is adapted to delegate the task of loading xmlparser.jar to a single shared classloader 216. As a result, only one copy of xmlparser.jar is loaded, thus saving memory.

[0026] In a preferred embodiment, each shared classloader 216 is preferably an "atomic" classloader, i.e., adapted to look at one library or a small set of libraries (e.g., one library and all libraries that it references). This makes it possible to reuse shared classloaders 216 since they provide only a relatively small set of resources and therefore do not expose application-specific resources to other applications.

[0027] FIG. 3 illustrates a preferred embodiment for the construction of a delegation classloader. To facilitate understanding of the preferred embodiments, aspects of FIG. 3 are described in connection with the specific scenario of FIG. 2. It should be recognized, however, that the delegation classloaders and other objects described below are capable of application in a wide variety of runtime environments.

[0028] Turning to FIG. 3, at 302, classloader creation is initiated. At 304, the application or other resource to be loaded is examined to determine whether or not it includes references to one or more libraries. If the application or other resource has no library dependencies, a delegation classloader is not required and a non-delegating loader is created to load the application or other resource (step 306). Otherwise, at 308, a delegation classloader 300 is created.

[0029] At 310, delegation classloader 300 identifies each library referenced by the application or other resource and adds the referenced libraries to a list. For example, in the illustrative scenario of FIG. 2, classloader 214a examines WebApp 206a to determine that it includes a reference to xmlparser.jar, and adds xmlparser.jar to its list.

[0030] Delegation classloader 300 is preferably adapted to distinguish between resources that it will load itself and other resources with respect to which it will delegate that task. For example, a delegation classloader tasked with loading an application may itself load the application but delegate the task of loading any libraries referenced by the application. Alternatively, the programmer may specify the resources to be loaded directly by the delegation classloader and the resources for which the delegation classloader should delegate loading to another loader. In the illustrative scenario of FIG. 2, classloader 214a preferably itself loads webapp.war. By contrast, with respect to libraries referenced by WebApp 206a (e.g., xmlparser.jar), classloader 214a preferably delegates the task of loading such libraries to another delegation classloader as described below.

[0031] At 312, the delegation classloader determines whether an existing loader is available to load each library in the list. In a preferred embodiment, this determination is made by consulting a store of existing loaders. One preferred embodiment for implementing such a store is shown in FIG. 4, described below. More generally, the determination may be made by communicating directly or indirectly (e.g., via an intermediary adapted to facilitate or broker the communication) with other loaders to determine whether a loader for the library is available.

[0032] As shown in FIG. 4, a store 400 preferably comprises a number of entries 402. Each entry preferably comprises a key 404 that uniquely identifies a library and a reference 406 to a loader for the library. In a preferred embodiment, the key may include a plurality of elements including a name 408, a size 410, and a version number 412.

[0033] In an alternative preferred embodiment, the delegation classloader may directly determine whether or not an instance of each library in the list (or any required resource generally) is available in memory. In a preferred embodiment, this determination may be made by consulting a store that includes available libraries (and resources) or references to such libraries (and resources). More generally, the determination may be made by communicating directly or indirectly with other loaders or system entities to locate and use an available instance of a required library (or resource) in memory.

[0034] Returning to FIG. 3, at 314, for each library in the list with an existing loader, a reference to the loader for the resource is returned. As shown in FIG. 4, in some cases the existing loader may be another delegation classloader (e.g., entry 418). In other cases, the existing loader may be some other kind of loader (e.g., entry 420). At 316, delegation classloader 300 delegates the task of loading the library to the existing loader for the resource. For example, in the illustrative scenario of FIG. 2, if a loader for xmlparser.jar had previously been created, classloader 214a would find a reference to this loader in store 400 and delegate the task of loading xmlparser.jar to it.

[0035] By contrast, for each resource in the list without an existing delegation classloader, delegation classloader 300 returns to step 302 to initiate creation of an appropriate classloader to load the library. For example, in the illustrative scenario of FIG. 2, if no loader for xmlparser.jar had previously been created, delegation classloader 214a creates one and delegates to it the task of loading xmlparser.jar.

[0036] To maximize efficient sharing of libraries, the classloaders created in this step preferably load only a single library. In some preferred embodiments, however, the system may be configured to allocate loading of some set of libraries to a single classloader. This may be advantageous where, for example, one library includes references to several others or where some small set of libraries are otherwise related.

[0037] In a preferred embodiment, the present system and method may comprise a number of different delegation classloaders each adapted to load a different library format. Each library preferably comprises a collection of resources which may, for example, comprise one or more classes, Web pages, or other resources that may be served by a classloader. Suitable delegation classloaders may be provided for Java archives, Web archives, or any other desired format. Delegation classloader 300 preferably identifies the format of the required library and creates a suitable delegation classloader (or other loader) for loading that format.

[0038] A reference to the new delegation classloader for the referenced resource is preferably added to store 400 (in
embodiments that comprise such a store). Consequently, the next time a delegation classloader consults the store, the reference will be found, thus avoiding the need to load a second copy of the library.

[0039] As demonstrated by FIG. 3, in a preferred embodiment, the delegation classloader model of the present system and method is recursive. Thus, for example, a delegation classloader created to provide a particular library may itself examine the library for any dependencies and identify or create any necessary delegation classloaders or other loaders for libraries referenced by the library that it looks at.

[0040] In some cases, two or more libraries required by a program may include circular references to each other. In such cases, the recursive nature of the model shown in FIG. 3 may lead to an infinite loop in which, for example, each library’s delegation classloader repeatedly delegates to the other. A preferred embodiment for handling such circular references is described in connection with FIG. 5.

[0041] Shown in FIG. 5 is an additional step 518 in which the delegation classloader checks for circular references. If a circular reference is identified, the resource is preferably loaded using a non-delegating classloader (step 520). In other respects, the steps shown in FIG. 5 are the same as in FIG. 3.

[0042] As noted above, delegation classloaders are preferably provided for any required library formats. In a preferred implementation of the present system and method which simplifies classloader construction, format-specific aspects of the classloader function are allocated to an abstraction layer that comprises objects referred to herein as bytes providers. The resulting simplified delegation classloaders are referred to herein as bytes provider delegation classloaders or BP delegation classloaders.

[0043] FIG. 6 illustrates the relationship between bytes providers and BP delegation classloaders in one preferred embodiment. As shown in FIG. 6, a BP delegation classloader 602 preferably creates an appropriate bytes provider 604 adapted to read specified resources and provide them to BP delegation classloader 602.

[0044] As above, BP delegation classloader 602 preferably delegates the task of loading libraries referenced in the application or other resource for which it is directly responsible to a distinct software object, such as another BP delegation classloader 606. In a preferred embodiment, each BP delegation classloader 606 is an atomic classloader responsible for a single library or small set of libraries. Each BP delegation classloader 606 is also preferably paired with an associated bytes provider adapted to look at the library or small set of libraries and provide bytes from the library or libraries to BP delegation classloader 606.

[0045] A preferred embodiment of a bytes provider hierarchy is shown in FIG. 7. As shown in FIG. 7, a hierarchy 700 preferably comprises a bytes provider interface 702 which defines methods for obtaining certain information from a bytes provider. Interface 702 preferably defines a getClassLoader method 704 that returns the bytes provider’s associated BP delegation classloader and a getURL method 706 that returns the URL the bytes provider looks at.

[0046] Hierarchy 700 further preferably comprises an abstract bytes provider 708 which extends interface 702. Abstract bytes provider 708 preferably defines a plurality of methods for reading data out of a bytes provider. In a preferred embodiment, these include a getResourceAsBytes method 710, a getResourceAsStream method 712, and a getResourceURL method 714.

[0047] Hierarchy 700 further preferably comprises a plurality of specific bytes providers 716-720. These include a URLLBytesProvider 716 that looks at a URL, a DiskCacheJarBytesProvider 718, and a MemoryCacheJarBytesProvider 720. Bytes providers 718 and 720 preferably look at JAR files, and are adapted to cache a JAR by loading it into memory or expanding it on disk, respectively. In a preferred embodiment, the decision whether to cache a JAR in memory or on disk is preferably based on an evaluation of the cache watermark settings and available memory.

[0048] In a preferred embodiment, additional bytes providers suitable for reading any desired format may be provided. These may be necessary, for example, in cases where a loading mechanism is required that is not provided by any of the bytes providers shown in FIG. 7, such as loading out of an EARSCO directory structure.

[0049] Also shown in FIG. 7 are two types of wrapper bytes providers including a SubdirectoryWrapperBytesProvider 726 and a GroupedWrapperBytesProvider 728. SubdirectoryWrapperBytesProvider 726 preferably provides a wrapper around two or more bytes providers in a single directory, for example, one bytes provider that looks at a directory’s root and a second one that looks at a subdirectory. The SubdirectoryWrapperBytesProvider 726 may thus be used to provide a single interface to all (or some subset of) libraries in a directory.

[0050] GroupedWrapperBytesProvider 728 is preferably adapted to wrap around an arbitrary set of bytes providers and provide a single interface to all libraries looked at by those bytes providers. This may be useful, for example, if the bytes providers serve as one logical unit or they are associated with a single classloader. In addition, by creating a GroupedWrapperBytesProvider 728 around all bytes providers, it is possible to effectively disable delegation classloading and provide a single interface for loading any library.

[0051] In operation, when a BP delegation classloader wishes to delegate the task of loading a resource to another classloader, it determines whether a bytes provider or other loader for the resource exists. As above, this step may be achieved by consulting a store or by alternative means.

[0052] If an existing bytes provider for the resource is found, the calling BP delegation classloader invokes the bytes provider’s getClassLoader method to obtain the bytes provider’s BP delegation classloader. It then delegates the task of providing the resource to the BP delegation classloader. Alternatively, if another loader for the resource is found, the BP delegation classloader delegates the task of loading the resource to the loader.

[0053] One preferred embodiment for a store 800 is shown in FIG. 8. As shown in FIG. 8, store 800 preferably comprises a plurality of entries 802. Each entry preferably comprises a key 804 and a reference 806 to a bytes provider or loader for the library or libraries identified by the key. As will be recognized, store 800 may include keys for single-library bytes providers, as shown at entry 808, and for GroupedWrapper bytes providers and SubdirectoryWrapper bytes providers as shown at entries 812 and 814, respectively. As noted, store 800 may also comprise references to other loaders as shown at entry 810.

[0054] It should be noted that, although the concept of wrapper objects for multiple libraries has been described in connection with the bytes provider implementation, analogous wrappers may be provided in alternative implementations that do not employ bytes providers. Thus, as shown in
FIG. 4, store 400 may, for example, include references to grouped wrapper delegation classloaders (e.g., entry 414) and subdirectory wrapper delegation classloaders (e.g., entry 416).

[0055] It should also be noted that although the above embodiments have been described primarily in terms of a Java runtime environment, those skilled in the art will recognize that the principles of the present invention are capable of application in other runtime environments such as Microsoft’s .Net runtime environment. For example, as understood in the art, .Net uses the term assembly resolver to describe a software component that is functionally analogous to a classloader in Java. For purposes of the present application, applicant intends that terms such as classloader, class, loading, and library be understood in their functional sense, and be construed to cover their respective analogs in other runtime environments.

[0056] While the invention has been described in conjunction with specific embodiments, it is evident that numerous alternatives, modifications, and variations will be apparent to those persons skilled in the art in light of the foregoing description.

1-29. (canceled)

30. A computer-based system for reducing memory consumption in a run time environment by reducing copies of code in memory, comprising:
   a. processor; and
   computer storage storing computer readable program code that is executable by said processor to reduce memory consumption in a run time environment by reducing copies of code in memory, said computer storage comprising:
   computer readable program code loading an application in said run time environment;
   computer readable program code implementing a plurality of class loaders in memory, each class loader in said plurality of class loaders being associated with a resource;
   computer readable program code implementing a store in memory comprising a plurality of references to existing class loaders stored in the memory;
   computer readable program code implementing a first object adapted to identify at least one resource required by said application, said first object creating a second object adapted to retrieve said at least one resource, said second object determining if an existing class loader is available for loading said at least one resource, said second object determining if a reference exists to said existing class loader for said at least one resource, and said second object delegating task of loading said at least one resource to said existing class loader based on said determined reference; and
   computer readable program code limiting, based on said delegation, identical resources in memory and reducing memory consumption by not creating class loaders to load resources that are already loaded by existing class loaders.

31. The computer-based system of claim 30, wherein said first object is a delegation class loader.

32. The computer-based system of claim 30, wherein said second object is a delegation class loader.

33. The computer-based system of claim 30, wherein said at least one resource is a single library.

34. The computer-based system of claim 30, wherein said at least one resource is a single library and its direct dependencies.

35. The computer-based system of claim 30, wherein said at least one resource is a single library and its related libraries.

36. The computer-based system of claim 30, wherein said at least one resource includes one or more classes.

37. The computer-based system of claim 30, wherein said at least one resource includes one or more web pages.

38. The computer-based system of claim 30, wherein said at least one resource is a .Net assembly.

39. The computer-based system of claim 30, wherein said run time environment is a virtual machine environment.

40. A computer-based method for reducing memory consumption in a run time environment by reducing copies of code in memory, steps of said method executed on a computer, said method comprising:
   loading an application in said run time environment;
   implementing a plurality of class loaders in memory, each class loader in said plurality of class loaders being associated with a resource;
   implementing a store in memory comprising a plurality of references to existing class loaders stored in the memory;
   implementing a first object adapted to identify at least one resource required by said application, said first object creating a second object adapted to retrieve said at least one resource, said second object determining if an existing class loader is available for loading said at least one resource, said second object determining if a reference exists to said existing class loader for said at least one resource, and said second object delegating task of loading said at least one resource to said existing class loader based on said determined reference; and
   limiting, based on said delegation, identical resources in memory and reducing memory consumption by not creating class loaders to load resources that are already loaded by existing class loaders.

41. The computer-based method of claim 40, wherein said first object is a delegation class loader.

42. The computer-based method of claim 40, wherein said second object is a delegation class loader.

43. The computer-based method of claim 40, wherein said at least one resource is a single library.

44. The computer-based method of claim 40, wherein said at least one resource is a single library and its direct dependencies.

45. The computer-based method of claim 40, wherein said at least one resource is a single library and its related libraries.

46. The computer-based method of claim 40, wherein said at least one resource includes one or more classes.

47. The computer-based method of claim 40, wherein said at least one resource includes one or more web pages.

48. The computer-based method of claim 40, wherein said at least one resource is a .Net assembly.

49. The computer-based method of claim 40, wherein said run time environment is a virtual machine environment.

* * * * *