Semantic Versioning

Technical Whitepaper

Revision 1.1
December 2018

Executive Summary

The OSGi Alliance recommends the versioning of exported packages. These versions should reflect the evolution of these packages. For this reason, a change in the first (major) part of the version signals backward incompatible changes to the artifacts. That is, going from version 1.5 to version 2 signals that another artifact compiled against 1.5 up to (but not including) version 2 of that initial artifact is *not* compatible with the new version of the initial artifact.

A package can contain an API. There are two types of clients for these API packages: *API consumers* and *API implementation providers*. A change in the second (minor) part of the version signals that the change is backward compatible with consumers of the API package but not with the providers of that API. That is, when the API package goes from version 1.5 to 1.6 it is no longer compatible with a provider of that API but consumers of that API are backward compatible with that API package.

The third and fourth part of the version (micro and qualifier) signal bug fixes and build stamp and have no effect on backward compatibility.

Importers of packages know their role (consumer or provider of an API) and can limit the import range of the package. For the previous example, a consumer would import [1.5,2) while a provider would import [1.5,1.6).

API designers must carefully evolve an API package, especially when consumers implement interfaces in this API package. Changes to such interfaces break binary compatibility for consumers and necessitate a major update of the version.

Introduction

Versioning is one of the unique selling points of OSGi; there are no other execution environments that have taken the evolution of the artifacts so seriously. In OSGi, all parts that can evolve over time (packages, bundles, fragments) are versioned. Not only did the OSGi Alliance provide the mechanisms for versioning, since its beginning, the OSGi Alliance has used strong semantic versions for all its specified packages.

However, despite this focus on versioning, the Core specification does not mandate a specific *version policy*. The Core specification only contains a recommendation of how to version the artifacts. At the time this recommendation was put in place, there was

insufficient experience with versions in an environment like OSGi. The different members of the OSGi ecosystem needed the freedom to version their artifacts as they wanted and therefore only a recommendation was put in the Core specification.

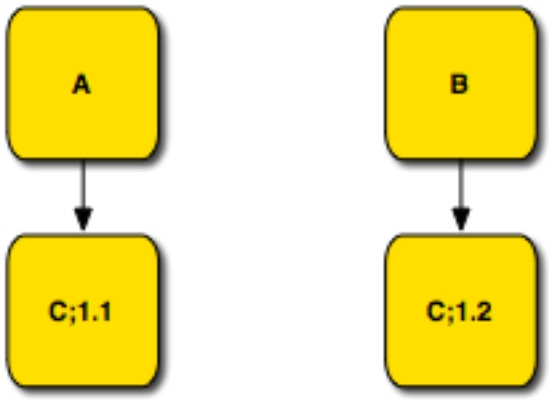
Over the past 10 years, the OSGi Alliance has used a semantic model for versioning artifacts. The current OSGi build contains around 130 projects that generate around 1300 bundles. In any release, a large number of disparate teams develop Reference Implementations and Compliance Tests that are integrated in this build system. Doing this work for almost ten years with a specification that has undergone significant enhancements has given the OSGi Alliance abundant experience with semantic versioning.

This document describes the lessons learned regarding versioning. Though not mandated by the specifications, it is strongly recommended to follow its guidelines so as to remain compatible with other parties using OSGi.

Background

The problem that versioning addresses is the independent evolution of dependent *artifacts*. In OSGi terms, a bundle can import a package exported by another bundle. The importer therefore has a dependency on the exporter. An *importer* will use a specific *exporter* during the compilation and build process. Ideally, this same exporter is used during the deployment process to create *fidelity* between the build and run time. This fidelity reduces the number of potential problems because many aspects are guaranteed to be verified during the build process.

However, in reality, the importer and exporter evolve independently. Creating a requirement for strict fidelity between importers and exporters requires a coordination between build and deploy time that is often not feasible in practice. Pure fidelity would require a complete rebuild of all artifacts when one artifact changes even slightly. This might be feasible when all software as sourced in-house and built together, but in today's world the projects that do not depend on open source projects or external suppliers are extremely rare. The key problem is that in a large system there will be many shared dependencies. For example, in a system with three bundles, A, B, and C, both A and B could depend upon C. See the following figure. With strict fidelity, A and B must be compiled against the same C to be able to deploy them together. In large systems with hundreds of bundles, requiring all components to be compiled against the identical dependency providers quickly becomes infeasible.



Backward compatibility is the lubrication that reduces the friction to make it possible to run large systems based on disparate bundles that have shared dependencies. Backward compatibility decouples the importer and exporter by allowing a *range* of exporters to satisfy the needs of an importer. This allows an exporter to evolve without requiring the importer to change, or even to be rebuild, as long as the exporter's version remains in range.

Java has well defined binary compatibility rules. The dynamic linking of Java code permits quite a lot of changes between compilation time and runtime. Classes can get new fields and methods, the hierarchy can change, order of fields and methods can change, and more. However, interfaces in Java have very different rules for binary compatibility between *users* of an interface and *implementers* of an interface. From the perspective of a user, an interface can be changed significantly. In contrast, almost any change will not be backward compatible for an implementer of that interface. For example, adding a new method to an interface is invisible to an importer that uses that interface; such a change is therefore binary compatible for this user. However, an importer that implements that interface will be broken by such an addition. The distinction between the different roles to decide backward compatibility is a crucial one that has implications for the importers. Users and implementers must specify different version ranges for the exporters with which they can be compatible. An importer that implements interfaces from a package will require a much narrower range than an importer that only uses such interfaces.

For example, a bundle A exports an API package containing the following interface:

```
Bundle A:
    package com.acme.foo;
    public interface Foo {
          void bar();
    }
```

An implementation bundle B implements this interface:

```
Bundle B:
    package com.acme.impl.foo;
    import com.acme.foo.*;
    public class FooImpl implements Foo {
          public void bar() {}
    }
```

And the client bundle C uses the interface:

```
Bundle C:
    package com.acme.user.foo;
    import com.acme.foo.*;
    public class Client {
          public void foo(Foo foo) {
                foo.bar();
          }
    }
```

In the next release, the Foo interface exported by bundle A is updated to:

```
Bundle A:
    package com.acme.foo;
    public interface Foo {
          void bar();
          void baz();
    }
```

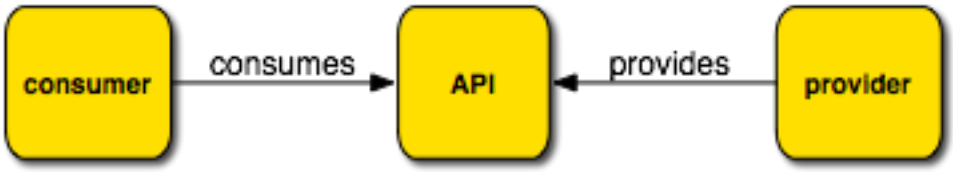
This change is *not* backward compatible for bundle B but is backward compatible for bundle C.

There are different types of importers that have different rules around backward compatibility. What is backward compatible for one importer is backward incompatible for another. This difference is introduced by the separation of API and implementation. An API package has two users: the *consumers* that use the API and the *providers* that implement the API.

Though the Java interface crisply demonstrates the issue with different backward compatibility rules for users and implementers, the model turns out to be simplistic in practice. In OSGi, the dominant artifact to be shared is a *package*. A package should consist of a cohesive set of classes, interfaces, and resources. Broadly

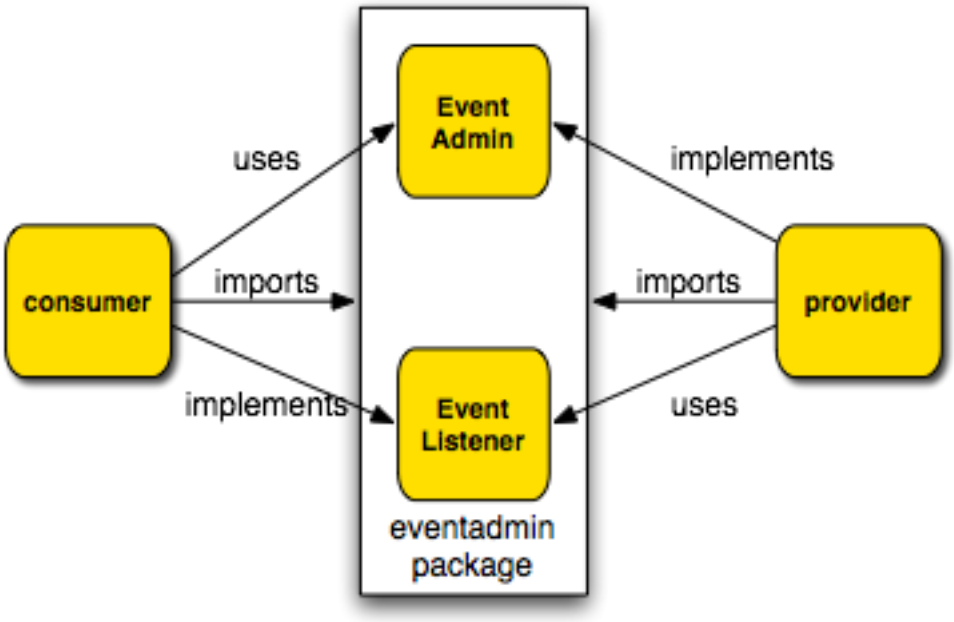
speaking, there are two types of packages: libraries and API. A library unifies the API and the provider of this API. For example, a library like ASM[1] does not attempt to separate the concrete implementation classes from their API, the API is the implementation.

API packages specify an abstract API to be implemented by an unmentioned provider. API packages are the core of the OSGi service model, whereby the provider of an API is represented by a service object. For example, the `org.osgi.service.eventadmin` package contains the API for an Event Admin provider. A consumer of Event Admin imports this package and a provider of Event Admin imports this package as well. Similar to interfaces, API packages have two types of importers; to distinguish between these roles they are called *consumers* and *providers*.



Consumers are not always only users of the interfaces in the API packages, nor are providers always only implementers of such interfaces. A consumer of an API package can actually be required to implement an interface that is then used by the provider of the API package. For example, in the Event Admin specification the Event Listener interface is implemented by the consumer of the Event Admin API and then used by the provider of the Event Admin API.

[1] http://asm.ow2.org/



To prevent confusion, this document always makes it explicit if an interface or package is discussed. A *consumer* consumes an API package, a *provider* provides an implementation of an API package. An *implementer* implements an interface and a user uses an interface. Last but not least, an *importer* imports a package and an *exporter* exports a package. Though this constellation might be confusing, all combinations of these concepts in real systems are quite common, these are all orthogonal concepts.

It should be obvious that binary compatibility plays an important role in backward compatibility. However, backward compatibility is also very dependent on the semantics. If the responsibility of an interface changes it could still be binary compatible but no longer be backward compatible.

Semantic Versions

OSGi versions are called semantic because they have *meaning*. Most versioning models are based on a gradually increasing version number where the only thing defined in the syntax was the ordering of version strings. Several package systems on Unix go a small step further by allowing comparisons to be done numerically or lexicographically. As a consequence, these version systems leave backward *incompatibility* undefined. Any version that is compared higher is assumed to be compatible, thereby making it impossible to ever introduce a breaking change. In

reality, not everything can be kept backward compatible. With existing version systems, the only escape is to rename the artifact.

In the real world, not everything is backward compatible, indicating that a later version is actually not a suitable choice. The OSGi Framework has the unique capability that it can actually host different versions of the same package and correctly handle the dependencies. Without semantic versions, the importer and exporter of a package have no way of communicating backward compatibility and incompatibility.

The core mechanisms provided by the OSGi specifications are the *version* and *version range*[2]. An exported package that can be used by other bundles has a *version*. Such an artifact is called an *exporter* in this document. A bundle that depends on a package uses a *version range* to limit the possible candidates. Such a bundle is called an *importer*. For example:

```
Export-Package: com.acme.foo; version=1.2.3.201003030903
Import-Package: com.acme.foo; version="[1.2,2.0)"
```

A version can consist of maximum 4 parts: major, minor, micro, and qualifier. The syntax is:

```
version ::= <major> [ '.' <minor> [ '.' <micro> [ '.' <qualifier>
]]]
```

Later parts can be ignored, implying 0 for minor and micro and empty string for qualifier. A version range has a syntax based on the interval notation from mathematics:

```
range   ::= ( '[' | '(' ) version ',' version ( ')' | ']' )
```

Square brackets (`'['` and `']'`) indicate inclusive and parentheses (`'('` and `')'`) indicate exclusive. That is, `[1.2,2.0)` indicates the version range from version `1.2`, including version `1.2`, up to but not including version `2.0`.

For importer version ranges to work, it is necessary that an exporter changes its version in a predictable way. This predictability allows an importer to use a range that matches its expectations of the exporter's evolution. For example, an importer could then limit the range it accepts, allowing the exporter to change its version in

[2] See sections 3.2.5 and 3.2.6 of the OSGi Core Specification v4.2.

such a way that it is no longer acceptable to previous importers, thereby signaling a breaking change.

An attractive solution to this problem would have been to allow the exporter to have multiple export versions (or an export version range). The exporter is in a much better position to judge backward compatibility than the importer; no prior agreement on how to version an artifact would be necessary with such a solution. If a package evolves, the exporter just lists the versions with which it is backward compatible. For example, when going from version 1 to version 2, the exporter would just list both versions in its export clause if the change was backward compatible.

Unfortunately, this simple and attractive model fails because the goals of all importers are not equal. Bundles that consume an API package have different backward compatibility rules than a provider of that API. Any semantic change in the API package must be handled by a provider to honor the change in the API contract while many of those changes are backward compatible for consumers. It is therefore paramount that API consumers and providers can describe their different import requirements on the exporter in a concise and easy to understand way. In OSGi, this is achieved by placing semantics on the parts of the version. In this way, a version acts as a small Domain Specific Language (DSL) that is used to document the evolution of the exporter. In this model, the exporter encodes its evolution in its version numbers and importers can then predict which version numbers are compatible with their needs and declare an appropriate import range.

The semantics for the version parts are therefore defined as:

1. *major* — Packages with versions that have different major parts are not compatible both for providers as well as consumers. For example, `1.2` and `2.3` are completely incompatible.
2. *minor* — API consumers are compatible with exporters that have the same major number and an equal or higher minor version. API providers are compatible with exporters that have the same major and minor version number. For example, `1.2` is backward compatible with `1.1` for consumers but for providers it is incompatible. Consumers should therefore import `[1.2,2)` and providers should import `[1.2,1.3)`.
3. *micro* — A difference in the micro part does not signal any backward compatibility issues. The micro number is used to fix bugs that do not affect either consumers or providers of the API.

4. *qualifier* — The qualifier is usually used to indicate a build identity, for example a time stamp. Different qualifiers do not signal any backward compatibility issues.

Assuming an exported version 1.2.3.built, the following ranges provide the given semantics:

`[1.2,2)`	Consumer importer policy: will not match when exporter goes to version 2 or later.
`[1.2,1.3)`	Provider importer policy: will not match when exporter goes to version 1.3 or later.
`[1.2.3,1.2.4)`	Strict importer policy: only accepts exporter of version 1.2.3.

Exporter Policy

Exporters must carefully version any exported package. It is the experience of the OSGi Alliance that virtually any change in an API package is not backward compatible for providers implementing that API. This usually means that any change causes an increment of the minor version part. In the history of the OSGi Alliance, there have been no changes in the major versions of its specifications. The `org.osgi.framework` package is currently at version 1.5.1, indicating it underwent 5 modifications that required changes in the framework providers (for example, Knopflerfish, Felix, Equinox) but so far no modifications that required changes in consumers of the framework API.

The Java Language Specification defines binary compatibility in chapter 13[3]. Any semantic changes must be judged by humans. Tools will be able to detect syntactic violations of the semantic versioning by comparing a previous version and the new version. However, such tools will not be able to detect semantic changes in the code.

Exporters of implementation code should treat the versioning of this code as API packages merged with its provider. There is therefore no concern for providers, only consumers have to be considered. In practice, this means that such a package will only undergo major changes.

An interesting problem for exporters is the policy applied to interfaces that are implemented by the consumers of the API. For example, in the OSGi framework, the Bundle Activator interface is not implemented by a framework provider but by consumers of the framework API: bundles. A change in such an interface will not

[3] http://java.sun.com/docs/books/jls/second_edition/html/binaryComp.doc.html

be backward compatible with any consumers of the API and therefore requires a change in the major part of the version. API designers must therefore be acutely aware of the usage pattern of their interfaces and try to prevent making changes to interfaces that are implemented by the consumers of the API. Any change in an interface that is implemented by a consumer of the API breaks backward compatibility for all consumers. This policy has been strictly followed in the OSGi specifications. Virtually all consumer implemented interfaces are kept very simple and, to date, have not been changed.

Importing the Packages You Export

OSGi has always strongly promoted importing the packages you export. The resulting substitutability can be used by a framework to minimize the number of different class spaces. There are, however, a number of caveats around this model. Importing exported packages *only* works well when the exported packages are not bound to private packages. Once an exported package uses a private package, it is not pure API: the package is coupled to implementation packages. Such exported packages cannot be safely substituted by an import from another bundle. Good API design, such as the service API in OSGi, ensures that API packages are not coupled to implementation packages. However, many libraries have packages that are not that cleanly separated. Bundles containing these libraries should not import their exported packages.

When exported packages are imported, care must be taken to specify the import version in the proper way. If an exported package includes the implementation of the API in the package, then the import of the package must use a version range that represents the compatibility requirements for providers of an API implementation.

Importer Policy

Having semantic versions offers the possibility to standardize the usage of versions. Import versions can be derived from their build time dependency. For example, if bundle B provides the implementation of an API then it should depend upon a version range that limits changes to the micro and qualifier only. Such a rule is a called a *version policy*.

For stability reasons, an importer should be bound to the lowest possible version it can correctly compile against. It is a common misconception that code should be updated when newer versions of libraries it depends on become available. Not only would this increase the chores of maintaining versions, it easily causes a continuous update cycle in deployment. Any change should change the version of the corresponding package, which requires an update of all the dependencies, which could trigger more changes. Systems designed that way become very brittle very quickly.

An organization must decide upon a version policy regarding whether to depend upon bug fixes or not. Using import version ranges that include the micro part of the version against which the code is being compiled has the advantage that the deployment is forced to upgrade to the bug fix. However, this easily increases the volatility of the deployment. Not including the bug fix in the version range means that the code will not disrupt the deployment but can run against code that was not properly tested and may have a bug. This tradeoff decision must be made by the importer. In the OSGi Alliance build, the micro part is removed from import version ranges, assuming that the deployer is in charge of maintaining a system with the latest updated artifacts but not forcing this upgrade when a third artifact just happened to have been compiled against the fixed component.

Providers implementing an API package must import with a version range that specifies the floor of the version used for compilation and the ceiling of the next minor part. For example, when compiled against version `2.1.4`, the import version range for a provider is `[2.1,2.2)` (assuming bug fixes are ignored).

Consumers of an API package should import with the same floor but can increase the ceiling to the next major version. For example, when compiled against version `2.1.4`, the import version range for a consumer is `[2.1,3)` (again, assuming bug fixes are ignored).

There exist tools that can calculate the import ranges based on the export version and a consumer and provider importer policy.

Bundles and Fragments

The previous chapters use package imports and exports as examples. The reason is that the asymmetry between API consumers and providers is very clear with packages, especially with the OSGi service model where exported packages are always API. However, some analogies can be made with bundles and fragments.

Requiring another bundle is similar to a short form of importing all the exported packages of that required bundle. The version of a bundle must therefore semantically aggregate the semantics of all its constituent packages. If any of these packages is incompatible with its providers then the bundle version must increment the minor version. If any of these packages is incompatible with consumers, the bundle version must increment the major version. It is clear, that on average, the version of a bundle will be much more volatile than the versions of its constituent packages, increasing the dependency problems.

Conclusion

Versioning is one of the major chores of software. With systems consisting of hundreds to thousands of bundles, tooling becomes a necessity to manage versions. Humans are already incapable of handling version management on the scale that is required today. Tools need rules and guidelines; rules as laid down in this document. To allow applications to grow even larger it is paramount that versions have semantics and are therefore predictive.

The OSGi Alliance

The OSGi Alliance is a worldwide consortium of technology innovators that advances a proven and mature process to enable the componentization of applications into well-defined software modules, and ensure interoperability of applications and services over a broad variety of devices.

The Alliance provides specifications, reference implementations, test suites and certification to foster a valuable cross-industry ecosystem. OSGi technology is shipping in millions of units worldwide, and is deployed by Fortune Global 500 companies in enterprise, desktop, embedded home and telematics markets. Member companies

collaborate within an egalitarian, equitable and transparent environment and promote adoption of OSGi technology through business benefits, user experiences and forums.

For more information on the non-profit technology corporation, visit http://www.osgi.org or contact help@osgi.org.

OSGi is a trademark of the OSGi Alliance in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

All other marks are trademarks of their respective companies.