2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

Semantic Versioning versus Breaking Changes: A Study of the Maven Repository

Steven Raemaekers
Software Improvement Group
Amsterdam, The Netherlands
Email: s.raemaekers@sig.eu

Arie van Deursen
Technical University Delft
Delft, The Netherlands
Email: arie.vandeursen@tudelft.nl

Joost Visser
Software Improvement Group
Amsterdam, The Netherlands
Email: j.visser@sig.eu

***Abstract*—For users of software libraries or public programming interfaces (APIs), backward compatibility is a desirable trait. Without compatibility, library users will face increased risk and cost when upgrading their dependencies. In this study, we investigate semantic versioning, a versioning scheme which provides strict rules on major versus m inor and patch releases. We analyze seven years of library release history in Maven Central, and contrast version identifiers with actual incompatibilities. We find that around one third of all releases introduce at least one breaking change, and that this figure is the same for minor and major releases, indicating that version numbers do not provide developers with information in stability of interfaces. Additionally, we find that the adherence to semantic versioning principles has only marginally increased over time. We also investigate the use of deprecation tags and find out that methods get deleted without applying deprecated tags, and methods with deprecated tags are never deleted. We conclude the paper by arguing that the adherence to semantic versioning principles should increase because it provides users of an interface with a way to determine the amount of rework that is expected when upgrading to a new version.**

***Keywords*-Semantic versioning, Software libraries**

I. Introduction

For users of software libraries or public programming interfaces (APIs), backward compatibility is a desirable trait. Without compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security fixes. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

Determining whether there are incompatibilities, however, is hard to do for the library user (it is, in fact, undecidable in general). Therefore, it is the library creator's responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, *semantic versioning*[1] (`semver`) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

- MAJOR: This number should be incremented when incompatible API changes are made;
- MINOR: This number should be incremented when functionality is added in a backward-compatible manner;
- PATCH: This number should be incremented when backward-compatible bug fixes are made.

These principles were formulated in 2010 by (GitHub founder) Tom Preston-Werner.[2] As argued in the semantic versioning specification, *"these rules are based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software."*

But how common are these practices in reality? Are such changes just harmless, or do they actually hurt by causing rework? Do breaking changes mostly occur in major releases, or do they occur in minor releases as well? Do major and minor releases differ in terms of typical size? Furthermore, for the breaking changes that do occur, to what extent are they signalled through, e.g., *deprecation tags*? Finally, does the presence of breaking changes affect the time (delay) between library version release and actual adoption of the new release in clients?

In this paper, we seek to answer questions like these. To do so, we make use of seven years of versioning history as present in the collection of Java libraries available through Maven's central repository.[3] Our dataset comprises around 150,000 binary jar files, corresponding to around 22,000 different libraries for which we have 7 versions on average. Furthermore, our dataset includes cross-usage of libraries (libraries use other libraries in the dataset), permitting us to study the impact of incompatibilities in concrete clients as well.

As an approximation of the (undecidable) notion of backward compatbility, we use *binary compatibility* as defined in the Java language specification. This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities

[1] http://semver.org

[2] Github actively promotes `semver` and encourages all 10,000,000 projects hosted by GitHub to adopt it.

[3] http://search.maven.org/

978-1-4799-6148-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SCAM.2014.30

as well. As a measurement for the amount of changed functionality in a release, we will use the *edit script size* between two subsequent releases. Equipped with this, we will study versioning practices in the Maven dataset, and contrast them with the idealized guidelines as expressed in the `semver` specification.

This paper is structured as follows. We start out, in Section II, by sketching related work in the area of version analysis. In Section III, we formulate the research questions we seek to answer. Then, in Section IV, we describe our approach to answer these questions, and methods of measurement. In Section V we provide descriptive statistics of the Maven dataset. In Sections VI–IX we present our analysis in full detail. We discuss the wider implications and the threats to the validity of our findings in Sections X and XI. We conclude the paper in Section XII.

II. Related Work

To the best of our knowledge, our work is the first systematic study of versioning principles in a large collection of Java libraries. However, several case studies on backward compatible and incompatible changes in public interfaces as appearing in these libraries have been performed [2], [7], [9]. For instance, Cossette et al. [2] investigate binary incompatibilities introduced in five different libraries and aim to detect the correct adaptations to upgrade to the newer version of the library. Similarly, Dig et al. [9] investigate binary incompatibilities in five other libraries and conclude that most of the backward incompatible API changes are behavior-preserving refactorings. Dietrich et al. [7] have performed an empirical study into evolution problems caused by library upgrades. They manually detect different kinds of *source* and *binary* incompatibilities, and conclude that although incompatibility issues do occur in practice, the selected set of issues does not appear very often.

Another area of active research is to automatically detect refactorings based on changes in public interfaces [3], [4], [8]. The idea behind these approaches is that these refactorings can automatically be "replayed" to update to a newer version of a library. This way, an adaptation layer between the old and the new version of the library can automatically be created, thus shielding the system using that library from backward incompatible changes.

While our work investigates backward incompatibilities for given version string changes, Bauml et al. [1] take the opposite approach, in the sense that they propose a method to generate version number changes based on changes in OSGi bundles. A comparable approach in the Maven repository would be to create a plugin that automatically determines the correct subsequent version number based on backward incompatibilities and the amount of new functionality present in the new release as compared to the previous one.

The Maven repository has been used in other work as well. Davies et al. [5] use the same dataset to investigate the provenance of a software library, for instance, if the source code was copied from another library. They deploy several different techniques to uniquely identify a library, and find out its history, much like a crime scene containing a fingerprint. Ossher et al. [11] also use the Maven repository to reconstruct a repository structure with directories and version based on a collection of libraries of which the groupId, artifactId and version are not known.

III. Research Questions

The overall goal of this paper is to understand to what degree versioning conventions are adhered to in the development of software libraries. This leads to a better understanding of how developers use versioning schemes to identify expected amounts of rework for users of the interfaces they offer.

We regard `semver` as a formalization of principles that developers already implicitly embraced, even before the manifesto was released in 2010. We use the explicit rules of `semver` as a way to formally test these principles. We want to find out if developers actually mean to give a signal, for instance, that a library contains only backward-compatible bug fixes when releasing a new patch version, or that a library introduces a substantial number of backward-incompatible changes to its public interface when releasing a new major version.

To achieve our overal goal, we seek to answer the following research questions in this paper:

- **RQ1**: How are semantic versioning principles applied in practice in the Maven repository in terms of binary (in)compatible changes?
- **RQ2**: Has the adherence to semantic versioning principles increased over time?
- **RQ3**: How are dependencies to newer versions updated, and what are factors causing systems *not* to include the latest versions of dependencies?
- **RQ4**: How are deprecation tags applied to methods in the Maven repository?

In the next section, we discuss our research method.

IV. Method

In this paper, we analyze a snapshot of the Maven's Central Repository, dated July 11, 2011.[4] Maven is an automated build system that manages the entire "build cycle" of software projects. To use Maven in a software project, a `pom.xml` file is created that specifies the project structure, settings for different build steps (e.g. compile, package, test) as well as libraries that the project depends on. These libraries are automatically downloaded by maven, from specified repositories. These repositories can be private as well as public. For open source systems, the *Central*

[4]Obtained from http://juliusdavies.ca/2013/j.emse/bertillonage/maven.tar.gz based on [5], [6]

Repository is typically used, which contains jar files and sources for the most widely used open source Java libraries.

Our dataset extracted from this central repository contains 148,253 Java binary jar files and 101,413 Java source jar files for a total of 22,205 different libraries. This gives an average of 6.7 releases per library. For more information on our dataset, which includes resolved and versioned dependencies at the method level, we refer to [12].

A. Determining backward incompatible API changes

Determining full backward compatibility amounts to determining equivalence of functions, which in general is undecidable. Instead of such semantic compatiblity, we will rely on binary incompatibilities.

Binary incompatible changes, in this paper also called *breaking changes*, are formally defined by the Java Language specification as follows: "*a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.*"[5] In this paper, we will use the following working definition: breaking changes are any changes to a library interface that require recompilation of systems using the changed functionality. Examples of breaking changes are method removals and return type changes[6].

To detect breaking changes between each subsequent pair of library versions, we use Clirr[7]. Clirr is a tool that takes two jar files as input and returns a list of changes in the public API. Clirr is capable of detecting 43 API changes in total, of which 23 are considered breaking and 20 are considered non-breaking. Clirr does not detect all binary incompatibilities that exist, but it does detect the most common ones (see Table 2). We executed Clirr on the complete set of all subsequent versions of releases in the Maven repository. The approach to determine subsequent versions is described next.

Whenever Clirr finds a binary incompatibility between two releases, those releases are certainly not compatible. However, if Clirr fails to find a binary incompatibility, the releases can still be semantically incompatible. As such, our reports on e.g., the percentage of releases introducing breaking changes is an underestimation: The actual situation may be worse, but not better.

B. Determining subsequent versions and update types

In the Maven repository, each library version (a single jar file) is uniquely identified by its `groupId`, `artifactId`, and `version`, for instance "`org.springframework`", "`spring-core`" and "`2.5.6`". To determine subsequent version pairs, we sort all versions with the same `groupId` and `artifactId` based on their version string. We used the Maven Artifact API[8] to compare version strings with each other, taking into account the proper sorting given the major, minor, patch and prerelease in a given version string. For each subsequent pair of releases from this sorted list, the release type is determined according to the change in version number. For instance, a change in version number from "1.0" to "1.1" was marked as a minor release. We do not check whether version numbers are incremented properly, i.e. if there are no gaps in version numbers.

Since `semver` applies only to version numbers containing a major, minor and patch version number, we only investigate pairs of library versions which are both structured according to the format "MAJOR.MINOR.PATCH" or "MAJOR.MINOR". In the latter case, we assume an implicit patch version number of 0.

Semantic versioning also permits prereleases, such as `1.2.3-beta1` or (as commonly used in a `maven` setting) `1.2.3-SNAPSHOT`. We exclude prereleases from our analysis since `semver` does not provide any rules regarding breaking changes or new functionality in these release types.

C. Detecting changed functionality

In order to compare major, minor, and patch releases in terms of size, we look at the amount of changed functionality between releases. To do so, we look at the edit script between each pair of subsequent versions, and measure the size of these scripts. We do so by calculating differences between abstract syntax trees (ASTs) of the two versions. Hence, we can see, for example, the total number of statements that needs to be inserted, deleted, updated or moved to convert the first version of the library into the second. We use the static code analysis tool ChangeDistiller [9] to calculate edit scripts between library versions. For more information on ChangeDistiller, we refer to [10].

D. Obtaining release intervals and dependencies

To calculate release intervals, we collect upload dates for each jar file in the Maven Central Repository. Unfortunately, not for all libraries a valid upload date is available. Ultimately, for 129,183 out of 144,934 (89.1%) libraries we could identify a valid release date.

E. Obtaining deprecation patterns

For API developers, the Java language offers the possibility to warn about future incompatibilities by means of the "`@Deprecated`" annotation.[10] Old methods can be marked as deprecated, but as they are not removed backward compatibility is retained. Also in `semver`, the use

[5] http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html

[6] For an overview of different types of binary incompatibilities and a detailed explanation, see http://wiki.eclipse.org/Evolving_Java-based_APIs

[7] http://clirr.sourceforge.net

[8] http://maven.apache.org/ref/3.1.1/maven-artifact

[9] https://bitbucket.org/sealuzh/tools-changedistiller

[10] http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html

#	*Pattern*	*Example*	*#Single*	*#Pairs*	*Incl.*
1	MAJOR.MINOR	2.0	20,680	11,559	yes
2	MAJOR.MINOR.PATCH	2.0.1	65,515	50,020	yes
3	#1 or #2 with nonnum. chars	2.0.D1	3,269	2,150	yes
4	MAJOR.MINOR-prerelease	2.0-beta1	16,115	10,756	no
5	MAJOR.MINOR.PATCH-pre.	2.0.1-beta1	12,674	8,939	no
6	Other versioning scheme	2.0.1.5.4	10,930	8,307	no
		Total	129,138	91,731	

Table 1. Version string patterns and frequencies of occurrence in the Maven repository.

of such annotations is required, before methods are actually removed.

We detect deprecated methods in the following way. We extract the source code from source jar files for each library and, for performance reasons, textually search for occurrences of the string "`@Deprecated`" first. Only when at least one deprecated tag is found, we parse the complete source code of the library using the JDT (Java Development Tools) Core library[11].

Using JDT, we create an abstract syntax tree for each source file, and apply a visitor to find out which methods have deprecation tags. Next versions of the same method are connected using method header (name and parameters) matching. Combining this information with the update types from Section IV-B makes it possible to distinguish between different types of deprecation patterns.

V. Descriptive Statistics

Before answering our research questions, we provide an overview of the actual use of version strings that comply with `semver`, and of the most common types of breaking changes in the Maven dataset.

A. Version string patterns

Table 1 shows the six most common version string patterns that occur in the Maven repository. For each pattern, the table shows the number of libraries with version strings that match that pattern (*#Single*) and the number of subsequent versions that both follow the same pattern (*#Pairs*) – we will use the latter to identify breaking changes between subsequent releases.

The first three versioning schemes correspond to actual `semver` releases, whereas the remaining ones correspond to *prereleases*. Since prerleases can be more tolerant in terms of breaking changes (`semver` does not state what the relationship between prereleases and non-prereleases in terms of breaking changes and new functionality is)[12] we exclude prereleases from our analysis.

The table shows that the majority of the version strings (69.3%) is formatted according to the first two schemes, and 22.3% of the version strings contains a prerelease label (patterns 4 and 5). The difference between the single and the pair frequency is due to two reasons: (1) the second version string of an update can follow a different pattern than the first; and (2) a large number of libraries only has a single release (6,442 out of 22,205 libraries, 29%).

[11] http://www.eclipse.org/jdt/core

[12] Pre-releases in maven correspond to -SNAPSHOT releases, which should not be distributed via Maven's Central Repository (see https://docs.sonatype.org/display/Repository/Sonatype+OSS+Maven+Repository+Usage+Guide)

Breaking changes		
#	*Change type*	*Frequency*
1	Method has been removed	177,480
2	Class has been removed	168,743
3	Field has been removed	126,334
4	Parameter type change	69,335
5	Method return type change	54,742
6	Interface has been removed	46,852
7	Number of arguments changed	42,286
8	Method added to interface	28,833
9	Field type change	27,306
10	Field removed, previously constant	12,979

Non-breaking changes		
#	*Change type*	*Frequency*
1	Method has been added	518,690
2	Class has been added	216,117
3	Field has been added	206,851
4	Interface has been added	32,569
5	Method removed, inherited still exists	25,170
6	Field accessibility increased	24,954
7	Value of compile-time constant changed	16,768
8	Method accessibility increased	14,630
9	Addition to list of superclasses	13,497
10	Method no longer final	9,202

Table 2. The most common breaking and non-beaking changes in the Maven repository as detected by Clirr.

B. Breaking and non-breaking changes

Table 2 shows the top 10 breaking and non-breaking changes in the Maven repository as detected by Clirr. The most frequently occurring breaking change is the method removal, with 177,480 occurrences. A method removal is considered to be a breaking change because the removal of a method leads to compilation errors in all places where this method is used. The most frequently occurring non-breaking change as detected by Clirr is the method addition, with 518,690 occurrences.

Table 3 shows the number of major, minor and patch releases containing at least one breaking change. The table shows that 35.8% of major releases contains at least one breaking change, which in accordance with guidelines such as `semver`. We also see that 35.7% of minor releases and 23.8% of patch releases contain at least one breaking change. This is in sharp contrast to the requirement that minor and patch releases should be backward compatible. The overall number of releases that contain at least one breaking change is 30.0%.

The table shows that there does not exist a large difference between the percentage of major and minor releases that contain breaking changes. This indicates that `semver` is not adhered to in practice with respect to breaking changes. If this were the case, the number of minor and patch releases containing breaking changes would be 0 in the table. The

	Contains at least 1 breaking change				
Update type	*Yes*	*%*	*No*	*%*	*Total*
Major	4,268	35.8%	7,624	64.2%	11,892
Minor	10,690	35.7%	19,267	64.3%	29,957
Patch	9,239	23.8%	29,501	76.2%	38,740
Total	24,197	30.0%	56,392	70.0%	80,589

Table 3. The number of major, minor and patch releases that contain breaking changes.

	#Breaking		*#Non-break.*		*Edit script*		*Days*	
Type	μ	σ^2	μ	σ^2	μ	σ^2	μ	σ^2
Major	58.3	337.3	90.7	582.1	50.0	173.0	59.8	169.8
Minor	27.4	284.7	52.2	255.5	52.7	190.5	76.5	138.3
Patch	30.1	204.6	42.8	217.8	22.7	106.5	62.8	94.4
Total	32.0	264.3	52.2	293.3	37.2	152.3	67.4	122.9

Table 4. Analysis of the number of breaking and non-breaking changes, edit script size, and release intervals of major, minor, and patch releases.

total number of updates in Table 3 (80,589) differs from the total number of pairs in Table 1 because of missing or corrupt jar files, which have a version string but cannot be analyzed by Clirr.

VI. RQ1: Major vs Minor vs Patch Releases

To understand the adherence of semantic versioning principles for major, minor, and patch releases, Table 4 shows the average number of breaking changes, non-breaking changes, edit script size and number of days for the different release types. Each release is compared to its immediate previous release, regardless of the release type of this previous release.

As the table shows, on average there are 58 breaking changes in a major release. Minor and patch releases introduce fewer breaking changes (around half as many as the major releases), but 27 and 30 on average is still a substantial number (and clearly not 0 as semantic versioning requires). The differences between the three update types are significant with $F = 7.31$ and $p = 0$, tested with a nonparametric Kruskall-Wallis test, since the data is not normally distributed[13].

In terms of size, major releases are somewhat smaller than minor releases (average edit script size of 50 and 52, respectively), with patch releases substantially smaller (22), with $F = 117.49$ and $p = 0$. This provides support for the rule in `semver` stating that patch releases should contain only bug fixes, which overall would lead to smaller edit script sizes than new functionality.

With respect to release intervals, these are on average 2 (for major and patch releases) to 2.5 months (for minor releases), with $F = 115.47$ and $p = 0$. It is interesting to see that minor, and not major updates take the longest time to release.

Care must be taken when interpreting the mean for skewed data. All data in this table follows a strong power law, in which the most observations are closer to 0 and there are a relative small amount of large outliers. Nonetheless, a larger mean indicates that there are more large outliers present in the data.

Thus, to answer **RQ1**: *The strict principles of semantic versioning regarding breaking changes are not adhered to in practice. Instead of being free of breaking changes, minor and patch releases include 30 breaking changes on average.*

[13]Even if the data is not normally distributed, we still summarize the data with a mean and standard deviation to provide insight in the data.

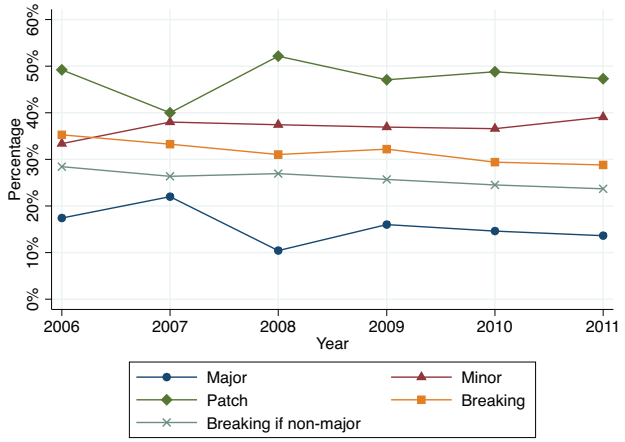

Figure 5. The percentage of major, minor, patch, breaking, and breaking if non-major releases through time.

In Section X we will get back to these results and try to provide explanations. We first continue with an analysis of adherence to `semver` through time.

VII. RQ2: Semantic Versioning Adherence Over Time

To find out if the adherence to `semver` has changed over time, we plot the number of major, minor and patch releases through time and the number of releases containing breaking changes over time. This plot is shown in Figure 5.

The figure shows that the ratio of major, minor and patch releases is relatively stable and around 15%, 30% and 50%, respectively. The percentage of major releases per year seems to decrease slightly in later years.

Regardless of release type, one in every three releases contains breaking changes. This percentage is relatively stable but slightly decreasing in later years. One out of every four releases violates `semver` ("breaking if non-major"), but this percentage also slightly decreases in later years: from 28.4% in 2006 to 23.7% in 2011.

To answer **RQ2**: *The adherence to semantic versioning principles has increased over time with a moderate decrease of breaking changes in non-major releases from 28.4% in 2006 to 23.7% in 2011.*

VIII. RQ3: Update Behavior

The key reason to investigate breaking changes is that they complicate upgrading a library to its latest version. To

	Update L			
Update S	*Major*	*Minor*	*Patch*	*Total*
Major	543	189	82	814
Minor	651	791	227	1,669
Patch	150	54	297	501
Total	1,344	1,034	606	2,984

Table 6. The number of updates of different types of S and simultaneous updates of dependency L.

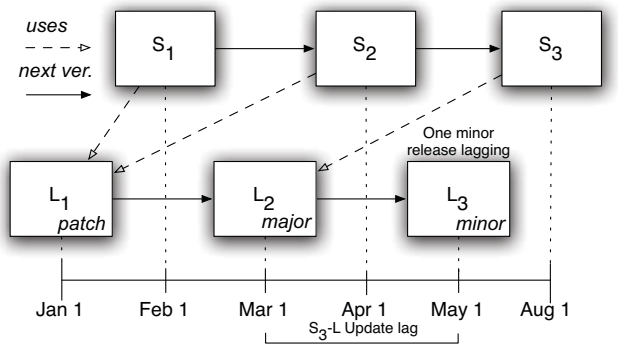

Figure 7. An example of a timeline with a system S updating library L.

what extent is this visible in the maven dataset? What delay is there typically between a library release and the usage of that release by other systems? Is this delay affected by breaking changes?

To investigate the actual update behavior of systems using libraries, we collected all updates from the Maven repository that update one of their dependencies. Thus, we investigate usage scenarios within the maven dataset.

We obtained a list of 2,984 updates from the Maven repository of the form $\langle S_x, S_{x+1}, L_y, L_{y+1} \rangle$, where L is a dependency of S which was updated from version y to version $y+1$ in the update of S from x to $x+1$. For example, when the Spring framework included version 3.8.1 of JUnit in version 2.0, but included version 3.8.2 in version 2.1, Spring framework performed a minor update of JUnit in a patch release.

Table 6 shows the number of updates of different types of S and L in the Maven repository. The table shows that most major updates of dependencies (543) are performed in major updates of S, and most minor updates of dependencies (791) are performed in minor updates of S. The same is true for patch updates of dependencies, which are most frequently updated in patch updates of S (297).

To further investigate update behavior of dependencies, we calculate the number of versions of L that S lags behind, as illustrated in Figure 7. The figure shows an example of three versions of S, and a dependency L of S. On January 1, L_1, a patch update, is released. S_1 decides to use this version in its system. On March 1, a major update of L is released, L_2. The next release of S, S_2, happens on April 1. This release still includes L_1, although L_2 was already available to include in S_2. The same is true for S_3, which could have included L_3 but still includes L_2. The period that S has been using L_1 is from February 1, to April 1. The total time that S has a dependency on L is from February 1 to August 1.

	min	*p25*	*p50*	*p75*	*p90*	*p95*	*p99*	*max*
Major	0	0	0	0	1	1	4	22
Minor	0	0	0	1	2	4	6	101
Patch	0	0	0	1	5	6	13	46

Table 8. Percentiles for the number of major, minor and patch dependency versions lagging.

This example illustrates that there can exist a lag between the release of a new version of L and the inclusion in S. In this example, S_3 lags one minor release behind, and could have included L_3. The time S_3 theoretically could update to L_3 is between May, 1 and August, 1.

For each system S and each of its dependencies L, we calculate the number of major, minor and patch releases that version of S lags behind. The release dates of S_x and L_y are used to determine the number of releases after L_y but before S_x.

Table 8 shows percentiles for the number of major, minor and patch versions that dependencies L of system S are lagging as compared to the latest releases of L at the release date of S. For instance, when a system released a new version at January 1, 2013 and that release included a library with version 4.0.1 but there have been 10 minor releases of that library before January 1 and after the release date of version 4.0.1 that could have been included in that release of S, the number of minor releases lagging is 10 for that system-library combination. These numbers are calculated for each system-library combination separately.

The table shows that the number of major releases that S lags on average tends to be smaller than the number of minor and patch releases lagging. The distributions are highly skewed, with a median of 0 for all three release types and a 75th percentile of 1 for minor and patch releases, indicating that the majority of library developers include the latest releases of dependencies in their own libraries. The numbers also indicate that developers tend to better keep up with the latest major releases than with minor and patch releases, as indicated by the 90th percentile of 1 for major releases and a 90th percentile of 5 for patch releases.

To better understand the reasons underlying the update lag, we investigate two properties of libraries that could influence the number of releases that systems are lagging: the edit script size and the number of breaking changes of these dependencies. We hypothesize that people are reluctant to update to a newer version of a dependency when it introduces a large number of breaking changes or introduces a large amount of new or changed functionality. To test this, we investigate whether a positive correlation exists between the number of major, minor and patch releases lagging in libraries using a dependency and the number of breaking changes and changed functionality in new releases of that

	Breaking changes	Edit script size
Major versions lagging	0.0772	-0.0701
Minor versions lagging	0.1440	0.1272
Patch versions lagging	0.0190	0.0199

Table 9. Spearman correlations between the size of the update lag of L and breaking changes and the edit script size in the next version of L.

dependency. We calculate Spearman correlations between the number of versions lagging and the number of breaking changes and edit script size in these versions.

The results are shown in Table 9. The table shows Spearman correlations, which are calculated on 13,945 observations and all have a p-value of 0. The correlations are generally very weak, with the maximum correlation being 0.1440 between the number of minor versions lagging and the number of breaking changes in these dependencies.

The results indicate that although the number of breaking changes and the edit script size of a library does seem to have some influence on the number of library releases systems are lagging, the influence generally is not very large.

To answer **RQ3**: *updates of dependencies to major releases are most often performed in major library updates. There exists a lag between the latest versions of dependencies and the versions actually included, with the gap being the largest for patch releases and the smallest for major releases. There exists a small influence of the number of backward incompatibilities and of the amount of change in new versions on this lag.*

IX. RQ4: Deprecation Patterns

As we have seen, breaking changes are common. To deal with breaking changes, the Java language offers deprecation annotations. For the use of such annotations, semantic versioning provides the following rules for deprecation of methods in public interfaces: *"a new minor release should be issued when a new deprecation tag is added. Before the functionality is removed completely in a new major release, there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new API."*[14] Thus, whenever there is a breaking change (which must be in a major release), this should be preceded by a deprecation (which can be in a minor release).

In this section, we investigate whether this principle is adhered to in practice. We investigate how many libraries actually deprecate methods, and if they do, how many releases it takes before these methods get deleted, if at all. We also find out if there is indeed at least one minor change in between before the method is removed, as `semver` prescribes.

In total, 1196 out of 22,205 artifacts (5.4%) contain at least one method deprecation tag. Given our observation that 1 in 3 releases introduces breaking changes, this number immediately appears to be too low.

[14]http://semver.org/spec/v2.0.0.html

Table 10 shows different possible deprecation patterns. The table uses a typical library with 4 releases (two major, two minor). For each pattern in the table, we count its frequency in the maven data set. As the table shows, there are a couple of different ways to deprecate and delete methods in major or minor releases, some of which are correct according to `semver` (column *c*).

Cases 1 and 2 in Table 10 show an example of a private method with and without deprecation tags. As the table shows, the first case occurs in 24.24% of all methods. Since `semver` is only about versioning and changes in *public* interfaces, these cases are therefore not investigated further. Case 3 shows a public method that is neither deleted nor deprecated, which is the most common life cycle for a method (42% of the cases). Case 4 shows a public method that is deprecated, but is never removed in later versions. According to the principles regarding deprecation as stated in `semver`, this is correct behavior. As the table shows, this is the most common use of the deprecation tag, even though it is used in just 793 methods. Case 5 shows a public method that is removed from the interface but never declared deprecated, which is *not* correct: This is the typical case of introducing a breaking change in a minor release. Case 6 deprecates the method, but deletes it in a *minor* release, which would not be correct. This case does not occur. Case 7 declares the method deprecated in a major release, which would also be incorrect (and which does not occur). Case 8 shows an example of deprecation by the book, exactly as prescribed by `semver`. The method is declared deprecated in a minor release, there is another minor release that also declares the method deprecated and in the next *major* release, the method is removed. This correct pattern does not occur at all in the maven data set. Case 9 shows a method that is undeprecated, about which `semver` does not explicitly contain a statement.

As the table further shows, public methods without a deprecated tag in their entire history are in the majority with 42.27%. Surprisingly, the number of public methods that ever get deprecated in their entire history is only 793, or 0.30%. The number of public methods that get deleted without a deprecated tag is 86,449, or 33.03%. The number of methods that get deleted after adding a deprecated tag to an earlier version is 0 (cases 6 and 8). Finally, the number of methods that get "undeprecated" is 0.01%.

These results are surprising since they suggest that developers do not apply deprecation patterns in the way that `semver` proposes. In fact, developers do not seem to use the deprecated tag for methods very often at all. Most public methods get deleted without applying a deprecated tag first (case 5), and methods that do get a deprecated tag are almost never deleted (case 4). This suggests that developers are reluctant to remove deprecated functionality from new releases, possibly because they are afraid to break backward compatibility. Case 8 is, according to `semver`, the only

#	*v1 (maj.)*	*v2 (min.)*	*v3 (min.)*	*v4 (maj.)*	*c*	*i*	*Freq.*	*%*
1	pr m1	pr m1	pr m1	pr m1	y	n	63,698	24.34
2	pr m2	pr m2	pr @d m2	pr @d m2	y	n	113	0.04
3	pu m3	pu m3	pu m3	pu m3	y	n	110,613	42.27
4	pu m4	pu @d m4	pu @d m4	pu @d m4	y	y	793	0.30
5	pu m5	pu m5	-	-	n	y	86,449	33.03
6	pu m6	pu @d m6	-	-	n	y	0	0
7	pu m7	pu m7	pu m7	pu @d m7	n	y	0	0
8	pu m8	pu @d m8	pu @d m8	-	y	y	0	0
9	pu m9	pu @d m9	pu m9	pu m9	n	y	16	0.01

Table 10. Possible method deprecation patterns. @d = deprecated tag, *c* = correct, *i* = interesting; pr = private; pu = public; – = method deleted.

proper way to deprecate and delete methods. However, the pattern was not found in the entire Maven repository.

To answer **RQ4**: *Developers do not follow deprecation guidelines as suggested by semantic versioning. Most public methods are deleted without applying a deprecated tag first, and when these tags are applied to methods, these methods are never deleted in later versions.*

X. Discussion

The results of this study indicate that the stability of interfaces and mechanisms to signal this instability to developers leaves much to be desired. One in every three interfaces contains breaking changes, and additionally, one in three interfaces that should not contain breaking changes actually does. The usage of the deprecation tag and the deletion of methods in the Maven repository show that the average developer tends to disregard the effects his actions have on clients of a library.

A. Low adherence explained

Even though the used versioning schemes on itself of a large number of libraries conforms to the versioning scheme as endorsed by `semver`, developers apparently do not conform to the actual rules as set out by this standard. If developers would adhere completely to these principles and their releases contain the same amount of breaking changes as found in the Maven repository, the number of major releases should be much larger than is currently the case. This low adherence is surprising since there are no other mechanisms available, except versioning schemes and deprecation tags, which signal interface instability. We argue that the principles set out by semver should be followed by every developer of software libraries, or any piece of software that is used by external developers.

We argue that ultimately, better designed and more stable interfaces leads to a lower maintenance burden of software in general. When a library user, or a user of any piece of publicly available functionality knows that there are expected changes when upgrading to a newer version, the developer can anticipate this and choose to postpone or include the update. Strict adherence to semantic versioning principles also forces library developers to think hard about the functionality they release, and about the design of the public interface they are releasing. It is increasingly hard for library developers to change their overall design of their interface after it has been published. This problem becomes worse the more users actually use the interface. Releasing a new major release can effectively signal that continuity of the old interface should not be expected and that radical changes may be present. However, when this mechanism is only partially used, which we have shown is the case in the Maven repository, it becomes unclear what exactly a major release means.

A possible explanation for the low adherence to `semver` is that the Java modularization mechanism is not suited to provide all visibility levels as desired by developers. For instance, developers sometimes release "internal" packages. These are packages that should be hidden from outside developers and are only meant to be used by the developers themselves. The problem with internal packages is that they are publicly visible, meaning that outside developers have complete access to these packages, just like regular packages. What is missing from the Java language is another layer of visibility, which hides internal packages from outside users. An example of a mechanism that does provide this level of visibility is the modularization structure of the OSGi framework. Additionally, entire libraries are sometimes released that are only meant to be used by the developers themselves, even without the use of internal packages. Java or the Maven repository also do not provide support to prevent external users from using these libraries. In fact, these libraries should have never been released in the Maven repository to begin with.

The low number of methods that use the deprecation tag in the entire Maven repository was surprising. A possible explanation for this is that classes can also be deprecated completely, without individually deprecating all methods in that class. Our analysis will not detect these cases. Future work could further investigate whether developers deprecate entire classes instead of deprecating only single methods.

B. Actual usage frequencies

In our research, we do not take into account the difference between internal and non-internal packages. We also do not take into account the actual usage of packages, classes and methods with breaking changes. It makes a difference

whether a public method in the interface of a library is used frequently by other developers, such as `AssertEquals` in JUnit, or the method is not used at all by other developers. We consider the impact of breaking changes on libraries using that functionality outside the scope of this paper. However, `semver` does not state that breaking changes in major releases can only occur in parts of the library that are never used, but instead states that breaking changes should never be present in minor and patch releases, regardless of actual usage.

C. Release interval and edit script size

Table 4 showed that major releases have smaller release intervals and also contain less functional change. We expected that major releases have larger release intervals instead. This could be explained by the fact that developers often start working on a major release alongside the minor or patch release (by creating a branch) of the previous version, which would decrease the actual release interval.

The table also shows that major releases generally contain less changed functionality than minor releases, as measured by edit script size. A possible explanation for this is that developers create a new major release especially for backward incompatible changes in its API, and new functionality is added later. Seen this way, a major release can be interpreted as a signal that gives information on significant changes in the interface of a library, while saying nothing about the amount of changed functionality in the release.

D. The birth of semantic versioning principles

The snapshot of the Maven repository that was analyzed in this paper contained releases until July 11, 2011. The commit history of the GitHub repository of `semver.org`[15] showed that the first commit was performed on December 14, 2009. The question rises how widespread the knowledge about `semver` was before the first version of `semver` was online.

It is unclear when semantic versioning principles were started to be used by developers, but we believe that the principles on `semver.org` are simply a summary of principles that were already known in the developer community, but had not been encoded in a comprehensive manifesto before. This hypothesis is supported by the fact that comparable semantic versioning principles have been encoded elsewhere, such as the one by the OSGi alliance[16], which released their semantic versioning principles on may 6, 2010 and which contains comparable guidelines as the ones by `semver`.

Furthermore, there exist several alternative versioning approaches[17], but the versioning schemes described in these approaches do not seem to be used in the Maven repository, as can be seen in Table 1. For this reason, only adherence to the principles stated by `semver` was checked in this paper.

E. Major version 0 releases

`Semver` states that "*Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.*". We did not consider whether the effects as tested in this paper also hold for releases with a major version of zero. The number of releases having a major version of 0 is 10.44% (13,162 / 126,070), which is a substantial part of all releases. Future work could investigate whether the principles as tested in this paper are also visible in releases with a major version of 0. We expect that the number of breaking changes in these releases will be considerably higher than other releases.

XI. Threats to Validity

A. Internal validity

The release dates of libraries as obtained from the central Maven repository are sometimes incorrect, as demonstrated by the disproportionally large number of libraries with a release date of November 5th, 2005 (2,321, 1.5%). These data points were excluded from our analysis, but we do not have absolute certainty of the correctness of the remaining release dates. Another indication that release dates were not always correct is the fact that an ordering based on release dates and an ordering based on version numbers of a single artifact does not always give the same rankings. In these cases, the ordering in version numbers was assumed to be correct. These possibly invalid data points do influence our analysis on the number of days between releases, however, but we assume that on average, our statistical analyses provides us with a robust average. A manually checked sample of 50 random library versions and their release dates on the corresponding websites were all correct. This sample gives us confidence in the overall reliability of the release dates in the repository.

The low number of deprecation tags detected in the Maven repository is surprising. However, we have confidence in our methodology to detect these tags since deprecation patterns were scanned in two different ways. First, a textual search was performed to search for literal occurences of the string "`@Deprecated`". Second, when a deprecated tag was found in a library, the complete library was parsed and and AST's were created. This approach therefore makes it impossible to miss a deprecated tag. In future work, we could further investigate causes for the low number of deprecated tags.

B. External validity

While our findings are directly based on an exploration of semantic versioning principles in Maven, we believe many of them will hold beyond this setting. For example, in other

[15] https://github.com/mojombo/semver.org/commits/gh-pages?page=5
[16] http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf
[17] http://en.wikipedia.org/wiki/Software_versioning

eco-systems, such as .NET libraries, `nuget`[18] packages (the .NET counterpart of Maven), OSGi bundles, or Ruby gems[19], similar phenomena may be observed. In the domain of software services, versioning and compatibility play a role not just at compile time, but also at runtime, as services may be dynamically replace with (hopefully compatible) updates. Here, again, a need for dealing with breaking changes will occur, as well as a need for managing this through deprecation tags.

Future work could investigate to what degree the patterns found in our dataset are representative for software libraries outside the Maven repository, software libraries written in other languages than Java or software systems in general. To test our hypothesis that other library repositories also show the same patterns, further research is needed. Future work could also replicate the same patterns in a set of industrial software systems.

C. Reproducability and reliability

There was substantial computing power involved to obtain data for this paper: data was obtained on a supercomputer with 100 processing nodes with an aggregated running time of almost six months. Without access to the same amount of computing power, the data will be very hard to reproduce.

XII. Conclusion

In this paper, we have looked versioning as adopted by over 22,000 open source libraries distributed through Maven Central. In particular, we investigated whether principles as formulated by semantic versioning are adhered to, which specifies rules about the introduction of breaking changes in relation to version number increments.

Our findings are as follows:

- The introduction of breaking changes is widespread: Around one third of all releases introduce at least one breaking change.
- While semantic versioning prescribes that breaking changes are only permitted in major releases, we see little difference between these two: One third of the major as well as one third of the minor releases introduce at least one breaking change.
- The presence of breaking changes has little influence on the actual delay between the availability of a library and the use of the newer version of that library.
- Deprecation tags are used very little, and *never* in the way as strictly suggested by semantic versioning.

The results indicate that the current mechanisms to signal interface instability are not used properly.

[18]http://www.nuget.org

[19]http://www.rubygems.org

References

[1] J. Bauml and P. Brada. Automated versioning in OSGi: A mechanism for component software consistency guarantee. In *Proceedings of the 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '09, pages 428–435, 2009.

[2] B. E. Cossette and R. J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 55:1–55:11, New York, NY, USA, 2012. ACM.

[3] I. Şavga and M. Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 175–184, 2007.

[4] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 481–490, 2008.

[5] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 183–192, 2011.

[6] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage - determining the provenance of software development artifacts. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[7] J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *CSMR-WCRE*, pages 64–73. IEEE, 2014.

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 404–428, 2006.

[9] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, 2006.

[10] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[11] J. Ossher, H. Sajnani, and C. Lopes. Astra: Bottom-up construction of structured artifact repositories. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 41–50, 2012.

[12] S. Raemaekers, A. v. Deursen, and J. Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 221–224, 2013.