Java™ Product Versioning Specification

November 30, 1998

Evolution in open distributed systems needs to be managed carefully because correct operation depends on a number of dependencies between packages. Changes within a distributed system can have a significant impact on many groups of individuals, including users, support organizations, web administrators, and developers. Packages within a distributed system need to operate correctly with only partial knowledge about the state of the whole system. The difficulty level increases because the packages of the system must be able to evolve at different rates. Evolution in such a system is made possible by explicitly managing the dependencies between the packages using techniques of object oriented design to govern how individual packages evolve. The Java language defines packages that are a natural for the consistent unit of update, packages that expose only public interfaces and that consume only public interfaces of other classes.

Introduction

In any system, support must be provided for the system to evolve over time. Most existing systems have conventions and mechanisms that specify how change is accommodated. These systems have been based on the model where software programs are installed on a computer. Typically, developers specify what versions of other packages are required and the installation process verifies and configures the system.

In open distributed systems, however, the static assumptions of existing systems do not work. Evolution is more difficult because it is not possible to control how or when packages change. Correct operation depends on a greater number of dependencies between packages. There is a real need for an updated set of conventions and mechanisms which specify how packages of a system should evolve so that the goal of an open reliable scalable distributed system can be achieved.

This document specifies the following:

- How classes, resources, and files that make up a Java package are versioned. Packages define a consistent unit that can be developed, packaged, verified, updated, and distributed. Per package manifest information identifies the contents of the package.
- Products are distributed by putting packages into archive files. Archives include a manifest, to identify the product version and packages it contains.
- The standards and conventions used by developers and administrators to build and deploy products that operate reliably as their packages and the packages they depend on are upgraded.

Requirements

Changes within a distributed system have a significant impact on the following groups:

- End Users
- Product Support Organizations
- Webmasters and Administrators
- Product Developers

Each group has different requirements on network-deployed products that evolve over time.

End Users

End users need to feel confident that Java-based products will become increasingly reliable and compatible over time. Their reluctance to upgrade must be addressed by building confidence in the "Write Once, Run Anywhere" philosophy. With Java, it should no longer be the norm that users will complain "If I upgrade, it will break something" or "I won't be able to read or write data that others can use."

- Users need to know that upgrading will neither break other programs nor will it obsolete existing data or produce data unusable by others.

- At the simplest level, users want to know if the features they need are in the product version they have and what version to ask for to get particular features.
- More knowledgeable users keep track of what bugs are associated with a specific product version, so they can work around them or avoid them.

Product Support Organizations

Product support organizations rely on being able to correctly and easily identify the product that is being used, the environment in which it is being used, and the integrity of the product packaging.

- Databases of known problems and solutions are indexed by product identification information.
- The interoperation of products and packages can introduce new kinds of problems and require all of the packages in a system to be identified. Problems can originate from public interfaces that are under-specified, from implementations that do not conform to the specification, or from clients that use implementation-specific details that are not part of the specification.

Webmasters and Administrators

Webmasters, administrators, and service providers need a reliable and supportable way in which to deploy applications for their clients via the web or network filesystems.

- The staff of these organizations must be able to support their sites, identifying problems with specific packages and interactions between packages.
- Site configuration must be able to support the scaling up of sites with automated site management tools.
- Installing updating packages must not present a risk to the correction operation of existing packages or active users.

Product Developers

Product developers need to know how to write and deploy applications and libraries that satisfy the requirements of users, administrators, and support personnel. They must be able to make products and packages that can do the following:

- Operate correctly in the open dynamic environment of the web
- Be upgraded without breaking compatibility with clients
- Take advantage of upgrades in the packages on which they rely
- Take advantage of their packages' dynamic extensions
- Identify the packages they rely on for reporting of problems
- Be packaged to support the needs of users, webmasters, and support organizations
- Have known packages and combinations that satisfy the auditing and security requirements appropriate for the application and organization

Problems of Evolution in Distributed Systems

Problems can occur in open distributed systems when packages evolve and are frequently updated. If the specified behavior inherent in the use of public interfaces is not maintained, the system can fail in unexpected ways. Open systems are comprised of many packages from different companies and organizations. These organizations operate asynchronously, introducing and upgrading their products according to their own schedules. Distribution of upgraded products takes time and adoption is not universal.

In Java, the components of local and distributed systems rely on the public interfaces and contracts for the behavior of other packages. Those packages will themselves evolve over time. In order for a package to operate correctly, packages that it depends upon must continue to provide the expected behavior even if those packages have been updated.

Only partial consistency is possible in distributed systems, since it is impossible to have knowledge of the entire state of the system. Each process and each package of the system has its own partial view of the current state of the system, accumulated incrementally by requesting information from other parts of the distributed system. Each piece of information, whether from an applet that was started, a class that was loaded, a remote method invoked, or a web page retrieved, must be treated carefully so that it can be used consistently with the rest of that partial view.

Several types of errors can result from inconsistencies in the classes that are loaded: class verification errors, classes compute incorrectly but without recognizable errors, or user requested functions that exhibit arbitrary failures.

These problems can occur in the following typical scenarios:

- A running applet has loaded only some of its classes when the web server is updated with a newer version. When the applet incrementally loads additional classes, these classes could be inconsistent with ones that have already been loaded.
- An application using libraries from multiple websites has loaded only some of the classes it needs. If the libraries are updated, there is a potential for incompatibilities that either the applet or the user needs to detect.
- A running application or applet makes an RMI call that returns an object which the class needs to be loaded. The class that is loaded could be inconsistent with other already loaded classes.
- A running application or applet makes a RMI call that returns an object that is for a newer or an older version of the class.
- Bugs exist in a library. If the clients have worked around the bug, a cascade of problems could be introduced when the bug is fixed.

These problems cannot be prevented or solved directly because they arise out of inconsistencies between dynamically loaded packages that are not under the control of a single system administrator and so cannot be addressed by current configuration management techniques.

Design for Evolution

The key to dealing with these problems and meeting the requirements stated above is the careful design of the packages and packaging of the system so that they may be updated, distributed, and loaded in consistent units. Typical to mass produced products is the notion of the field replaceable unit. It is the smallest unit of a product that can be identified with a specification, a supplier, can be distributed and redistributed, and can be replaced if faulty. This same model is used for software distribution, products have a name, a version number, adhere to one or more specifications, are distributed on the network or CD-ROM and its problems can be reported to support organizations. These packages are the smallest unit that can be distributed, used, validated and replaced or upgraded when necessary. Packages can be assembled with other packages and each package can still be identified, verified, and distributed.

The Java language-based package mechanism fits well with the idea of a replaceable unit. Java packages expose only public interfaces and use only the public interfaces of other packages. The JavaTM Language Specification defines the approaches for compatible evolution of packages.

Java Language Specification on Backwards Compatibility

The Java Language Specification lays the groundwork for developing packages that can be expected to evolve gracefully over time. It defines how classes can change and still be backward compatible with other classes previously compiled and distributed. Essential to robust evolution is the stability of the public, protected, and package interfaces and behavior as the implementations evolve. It defines "compatible" changes as "those changes that do not change existing interfaces or behavior." Thus, if a class defines a method, and the method had a particular behavior, that same contract must be supported by the all later evolutions of the class. Detailed rules are given in Chapter 13 of the JavaTM Language Specification. One additional incompatible change has been added: it is incompatible to add methods to a public interface.

Although incompatible changes are not permitted, new or similar functionality can always be added in new or existing interfaces or classes.

By choosing the Java package as the unit of update, the package and private methods of the classes can change, thereby allowing flexibility in the implementation of the package while the public and protected classes and methods maintain the external interfaces and semantics.

Object Serialization Specification on Backwards Compatibility

Robust persistent storage and robust communication between the components is important to distributed systems. Components must be able to maintain persistent storage as they evolve, being able to evolve classes and yet have them read data previously written to storage. Components in a distributed system evolve at different rates and must still be able to communicate reliably.

Adhering to the compatibility requirements of object serialization allows newer and older versions to communicate in a predictable and consistent way. The details are in Chapter 5 of the JavaTM Object Serialization Specification.

Package Version Specification

There are several categories of artifacts that need to be identified including specifications, implementation, the Java Virtual Machine and Java Runtime Environment.

Specification Versioning

Open systems are based on the idea that a specification may have multiple implementations. Specifications evolve under the auspices of an organization or company. It is highly undesirable if a specification has multiple incompatible versions. Each version of a specification or implementation must evolve only into a single subsequent version. The philosophy of requiring specifications to be backward compatible allows specifications to be identified as supersets of the previous specification. Since there is a single sequence of version specifications they can meaningfully be identified by version numbers with specific semantics that imply the ordering. Specification version numbers use a Dewey decimal notation consisting of numbers separated by periods.

A specification is identified by the:

- Owner of the specification
- Name of the Specification
- Version number - *major.minor.micro*

Major version numbers identify significant functional changes.
Minor version numbers identify smaller extensions to the functionality.
Micro versions are even finer grained versions.

These version numbers are ordered with larger numbers specifying additions to the specification.

Virtual Machine Versioning

An implementation of the Java Virtual Machine should be identify both the specification and the implementation. These properties should be added to those already available using `java.lang.System.getProperties`.

java.vm.specification.version	i.e. 1.2
java.vm.specification.vendor	i.e. Sun Microsystems Inc.
java.vm.specification.name	i.e. Java™ Virtual Machine Specification
java.vm.version	i.e. Solaris 5.5 Native 1.0 build32
java.vm.vendor	i.e. Sun Microsystems Inc.
java.vm.name	i.e. Solaris 5.x JVM

These properties are accessed using the method `java.lang.System.getProperty` and each returns a string.

Version Identification of the Java Runtime

The requirement to identify the JavaTM Runtime is already partially met via the properties specified by the JavaTM Language Specification, §20.18.7 using `java.lang.System.getProperties`.

java.version	i.e. Solaris 1.2
java.vendor	i.e. Sun Microsystems Inc.

Currently these identify the implementation of the Java™ runtime and the core classes that are available. These properties do not identify the *Java™ Language Specification* version that this JDK™ implements.

The following additional properties are needed to identify the version of the Java™ Runtime Environment specification to which this implementation adheres:

java.vm.specification.version	i.e. 1.1
java.vm.specification.name	i.e. *Java™ Language Specification*
java.vm.specification.vendor	i.e. Sun Microsystems Inc.

These properties are accessed using the method `and return their values as strings.`

Package Versioning

Each JavaTM package consists of class files and optional resource files. The information needed to identify the contents of the package is stored with the package contents.

This specification applies to all packages, regardless of whether they are developed as a core package distributed with a JavaTM Runtime, a standard extension, an applet or application package.

Unlike version numbers for specifications, version information for implementations cannot be used to identify the package as being backward compatible with earlier versions. Package version numbers are present to identify differences between the specification and the implementation, i.e. bugs. New versions of implementations are specifically produced to remove (undesirable or incorrect) behavior and thus are intended not to be backward compatible. Therefore, package version strings can have any unique value and can only be compared for equality. For a complete explanation of this rationale, see Rationale for Limiting Implementation Version Numbers to Identity.

The following attribute names are defined for a package. The value of each attribute is a string:

Implementation-Title	Title of the package
Implementation-Version	Version number
Implementation-Vendor	Vendors company or organization
Specification-Title	Title of the specificationThese attributes are stored in the manifest and retrieved by programs using the API described below.
Specification-Version	Version number
Specification-Vendor	Vendors company or organization

These attributes are stored in the manifest and retrieved by programs using the `API described below.`

JAR Manifest Format

The current manifest format is extended to allow the specification of the attributes for package versioning information. A manifest entry should be created for each Java™ package. The name of the entry will be the directory within the archive that contains the package's class and resource files. For example:

```
Manifest-version: 1.0
Name: java/util/
Specification-Title: "Java Utility Classes"
Specification-Version: "1.2"
Specification-Vendor: "Sun Microsystems Inc.".
Implementation-Title: "java.util"
Implementation-Version: "build57"
Implementation-Vendor: &quotSunMicrosystems. Inc."
```

To insert these attributes in the manifest, create a prototype manifest file and use JarTool's `-m` switch to merge the attributes into the manifest when the manifest is built. JarTool will be extended to browse and set the versioning attributes in the manifest.

How Users Know What is Running

When bugs occur, users must be able to report the identities of the packages in use. The application, applet, or browser must be able to expose the available information to the user, either on demand or when an error occurs. Available APIs can report the following information:

- What packages are loaded?

  The `package.getAllPackages` method returns a list of the active packages.

- What are the package versions?

  The `java.lang.Package` methods allow the attributes for names and versions to be examined, as listed in Package Versioning.

- What version of the Java™ Runtime is active?

  The `System.getProperties` method can be used to get the properties of this virtual machine, as listed in Version Identification of the JavaTM Runtime.

- What version of the Java™ VM is active?

  The `System.getProperties` method can be used to get the properties of this virtual machine, as listed in Virtual Machine Versioning.

Rationale for Limiting Implementation Version Numbers to Identity

Implementations evolve independently over time to fix bugs, to improve performance, or to add new functions that are called for by more recent revisions of the specifications. Packages implement specifications and must identify which version of each specification they implement. Interactions occur between packages through their public and protected interfaces and classes only. The public API and behavior must remain stable over time, so changes can be allowed in the implementation of one package without affecting the behavior of another package.

If the classes of a package always faithfully implemented the specification, it would be sufficient just to identify the specification. Since in the real world this rarely happens, packages need to identify themselves so that bugs can be reported against the packages that may have contributed to the problem.

There is a significant tendency to try to attach some significance to version identifiers of implementations. If the purpose is to allow the tracking of bugs then a unique number is sufficient. It is also sufficient for a client package to workaround a bug in a particular version of a vendors package since that version can be tested for and the bug avoided.

However, many additional problems can occur when one package attempts to work around bugs in other packages. They need to identify behavior that is not part of the specification and may try to use behavior that is only part of one implementation. Such implementation specific behavior cannot be relied upon to be in any particular version other than the one(s) seen and tested by the developer.

A bug first appears in some version of a vendors package and may or may not continue to be a problem in subsequent versions. If the client of the buggy package uses a workaround based on version numbers, it could correctly work around the bug in the specific version. Now, if the buggy package was fixed, how would the client package know whether the bug was fixed or not? If it assumed that higher versions still contained the bug, it would still try to work around the bug. The workaround itself might not work correctly with the non-buggy package. This could cause a cascade of bugs caused by fixing a bug. Only the developer, through testing with a new version, can determine whether or not the workaround for a bug is still necessary or whether it will cause problems with the correctly behaving package. The developer only knows that the bug exists in a particular individual versions.

CONTENTS | PREV