API evolution on Maven Central: do developers adhere to semantic versioning?

Simcha Vos
Supervisors: Mehdi Keshani, Sebastian Proksch
EEMCS, Delft University of Technology, The Netherlands

21-06-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

API evolution on Maven Central: do developers adhere to semantic versioning?

Simcha Vos
TU Delft
Faculty of EEMCS
Delft, The Netherlands

Supervisor: Mehdi Keshani
TU Delft
Faculty of EEMCS
Delft, The Netherlands

Professor: Sebastian Proksch
TU Delft
Faculty of EEMCS
Delft, The Netherlands

***Abstract*—In this paper, we investigate whether developers of artifacts on Maven Central adhere to semantic versioning. We also investigate whether there is a link between violations in semantic versioning and the popularity of the violating method. Developers can violate semantic versioning by removing or altering methods in their API, which we refer to as breaking changes. They can also violate semantic versioning by extending the API in a patch version, referred to as an illegal API extension. APIs that do not keep their promise of adhering to semantic versioning, will unexpectedly break their dependents during upgrading of dependencies.**

We have found that these two types of violations do occur in practice. We find that 24% of analyzed artifacts contain breaking changes and 24% of artifacts contain illegal API extensions. Finally, we show that popularity of a method does not have an impact on breaking changes.

We conclude that semantic versioning can not always guarantee that upgrading dependencies will not lead to incompatibility. This indicates a need for developers to be more aware of the impact that violating semantic versioning has.

***Index Terms*—Maven Central, compatibility, semantic versioning, API evolution, breaking changes**

I. Introduction

Maven Central is the most popular repository that distributes JVM-based artifacts [1]. A noteworthy aspect of Maven Central is that it accumulates all the versions of published artifacts, allowing one to analyze the evolution of an artifact. Artifacts hosted on Maven Central may include dependencies that point towards other artifacts. In this paper, we refer to artifacts as the unique combination of a *groupId* and an *artifactId*, which may have any number of versions associated with them.

In this study, we investigate APIs hosted on Maven Central. Over time, these APIs may evolve in their interface and functionality, which is a concept called API evolution. We investigate these changes on the method-level.

This paper looks at compatibility through different versions of an API, and investigates whether developers adhere to semantic versioning. Semantic versioning consists of three parts [2]: (1) major version: increases when new functionality is added, possibly breaking the API; (2) minor version: increases when new functionality is added and remains backward-compatible[1]; and (3) patch version: increases when backward-compatible bug fixes are added and the API is not changed.

[1]Backward-compatible: upgrading from an older version to a newer version which is backward-compatible can not lead to any compatibility issues.

In this study, we investigate two types of semantic versioning violations. Firstly, we investigate the alteration of the API in a minor or patch version, for example, the deletion of a method that is part of the API, which we refer to as breaking changes. Secondly, we investigate the addition of new functionality by extending the API in a patch version, which we refer to as illegal API extensions. Both of these are violations of semantic versioning.

The first violation is related to the fact that compatibility should only be broken after a so-called major release. In case a change breaks this compatibility, we call this change a breaking change if it occurs within the current major version.

The second violation is the extension of the API in a patch version. When this happens, new functionality is added and the API is changed. This is not allowed within a patch version.

While Kim et al. [3] and others have investigated the changes to an API through its evolution, as well as the reason and extent of these changes, most have not connected API evolution to the concept of semantic versioning. Understanding compatibility and adherence to semantic versioning is important, as an analysis of these concepts allows developers to know whether they can trust an API to remain compatible. Therefore, in this paper, we investigate whether semantic versioning can be trusted to guarantee compatibility.

Finally, we connect the popularity of methods to violations of semantic versioning. By analyzing which methods violate the principles of semantic versioning, we can tell whether there is a link between popularity and violating semantic versioning.

In this paper, we look at whether the evolution of an API generally leads to incompatibility. In addition, we investigate whether these changes in the API are in line with the principles of semantic versioning. This means that methods may only be removed in major versions and the API may only be extended in minor or major versions.

We analyze a representative set of 356 artifacts, each with some number of different versions. The contributions of this study are as follows:

- a quantitative analysis of the compatibility across versions of APIs concluding whether semantic versioning can be trusted to guarantee compatibility across minor and patch releases;
- a release of the tools of this study for public use.

II. Related works

To find the current gap in research, we will first study several works that are related to this study.

Mileva et al. [4] have employed what they call *wisdom of the crowds* in order to analyze the usage of individual library versions. Their paper presents an approach to support developers in their decision to upgrade a dependency. To this end, they mined the dependencies of 250 APACHE projects and consider the choice of the majority to come to a recommendation. Their findings are that trends in software will result in immediately useful recommendations.

Nguyen et al. [5] presents a system called LibSync, which is a tool for developers who want to upgrade their dependencies. It suggests users a way of adapting their API usage by learning from clients that have already migrated to a new library version. The system looks at the API usage at the method-level and attempts to suggest to users a way to deal with changes in method declarations.

Kim et al. [3] have performed a large analysis of API refactorings and bug fixes and come to several interesting conclusions. Firstly, the presence of an increase in the number of bug fixes after refactorings, secondly that it takes shorter to fix bugs after refactorings. Another observation they make is that refactoring revisions often include bug fixes or are related to later bug fix revisions, and finally that refactorings occur more frequently before than after major software releases. This study offers insights into the consequences of refactorings, but it does not focus on versioning of the library, as it looks at the commit history of Git repositories.

Hora et al. [6] reason that during software evolution, the source code of APIs is constantly refactored. The researchers propose a tool to extract rules by monitoring API invocation changes. This tool mines the changes at revision-level in the code history and can be used to keep track of the evolution of an API. The paper is therefore also concerned with monitoring the evolution of an API, which is something we will also study in this work.

Hora et al. [7] have performed an exploratory study aimed at observing API evolution and its impact on a large software ecosystem. The found information can be used to find the best way to alleviate the large impact of API evolution. They use multiple metrics to analyze the evolution, for example, changes in method declarations. The paper does consider more metrics than this study does, and it is focused on the impact on one large software ecosystem, which is something we will not restrict ourselves to in this study.

Koçi et al. [8] has investigated changes that happen to APIs and has identified and classified them to gain a bigger picture of API evolution. The paper defines a classification framework that considers the changes and the reason behind them. Our study does not concern itself with any classifying of API changes, however, both take a look at the history of APIs.

Macho et al. [9] has analyzed trends concerning changes in Maven build files. They present an approach called BuildDiff, that can be used to observe trends, such as frequently co-occurring changes, changes to the dependency management system, or dependency declarations. The study sets a foundation for future research, such as studies analyzing the evolution of build files.

Ochoa et al. [10] investigate breaking changes in APIs, which are changes that might trigger at any time and will cause clients to be hesitant with upgrading their dependencies. They note that conclusions drawn by studies like Hora et al. [7] do not hold outside the ecosystem in which the study was performed. The researchers have conducted an external and differentiated replication study of 2017's Raemaekers et al. [2] study. They contrast the original study and show that 83.4% of upgrades comply with semantic versioning and that the tendency to comply with semantic versioning has significantly increased over time. In addition, they find that most breaking changes affect code that is not used by any client, and that only 7.9% of clients are affected by breaking changes.

We conclude that research on adherence to semantic versioning on Maven Central is very limited, whilst it is one of the largest and oldest public repositories. Next to this, research has only investigated the first category of violations, breaking changes. It has not looked at the second category, where the API is extended in a patch upgrade. In addition, research has not connected this concept of adherence to semantic versioning to the popularity of the respective methods that are involved in breaking changes. These are the areas on which this study will be focused.

III. Study approach

In order to investigate the evolution of compatibility and adherence to semantic versioning, this work is articulated around three research questions. After describing the motivation for these research questions, we discuss the methodology used to answer these questions.

A. Research questions

The main interest of this study is the evolution of an API over time.

In order to study this, we answer three research questions. These research questions focus on adherence to semantic versioning and the connection between adherence and popularity.

The first question concerns itself with violations of semantic versioning through breaking changes, which are changes that break compatibility:

> *RQ1:* How often do breaking changes occur during upgrades of minor and patch versions, and to what extent do they happen?

We expect that the evolution of an API sometimes leads to incompatibility, due to developers removing or changing the existing set of method signatures of an artifact. If this is indeed the case, we should take a closer look at semantic versioning, and investigate whether the breaking changes only occur between major versions, or also within the same major version.

The second category of violations of semantic versioning is the extension of an API through patch versions. We investigate this category by answering the second research question:

RQ2: How often do extensions of API occur during patch version upgrades, and to what extent do they happen?

When we have answered both of the first two research questions, we have an insight into how often violations of semantic versioning occur in general.

Finally, we can combine semantic versioning and popularity to answer our last research question. It is particularly interesting to involve the popularity of methods. That is because the deletion of an unpopular method does not nearly give way to as many consequences as the deletion of a very popular method. To investigate this, we look at minor releases or patches within the same major release, and verify whether any breaking changes happen in these releases. Afterward, we can check whether these breaking changes occur in methods that are popular or less popular. We do this by answering the final research question:

RQ3: Does popularity of a method influence the appearance of breaking changes?

B. Methodology

In this section, we explain which methods we use to answer the three research questions.

1) RQ1: For the first research question, we concern ourselves with versions that introduce breaking changes.

To answer this question, we should precisely define breaking changes. Breaking changes are changes within a major version that break compatibility. This means that, when a new minor or patch version is released, a public method signature has been altered or removed. This leads to issues in case this method is called by some artifact. Therefore, changes that remove method signatures are not allowed, unless it is part of a major version upgrade.

We have to find a way to detect evolution that makes changes in the existing set of method signatures. Our approach looks at the set of method signatures, and finds the difference between the current version's set of method signatures and the previous version's set of method signatures:

$breakingChanges = methodSignatures_n - methodSignatures_{n+1}$

where n and $n+1$ are successive versions within the same major version.

Once we have found this difference, we know the set of method signatures that are involved in a breaking change. We categorize using a combination of the artifact and the corresponding major version.

2) RQ2: Now we look into the second category of violations: extending the API in a patch version.

We can use a similar approach as the first research question, however, we should make some small adaptations to the codebase. We iterate along the different patch versions, and detect if a new method signature is added. In this case, there has been an extension of the API within a patch version. We find illegal API extensions by finding the difference between the next version's set of method signatures and the current version's set of method signatures:

$illegalAPIExtensions = methodSignatures_{n+1} - methodSignatures_n$

where n and $n+1$ are successive versions within the same minor version.

Once more, we categorize the resulting set of illegal API extensions with the same combination of artifact and corresponding major version.

3) RQ3: To answer the final research question, we once again extrapolate a set of method signatures that break the semantic versioning of an artifact, according to the first and second research questions.

The most problematic category of violations is the breaking changes within the same major version, as according to semantic versioning, developers should be able to rely on the fact that upgrading a minor or patch version is not going to break the compatibility.

We will use a metric that produces the popularity of a method of an artifacts [11]. We can use this metric to learn how often it happens that a developer encounters breaking changes when upgrading minor or patch releases of an artifact.

Finally, we combine the methods, those involved in a breaking change and those that are not involved in a breaking change, with their respective popularity. Plotting this in a graph shows whether there is indeed a connection between popularity and involvement in a breaking change.

IV. Experimental setup

To gather our results we use a FASTEN[2] server which contains information about the past six months of Maven artifact releases.

To answer the three research questions we have created a Java package that uses the data obtained from the FASTEN server to execute the described approach in the methodology. The data we use from the server is described in more detail in section V.

The produced results are represented by a text file containing hundreds of entries, each giving information about a specific major version of an artifact.

The entries consist of the *groupId* and *artifactId*, followed by the major version. Then, the results are printed, consisting of the number of violations followed by the total number of different method signatures that were involved in the violation. Afterward, the unique identification numbers of these methods are printed. These methods have caused a violation, meaning they were either illegally removed in a minor or patch release, or illegally added in a patch release.

In order to investigate the impact of method popularity on the number of breaking changes, we make use of a metric which outputs the popularity of method.

[2]FASTEN: a dependency analysis framework that amongst others enables one to analyze dependencies between APIs. Available at https://github.com/fasten-project/fasten.

Finally, we make use of Python to explore the produced results and create graphs.

A reproducible package is available on the Docker Hub[3], which allows one to replicate the results of this study.

V. Data selection

For this study, we have used the last six months of artifacts available to us on the FASTEN server, and remove all artifacts related to testing. As Maven does not store any test code of its artifacts, we know nothing about the usage of artifacts that offer testing suites. That is because Maven offers close to zero calls to these testing artifacts.

Finally, through manual inspection, we have removed several artifacts that turned out to feature very high numbers of breaking changes. These were artifacts with an extremely rare and odd structure. For example, one of the artifacts had a structure with nested data types within Scala class files. Another artifact featured a library of packages within their artifact, which is not part of the API of this artifact. These artifacts had to be manually removed from the data set, as within this study, we want to only investigate the actual API of the artifacts.

The data set contains 356 different artifacts, each featuring up to seven different major versions. 286 artifacts of the complete set of artifacts have just one major version. In total, we analyze 2766 different versions, consisting of all the major, minor, and patch releases of the 356 artifacts, all published on Maven Central over the past six months.

VI. Results

Table I shows a readable representation of the output file, with several interesting entries. Firstly, one might notice that the SPRING-JDBC artifact has two major versions shown in the table, with the second one featuring a notable decrease in the number of breaking changes. Another artifact, TRUFFLE-API, has a huge number of breaking changes compared to the total number of unique methods, whilst artifacts like MYSQL-CONNECTOR-JAVA feature not a single breaking change in all their methods. TRUFFLE-API makes use of a versioning scheme with four digits. Their patch release 21.0.0.2 suddenly introduces a huge number of breaking changes, as well as adding a large number of methods. These numbers correspond with a major version upgrade. This artifact is one of the artifacts that was removed during the selection of data, as this outlier would skew the results.

One can notice that we differentiate between different major versions of the same artifact. This is because we have noticed some artifacts start using semantic versioning at a later point in their development. For example, we have found an artifact that did not make use of semantic versioning for its first major version but started using it correctly during the development of its second major version. In case we would not differentiate between these major versions, it would be impossible to notice different adherence during the development of different major versions.

A. Breaking changes in minor or patch releases

Now we investigate how often breaking changes occur, and if they occur, how many breaking changes exist per artifact. First, figure 1 shows the ratio of breaking changes compared to how many major versions feature such a ratio. 286 major versions have a ratio of around zero breaking changes compared to the number of methods. In total, the percentage of artifacts featuring breaking changes is 24%. In addition, we can notice that if we look at the higher ratios, only a small number of major versions have such a high ratio. In fact, barely any artifacts feature a breaking change ratio higher than 10%.

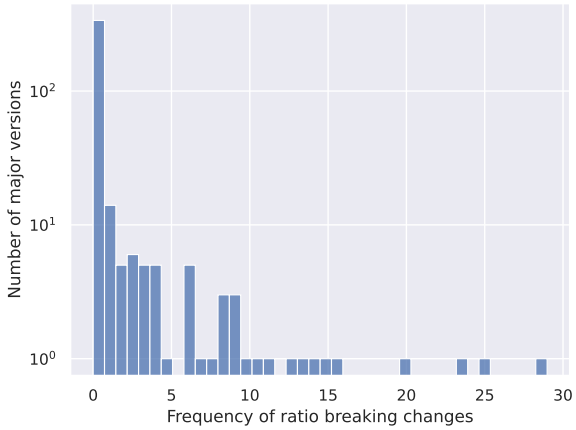


Fig. 1: Frequency of ratio of breaking changes over respective number of methods

Figure 2 contains a violin plot that shows the average ratio of breaking changes per major version. The input data of this figure consists only of versions that contain breaking changes, as the largest part of the data does not contain breaking changes, which would otherwise skew the plot towards zero. The graph shows that 75% of all artifacts feature a ratio that is not bigger than 0.15. However, several outliers exist which feature a noticeably higher ratio, indicating that these artifacts contain a lot of breaking changes.

Another figure, 3, shows the total number of methods against the total number of breaking changes, in order to see if a higher number of methods also leads to a higher number of breaking changes. This figure also shows a trend line with a 95% confidence interval. The figure shows that indeed a larger number of methods leads to a larger number of breaking changes. However, one can observe that the number of breaking changes does not grow as rapidly along with the number of methods.

B. Method addition in patch releases

Now we show the results of the analysis on the number of methods that were added in patch releases. In figure 1,

[3]The reproducible package is available at https://hub.docker.com/r/simayy/maven-reproducible-package.

groupId:artifactId	Major version	Breaking changes	Illegal API extensions	Number of methods
org.springframework:spring-jdbc	2	66	123	886
org.springframework:spring-jdbc	3	17	65	1368
org.graalvm.truffle:truffle-api	21	1369	8759	10501
mysql:mysql-connector-java	8	0	0	14545

TABLE I: Breaking changes and illegal API extensions of some artifacts

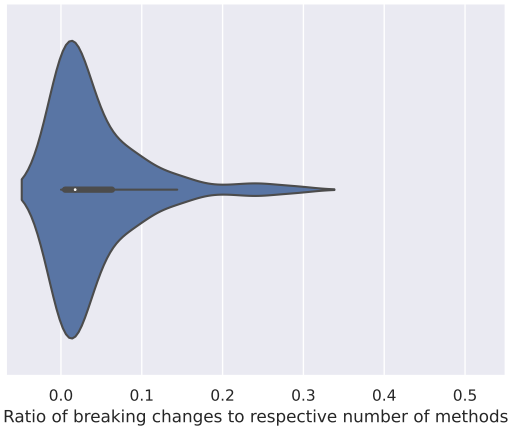


Fig. 2: Average ratio of breaking changes in total number of methods for artifacts that feature one or more breaking changes

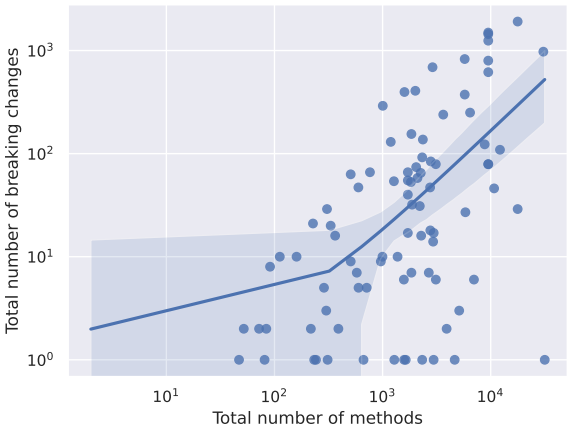


Fig. 3: Increase in number of breaking changes in relation to total number of methods

the number of artifacts is plotted against the number of API extensions. One may notice the distribution is quite similar to figure 4, however, the number of API extensions is overall a little lower than the number of breaking changes. The percentage of artifacts which feature illegal API extensions is also 24%, but the set of artifacts is not equal to the set of artifacts that feature breaking changes.

The violin plot in figure 5 agrees with this observation, as the 75th percentile is somewhat lower than figure 2, coming in at a ratio of around 0.1. This means that most of the artifacts do

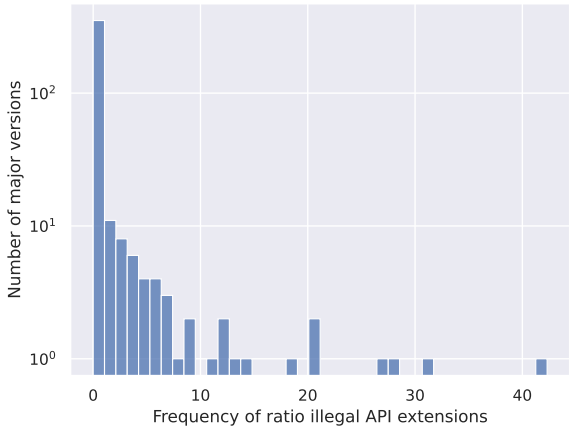


Fig. 4: Frequency of ratio of illegal API extensions over respective number of methods

not feature more than 10% illegal API extensions, compared to the total number of methods of the artifact. This plot confirms that illegal API extensions happen slightly less often than breaking changes. However, some outliers can be observed, which are generally higher than in figure 2.

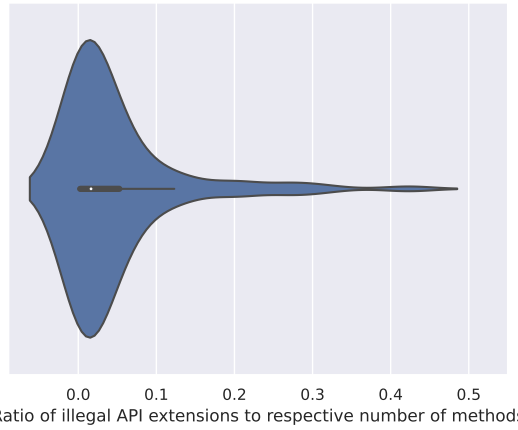


Fig. 5: Average ratio of illegal API extensions in total number of methods for artifacts that feature one or more illegal API extensions

Finally, we can notice from figure 6 that the number of API extensions also grows along with the number of methods. However, just as with the number of breaking changes, this number of API extensions grows a little slower than the

number of methods. Compared to figure 3, we notice that the number of illegal API extensions grows a little slower than the number of breaking changes if we compare these violations to the total number of methods of an artifact.

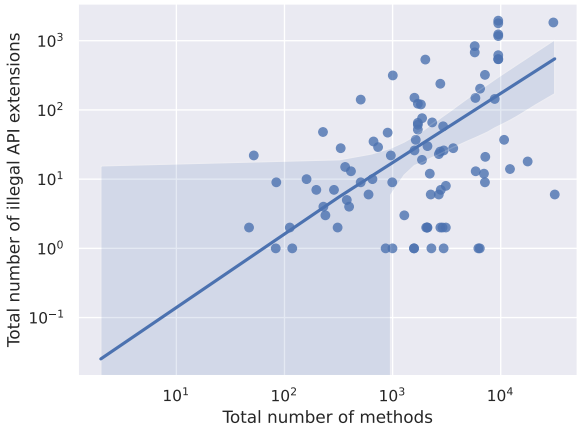


Fig. 6: Increase in number of API extensions in relation to total number of methods

C. Relation between popularity and involvement of a breaking change

Figure 7 shows the popularity of every method compared to the popularities of methods involved in a breaking change. The popularity is represented by the percentage of dependents which call the method. In case the x-axis is equal to one, the method is called by every dependent.

Both these figures do not include methods with a dependent percentage of zero percent. This is a fairly big portion of the popularity scores. We do not include these data points as these methods are generally in disuse. Including them will not be representative, as they will skew the density of methods which do not have a popularity of zero.

Figure 7a shows the popularity distribution of every method as well as the popularity distribution of all the methods involved in a breaking change. We notice the distributions are quite similar.

In figure 7b, we can compare the different graphs by investigating the difference in density at a certain point of the x-axis. If the graph with breaking changes has a higher density than the graph without breaking changes on a certain point x, this means that the popularity of the methods is distributed in a way that on average, a method with a breaking change more often has popularity x than a method without a breaking change.

However, in this figure it can be noted that both lines follow a very similar pattern. Whether the method is involved in a breaking change or not does not seem to be related to the popularity.

The graph shows us that unpopular methods are more often methods that are involved in breaking changes. Additionally, when we investigate more popular methods, for example, the methods which are called by more than half of the dependents of their artifact, we notice that these methods are more likely not to be involved in a breaking change than conversely.

VII. Discussion

In this section, we first interpret the results. Afterward, we discuss the implications of this study. Then, we introduce limitations and suggest possible future work. Finally, we shortly go into the ethical consequences of this study.

A. Interpretations

While a large number of artifacts do not feature any breaking changes, there is a large number of artifacts that do contain some number of breaking changes. This means that developers are going to run into issues when they unsuspectingly attempt to upgrade their dependencies, as the API is no longer compatible with their artifact.

The second category of violations – illegal API extensions – occurs somewhat less often. We have also seen that the increase in illegal API extensions is related to the total number of methods. An important thing to realize is that the addition of an extra method does not pose a threat to the compatibility of said artifact. Therefore, developers need to be less concerned with these illegal API extensions when upgrading their dependencies. However, the developers which extend their API illegally are employing bad coding practices by violating the principles of semantic versioning.

Finally, there does not seem to be a relation between the popularity of a method and whether it is involved in a breaking change or not. It could be concluded that developers are not interested in popularity of their methods when they choose to introduce breaking changes – if they have even made a conscious choice. However, there might still be some factors at stake, which possibly cancel each other out.

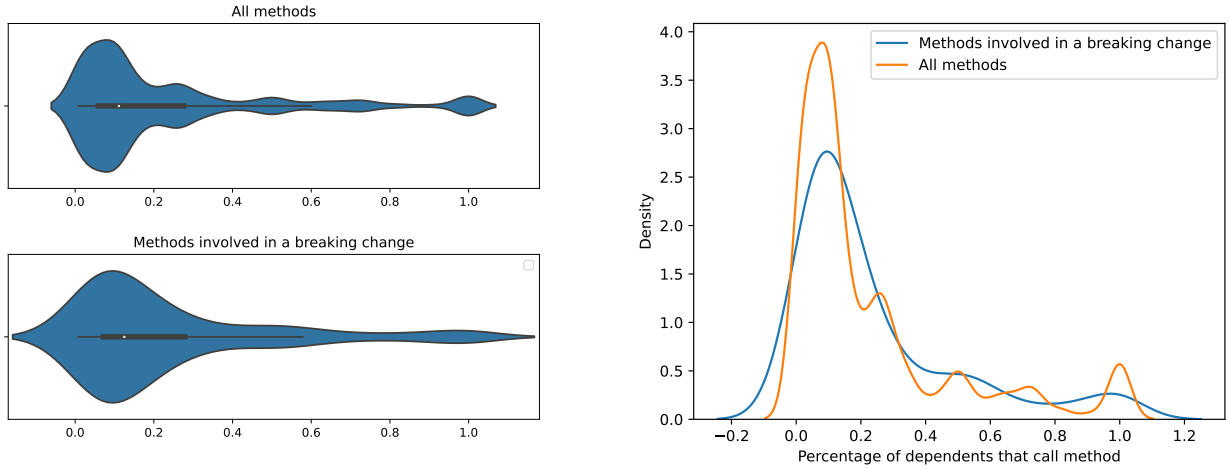
Methods that are more popular might gain more traction in the creation of pull requests. This can cause developers to notice problems with method signatures sooner than they would have if the method was unpopular.

On the other hand, the developer might realize that a method is popular, and be wary of the fact that altering the method signature could affect a lot of dependents.

It is good to realize that, when an artifact contains breaking changes, it is not guaranteed to cause problems. If the dependent does not use the methods which are involved in a breaking change, the dependent will not be broken.

B. Implications

It has become clear that breaking changes are a big part of the dependency upgrading process. While this study has shown that the majority of developers attempt to introduce zero breaking changes during their development, we have seen that a fair number of artifacts are not without violations of semantic versioning. This means that developers should remain careful while upgrading their dependencies.



(a) Violin plots

(b) Kernel Density Estimates

Fig. 7: Distribution of percentage of dependents that call methods

We expected that illegal API extensions would be present. This study could serve as a reminder to avoid the bad practice of introducing these API extensions in patch versions.

This study also investigated the popularity of the methods that are involved in a breaking change. We have shown that popularity does not have an impact on breaking changes. This serves as an encouragement and reminder for developers to avoid breaking changes, for methods that are in disuse but even more for methods which are popular.

The insights of this paper serve as a reminder to uphold the principles of semantic versioning, and to stimulate compatibility, so developers do not have to worry about broken dependencies.

The presence of these kinds of violations of semantic versioning indicate a need for more tools, either on the client side, or on platforms such as Maven Central. For example, build tools could offer warnings when they detect an inappropriate version upgrade. IDEs (integrated development environment) could recommend users to adjust their version appropriately whenever a change in method signatures is detected. Finally, Maven Central could warn developers for the appearance of incompatibilities, or impose strict requirements for versioning of artifacts. This would create a safer and trustworthier environment within the ecosystem of the repository. Developers would be able to put trust in the fact that standards of semantic versioning are upheld within Maven Central or other platforms, whenever they upgrade their dependencies.

C. Limitations and future work

In this study, we limit our selection of data to the last six months of artifacts published on Maven Central. It could be suggested that the study would be more general if a larger selection of data would be used, possibly even spanning multiple decades of evolution featured on Maven Central. However, we would argue that this would involve a tremendous increase in cost, and lead to a less relevant bulk of data. It is in fact more interesting to see the latest trends of API evolution, as we want to produce insights into the current state of affairs related to the adherence to semantic versioning. However, future work could take a more broad look at the history of Maven Central, which would allow the researchers to look at statistics such as the trend of breaking changes between different major versions. Most of the artifacts of this study only featured one major version during the past six months.

Another limitation of this study is that we can only investigate breaking changes that are caused by changes to the set of method signatures. While our definition of breaking changes only involves alteration of method signatures within the same major version, another type of breaking change could be considered. In case the functionality of a method is changed, leading to unexpected behavior while the API remains intact, as the method signature is not changed. We expect that this type of breaking change is not a large factor and does not happen too often, however, we do not have the tools to analyze this. Such a tool should be able to analyze whether, given a certain input, the method always produces the same output, even though the code might be different. We imagine creating such a tool will be very complicated and have a very high time complexity. For this reason, this study has only factored in changes to method signatures. Future work could investigate the existence of such tools or create such tools, to gain insights about this type of breaking change.

D. Responsible research

This study does not introduce bias in the selection of our data, as it consists of a random sample of Maven Central, filtered on a small number of conditions. Results were accumulated on an anonymous basis. Results from this study might stimulate developers to adhere to semantic versioning. We expect no other side effects and no ethical risks to come

forth from this study. By offering a reproducible package, we invite readers to reproduce the data we have produced, to be able to verify that our results are correct.

VIII. Conclusions

We have showed that a large number of artifacts do not completely adhere to semantic versioning. We know this, as 24% of analyzed artifacts contained breaking changes, causing compatibility within a major version to be broken. Illegal extensions of an API also pose a problem in 24% of artifacts, as developers violate this principle of semantic versioning equally often as they push breaking changes. Therefore, it seems that both of these violations form a big problem for the trustworthiness of semantic versioning.

However, we must also realize that the deletion or alteration of an unpopular method does not have the same impact as changing a popular method. Therefore, we attempted to verify whether a relationship between popularity of a method and involvement in a breaking change exists. We have noticed that popularity of a method does not have an impact on breaking changes.

Developers that upgrade their dependencies need not worry about illegal extensions of the API. While this is a violation of semantic versioning, this violation does not pose a problem to the dependents as adding a method does not influence backward compatibility.

We conclude that, as several artifacts do not completely adhere to semantic versioning – resulting in problems with compatibility – a negative impact is made on the trustworthiness of semantic versioning. We have shown that only breaking changes form a problem for dependent artifacts, as illegal extensions of the API do not change compatibility. While breaking changes occur in a large number of artifacts, the chance of a dependent relying on a method involved in a breaking change is very small. Therefore we conclude that, in practice, developers of dependents should not run into problems very often. While semantic versioning cannot be trusted completely to guarantee compatibility, the chance of a broken dependency is quite slim.

Acknowledgment

I want to thank Mehdi Keshani very much for his continued support and dedication to help me during my research, particularly during some of the struggles I encountered. I want to give another word of thanks to Sebastian Proksch, especially for his ability to come up with ideas and opportunities during the research process. Finally, I want to thank my team members, especially Thijs Nulle, for their insights and for our successful collaboration.

References

[1] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. “The emergence of software diversity in maven central”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 333–343 (cit. on p. 1).

[2] Steven Raemaekers, Arie van Deursen, and Joost Visser. “Semantic versioning and impact of breaking changes in the Maven repository”. In: *Journal of Systems and Software* 129 (2017), pp. 140–158 (cit. on pp. 1, 2).

[3] Miryung Kim, Dongxiang Cai, and Sunghun Kim. “An empirical investigation into the role of API-level refactorings during software evolution”. In: *Proceedings of the 33rd international conference on software engineering*. 2011, pp. 151–160 (cit. on pp. 1, 2).

[4] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. “Mining trends of library usage”. In: *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. 2009, pp. 57–62 (cit. on p. 2).

[5] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. “A graph-based approach to API usage adaptation”. In: *ACM Sigplan Notices* 45.10 (2010), pp. 302–321 (cit. on p. 2).

[6] André Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, and Marco Tulio Valente. “Apievolutionminer: Keeping api evolution under control”. In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 420–424 (cit. on p. 2).

[7] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. “How do developers react to API evolution? A large-scale empirical study”. In: *Software Quality Journal* 26.1 (2018), pp. 161–191 (cit. on p. 2).

[8] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. “Classification of changes in API evolution”. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE. 2019, pp. 243–249 (cit. on p. 2).

[9] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. “The nature of build changes: An empirical study of Maven-based build systems”. In: *Empirical Software Engineering* 26.3 (2021) (cit. on p. 2).

[10] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. “Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central”. In: *arXiv preprint arXiv:2110.07889* (2021) (cit. on p. 2).

[11] Thijs Nulle. “Popularity Distribution of Methods within Software Artifacts”. 2022 (cit. on p. 3).