The Journal of Systems and Software 129 (2017) 140–158

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Semantic versioning and impact of breaking changes in the Maven repository

S. Raemaekers[a,b,*], A. van Deursen[b], J. Visser[c]

[a] ING, Haarlemmerweg, Amsterdam, The Netherlands
[b] Technical University Delft, Delft, The Netherlands
[c] Software Improvement Group, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 16 February 2015
Revised 20 February 2016
Accepted 6 April 2016
Available online 22 April 2016

Keywords:
Semantic versioning
Breaking changes
Software libraries

ABSTRACT

Systems that depend on third-party libraries may have to be updated when updates to these libraries become available in order to benefit from new functionality, security patches, bug fixes, or API improvements. However, often such changes come with changes to the existing interfaces of these libraries, possibly causing rework on the client system. In this paper, we investigate versioning practices in a set of more than 100,000 jar files from Maven Central, spanning over 7 years of history of more than 22,000 different libraries. We investigate to what degree versioning conventions are followed in this repository. Semantic versioning provides strict rules regarding major (breaking changes allowed), minor (no breaking changes allowed), and patch releases (only backward-compatible bug fixes allowed). We find that around one third of all releases introduce at least one breaking change. We perform an empirical study on potential rework caused by breaking changes in library releases and find that breaking changes have a significant impact on client libraries using the changed functionality. We find out that minor releases generally have larger release intervals than major releases. We also investigate the use of deprecation tags and find out that these tags are applied improperly in our dataset.

1. Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward 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.

As an approximation of the (undecidable) notion of backward compatibility, we use the concept of a *binary compatibility* as defined in the Java language specification. The Java Language Specification[2] states that *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.* This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. For the purpose of this paper, we define any change that does not maintain binary compatibility between releases to be a *breaking change*.

* Corresponding author at: Technical University Delft, Delft, The Netherlands. Tel.: +31626966234.
E-mail addresses: stevenraemaekers@gmail.com (S. Raemaekers), arie.vandeursen@tudelft.nl (A. van Deursen), j.visser@sig.eu (J. Visser).

[1] http://semver.org.

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

http://dx.doi.org/10.1016/j.jss.2016.04.008
0164-1212/

Examples of breaking changes are method removals and return type changes.[3]

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. Even though we do not expect that all developers that submit code to the Maven repository are aware of the guidelines of `semver`, we still expect that most developers are aware that most other developers perceive a difference in changing a patch, a minor or a major version number when releasing a library.

Semantic versioning principles were formulated in 2010 by (GitHub founder) Tom Preston–Werner, and GitHub actively promotes `semver` and encourages all 10,000,000 projects hosted by GitHub to adopt it. Similarly, the Maven Central repository, the repository used to collect dependencies that are specified using the build tool Maven, strongly recommends following `semver` when releasing new library versions.[4]

Semantic versioning principles have also been embraced in the Javascript community. An example of a Javascript project that explicitly announced to follow `semver` is jQuery, which state that *"the team has tried to walk the line between maintaining compatibility with code from the past versus supporting the best web development practices of the present"*.[5] Another example is NPM (Node Package Manager),[6] a build tool for Javascript similar to Maven, which *requires* users to follow `semver` when submitting a new version of a library.[7]

An example of a software project which demonstrates that including breaking changes in non-major releases causes problems for software developers is JUnit. In its 4.12-beta-1 release, JUnit introduced breaking changes as compared to its previous release. In version 4.12-beta-2, these breaking changes have been reversed after complaints of library users.[8]

Another example of problems that can occur when backward compatibility is ignored is NuGet.[9] NuGet is a build tool for .NET systems and a software repository for software libraries, which automatically includes the latest version of dependencies in software projects. This leads to problems when these releases contain breaking changes.[10]

Although the NuGet build system ignores backward compatibility problems of users of libraries, Microsoft suggests the following distinction between major and minor releases[11] for .NET software:

> *Major:* "A higher version number might indicate a major rewrite of a product where backward compatibility cannot be assumed."
>
> *Minor:* "If the name and major version number on two assemblies are the same, but the minor version number is different, this indicates significant enhancement with the intention of backward compatibility."

Although not all developers of the projects mentioned before may be aware of the semantic versioning standard or other official rules regarding incrementing major, minor or patch versions, a lot of library users implicitly assume that non-major releases should not include breaking changes. 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, in open-source Java libraries? Are breaking 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? Furthermore, for the breaking changes that do occur, to what extent are they signaled through, e.g., *deprecation tags*? 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.[12] 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 using other libraries in the dataset), permitting us to study the impact of incompatibilities in concrete clients as well.

This paper is a substantially revised version of our earlier analysis of semantic versioning practices in maven. In this paper, we extend this analysis with an assessment of the actual *impact* of breaking changes. To approximate this impact, we introduce a new method to inject breaking changes in library clients and analyze the prevalence and dispersion of compilation errors caused by these changes. This results in estimates of the number of errors caused by each type of breaking change.

This paper is structured as follows. We start out, in Section 2, by discussing related work in the area of binary incompatibilities and change impact analysis. In Section 3, we formulate the research questions we seek to answer. Then, in Section 4, we describe our approach to answer these questions, and how we measure, e.g., breaking changes, changed functionality, and deprecation. In Section 5–11 we present our analysis in full detail. We discuss the wider implications and the threats to the validity of our findings in Section 12 and 13, after which we conclude the paper in Section 14.

2. 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 (Dig and Johnson, 2006; Tempero et al., 2008; Dietrich et al., 2014; Cossette and Walker, 2012; McDonnell et al., 2013).

2.1. Manual investigations

Cossette and Walker (2012) perform a manual retroactive study on API incompatibilities to determine the correct adaptations to migrate from an older to a newer version of a library. They also aim to determine recommender techniques for specific update types. In contrast, our method to inject breaking changes can be performed automatically, and only gives a global indication of the amount of work required to perform an update in terms of the number of compilation errors and the number of places that have to be fixed. Our method does not provide any guidance how to perform an update but can point to places where work has to be performed.

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

[4] http://central.sonatype.org/pages/requirements.html.

[5] http://blog.jquery.com/2014/10/29/jquery-3-0-the-next-generations/.

[6] http://www.npmjs.com.

[7] https://docs.npmjs.com/getting-started/semantic-versioning.

[8] https://groups.yahoo.com/neo/groups/junit/conversations/topics/24572.

[9] https://www.nuget.org/.

[10] http://blog.nuget.org/20141010/nuget-is-broken.html.

[11] http://msdn.microsoft.com/en-us/library/system.version%28v=vs.110%29.aspx.

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

Similarly, Dig and Johnson (2006) investigate binary incompatibilities in five other libraries and conclude that most of the backward incompatible API changes are behavior-preserving refactorings, which suggests that refactoring-based migration tools should be used to update applications. Dietrich et al. (2014) 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.

2.2. Automated suggestions

Another area of active research is to automatically detect refactorings based on changes in public interfaces (Şavga and Rudolf, 2007; Dagenais and Robillard, 2008; Dig et al., 2006; Henkel and Diwan, 2005; Xing and Stroulia, 2007; Balaban et al., 2005; Kapur et al., 2010). 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. Dagenais and Robillard (2008), for example, present a recommendation system that suggests adaptations to client programs by analyzing how a framework adapts to its own changes. Similarly, the tool of Xing and Stroulia (2007) uses framework usage examples to propose ways to upgrade to a new version of a library interface.

While our work investigates backward incompatibilities for given version string changes, Bauml and Brada (2009) 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.

2.3. Maven repository

The Maven repository has been used in other work as well. Davies et al. (2011) 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. (2012) 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. This can be useful because manually curating a repository such as Maven Central is an error-prone and time-consuming process.

2.4. Change impact analysis techniques

The methodology that we use to inject breaking changes and determine the impact of these changes can be regarded as a change impact analysis technique, for which there already exist several alternative approaches (Ren et al., 2005; Badri et al., 2005; Zhou et al., 2008). For instance, call graph analysis techniques can obtain a graph that can point developers to places where rework is expected, such as done by Ren et al. (2005). Other techniques use correlations of file properties or historically changed file pairs as a basis to determine files that are likely to change together, as in Zimmermann et al. (2005). For an overview of change impact analysis techniques, see Lehnert (2011).

Our automated change injection mechanism also bears similarities to approaches applied in the field of automated software testing and, more specifically, error injection. Error injection techniques inject faults to find out if the resulting errors are covered by test cases. The goal of this paper is different, however: we want to determine the amount of rework caused by applying library updates. For an overview of error injection techniques, see Duraes and Madeira (2006).

2.5. Other work

Issues with backward incompatibilities can also be found in web interfaces. Romano and Pinzger (2012) investigate changes in the context of service oriented architectures, in which a web interface is considered to be a contract between subscribers and providers. These interfaces are shown to suffer from the same type of problems as investigated in this paper, which leads to rework on the side of the subscribers of these interfaces. The authors propose a tool that compares subsequent versions of these web interfaces to automatically extract changes.

Developer reactions to API deprecations has been investigated for the Smalltalk language and ecosystem by Robbes et al. (2012). They have investigated a set of more than 2600 distinct Smalltalk systems which contained 577 deprecated methods and 186 deprecated classes, and found that API changes caused by deprecation can have a large impact on developers using that API.

Complete migrations to other libraries providing similar functionality has been investigated by Teyton et al. (2014). In contrast to our work, Teyton et al. are concerned with a migration between different libraries performing similar functionality, rather than a migration between different versions of the same library.

In previous work (Raemaekers et al., 2013), we empirically investigated the relationship between changes in dependencies and changes in systems using these dependencies. The difference with our previous approach is that we distinguish between different types of library updates, and that we use the edit script size as a measure for rework, which more accurately measures the difference between methods than the difference in LOC as used in our previous work.

3. Research questions

The overall goal of this paper is to understand to what degree developers of software libraries use versioning conventions in the development of these libraries, and what the impact of unstable interfaces is on clients using these libraries. We investigate instability of interfaces through the number of compilation errors caused by breaking changes and the dispersion of these errors through libraries using the changed interfaces.

Even though not all developers might be aware of the `semver` standard, we still regard `semver` as a formalization of principles that are considered to be best practices, even before the manifesto was released in 2010. As mentioned before, the prime example of such a best practice is not to include breaking changes in major releases.

In this paper, we seek to answer the following research questions:

- *RQ1*: How are semantic versioning principles applied in practice in the Maven repository in terms of breaking changes?
- *RQ2*: What is the impact of breaking changes in terms of compilation errors?
- *RQ3*: Has the adherence to semantic versioning principles increased over time?
- *RQ4*: How are dependencies actually updated in practice, what are typical properties of new library releases, and do these

properties influence the speed with which dependencies get updated?[13]
- *RQ5*: Which library characteristics are shared by libraries which frequently introduce a large number of breaking changes, and as a result, cause compilation errors?
- *RQ6*: How are deprecation tags applied to methods in the Maven repository?
- *RQ7*: What is the impact of breaking changes in terms of the spread of errors caused by these changes?

to answer these questions, a wide range of different kinds of data is required. This data is gathered from our repository using different methods, which are described in the next section.

4. Maven analysis approach

In this paper, we analyze a snapshot of the Maven's Central Repository, dated July 11, 2011.[14] 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 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 144,934 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, we refer to Raemaekers et al. (2013).

4.1. 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.

To detect breaking changes between each subsequent pair of library versions, we use Clirr.[15] 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.

In this paper, we only investigate differences between *subsequent* releases of a library and we do not compare previous major releases or minor releases with each other. For instance, when a library has released version 3,0, 3.1, 3.2, 4.0, and 4.1, respectively, we investigate the differences between 3.1 and 3.0, between 3.2 and 3.1, between 4.0 and 3.2 and between 4.1 and 4.0. We do not compare version 4.0 and 3.0 with each other. This is done because we assume that library developers typically do not update from major release to major release but rather from previous release to next release.

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.

4.2. Determining the impact of breaking changes

To detect the actual impact of breaking changes on client libraries using them, we inject breaking changes in the source code of a software library, link code of client libraries, and compile the code. Fig. 1 shows an example of a library update and its impact.

A library class is shown, `Lib1`, and a system class that uses it, `System1`. Two changes have been introduced in version 2 of `Lib1`: method `foo` added a parameter `bar` and method `doStuff` changed its return type from `int` to `String`. If we upgrade the dependency of `Lib1` from version 1 to version 2 in `System1`, this causes two errors: Calling `c1.foo()` now gives a compilation error since it expects an integer as parameter, and `c1.doStuff()` returns a `String` instead of an `int`, which also gives a compilation error.

The two changes to `Lib1` are both breaking, and require adaptation and recompilation of a client using the changed functionality. We investigate both libraries as released by developers as well as other libraries using these releases in the same repository. To distinguish between these two, we refer to any library that includes another library as (system) S_x, and we refer to the included library as L_y. Although we denote a next version of L with L_{y+1}, this does not mean that L_{y+1} has to be an immediate successor version of L_y. Any version of L which has an release date after L_y is included in the set of next versions of L_y.

To determine the impact of breaking changes (binary incompatibilities), we follow the general process as outlined in Fig. 2. First, source code of a client system (S_x) is scanned and compiled with source code of a single dependency L_y of S_x (denoted with ①).

Next, all breaking changes between L_y and its next version L_{y+1} are calculated, as well as the edit script (see Section 4.4) to convert the first version into the second ($\Delta L_{y,y+1}$, denoted with ②). Third, each breaking change is inserted individually in L_y. Errors appearing in S_x after inserting these changes are then stored. The edit script size and breaking changes in $\Delta L_{y,y+1}$ are combined to estimate the number of changed statements per breaking change (denoted with ③).

Furthermore, S_{x+1} denotes a next version of S_x, which could have updated L_y to L_{y+1}. Any breaking change in $\Delta L_{y,y+1}$ would lead to work in the update from S_x to S_{x+1}, if the changed code is actually used in S_x. The amount of work done in $\Delta L_{y,y+1}$ for clients with and without breaking changes in dependencies (denoted with ④) is analyzed as part of RQ1.

The procedure to inject library changes is formally described in Algorithm 1 and can be explained in more detail as follows. For each library L (e.g. "JUnit"), all versions are collected (line 3). For each of these versions, a list of all libraries using L_y is obtained ($usingL_y$, line 5). For each library version L_y (e.g., "Junit 3.8.1") in the repository, a list of all future versions is created (line 6). For each pair of current and next version $U\langle L_y, L_{y+1}\rangle$ (the transitive closure over all next versions of L_y), all public API changes are determined ($\Delta L_{y,y+1}$, line 10). Each change $C \in \Delta L_{y,y+1}$ is inserted into L_y and the compilation errors are collected in all systems S_x that use L_y (lines 11–22). First, all files in S_x and L_y are compiled and linked together (S_x–L_y, line 13). Then, pre-existing errors in S_x–L_y are stored in *errStart* (line 14).

A single change is then injected in the code of S_x–L_y (line 15). Code is recompiled with the inserted change (line 16). Errors are again collected in *errEnd* (line 17), and pre-existing errors are removed from *errEnd* (line 18). The remaining errors are stored for this combination of a change, system, library and library update

[13] In this paper, an included library in a client system is called a *dependency*.
[14] Obtained from http://juliusdavies.ca/2013/j.emse/bertillonage/maven.tar.gz based on Davies et al. (2011, 2013).
[15] http://clirr.sourceforge.net.

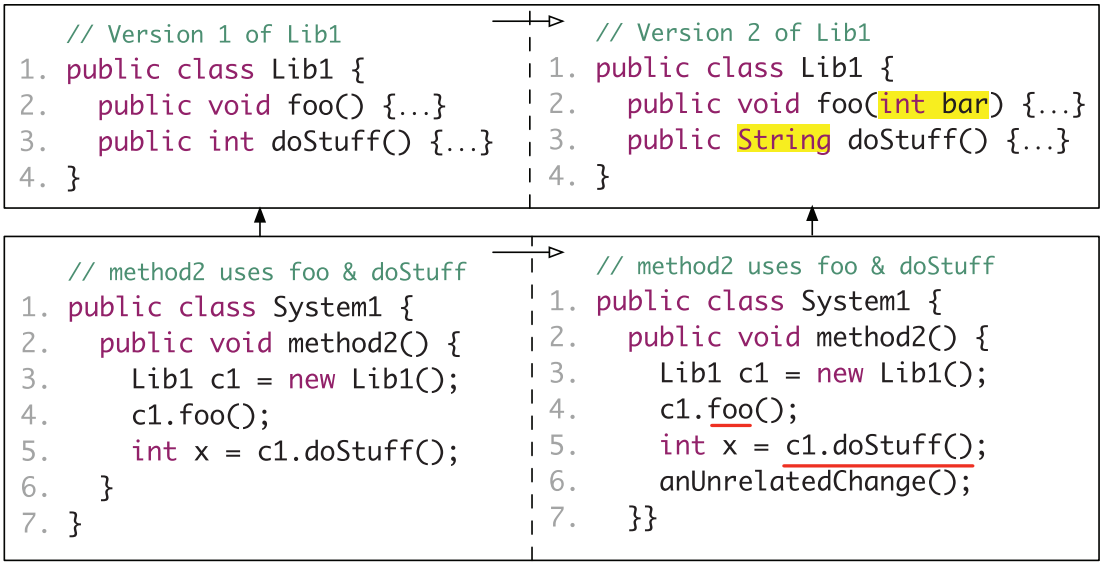

Fig. 1. Example of a library update and impact on a system. `Lib1` contains two changes, method `foo` with a new parameter `int bar`, and method `doStuff` with a return type of `String` instead of `int`. The compilation errors as a Java compiler would detect them are underlined in red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

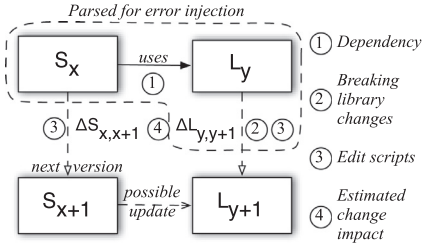

Fig. 2. Conceptual overview of our breaking change impact determination approach.

Algorithm 1: Change injection algorithm.

```
errStored ← ∅
for each library L do
  allVersions ← all versions of L
  for each version Ly ∈ allVersions do
    usingLy ← all source jars Sx using Ly ∈ repository
    possibleUpdates ← all possible updates
    {U⟨Ly, Ly+1⟩|Ly+1 ∈allVersions,
    Ly+1 newer than Ly}
    for each update U⟨Ly, Ly+1⟩ ∈ possibleUpdates do
      ΔLy,y+1 ← all changes between Ly and Ly+1
      for each Sx ∈ usingLy do
        for each change C ∈ ΔLy,y+1 do
          Compile code of Sx–Ly
          errStart ← collect compile errors in Sx–Ly
          Inject C in code of Ly
          Recompile code of Sx–Ly with C injected
          errEnd ← collect compile errors in Sx–Ly
          errors(Sx,Ly,Ly+1,C) ← errEnd − errStart
          errStored ← errStored ∪ errors(Sx,Ly,Ly+1,C)
          Revert C in code of Ly
        end for
      end for
    end for
  end for
end for
```

(line 19), and can later be grouped by change types, versions and libraries. Afterwards, the change is reverted (line 20).

From the build scripts (`pom.xml`) of each jar file, dependencies on other jar files were extracted. Source code in each source jar was automatically extracted and was compiled with the Eclipse JDT Core API,[16] which is the compiler of the Eclipse IDE. The Maven build system itself was used to obtain a list of other libraries that S_x and L_y need to compile successfully. The binary class files for each of these dependencies where added to the classpath of the compiler. Visitors for classes, methods and parameters were used to obtain data. The entire repository was processed on the DAS-3 Supercomputer[17] using 100 nodes in parallel in approximately 20 days, for an aggregate running time of 5.5 years.

In this paper, we perform several analyses on the same dataset but with a different number of observations. This is due to different selection criteria and exclusion of observations because of missing data, which depends on the specific analysis performed.

4.3. 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 "`junit`", "`junit`" and "`4.8.1`". 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[18] 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. The result is that each pair of subsequent versions is marked as either a major, a minor or a patch update.

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)

[16] http://www.eclipse.org/jdt/core.
[17] http://www.cs.vu.nl/das3.
[18] http://maven.apache.org/ref/3.1.1/maven-artifact.

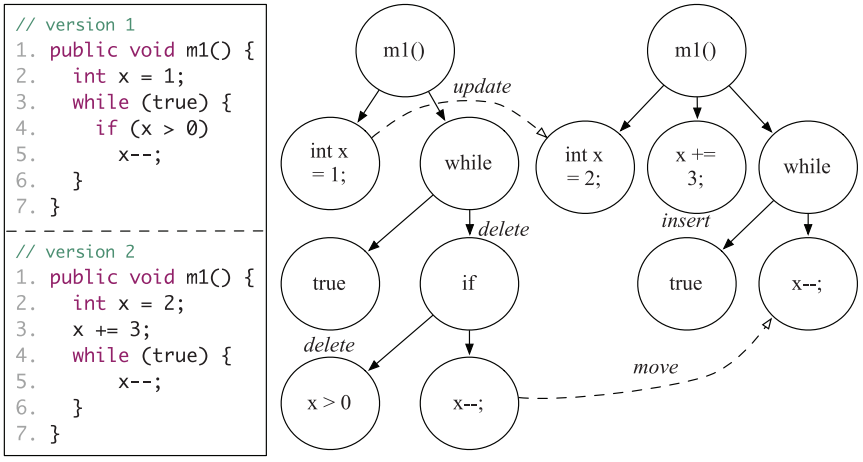

Fig. 3. An example of the calculation of an edit script between two version of a method. The resulting edit script has size of 5: one update, two delete, one insert and one move operation.

`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.

4.4. Detecting changed functionality and edit script size

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[19] to calculate edit scripts between library versions. For more information on ChangeDistiller, we refer to Fluri et al. (2007).

Fig. 3 shows an example of two pieces of code and the steps as determined by ChangeDistiller to convert the first version of the method into the second one. ChangeDistiller detects that the statement `int x = 1;` (line 2) is updated with a new value of 2. Also, it detects that the `if`-statement on line 4 of version 1 is deleted, and the statement `x--` (line 5) is moved. Altogether, the size of the edit script to convert the first version into the second is 5: one update, two delete, one insert and one move operation.

We use edit script as representation of changed functionality for the following reasons:

1. It closely resembles the actual work developers have performed between two releases.
2. It is not sensitive to changes in layout, whitespace, and comments.
3. It can be obtained automatically, which is a requirement given the large size of the repository.

To assess the amount of work that a library developer performs when breaking changes are introduced, we calculate the size of the edit script to convert L_y into L_{y+1}. The size of the edit script represents the total number of statements that must be inserted, deleted, moved or updated to transform L_y into L_{y+1}. The size of the edit script cannot be directly translated into effort in terms of man-hours since two edit scripts of the same length can each take a different time to implement, but it can nonetheless serve as an indicator for this effort. The edit script size is used as follows. First, the number of different change types in each update $\Delta L_{y,y+1}$ is determined. Then, we calculate the edit script size to update L_y to L_{y+1}. From this data, we estimate the amount of work that is associated with a single breaking change with a regression model.

Algorithm 2: Procedure to obtain edit script size data.

```
1:  for all library L do
2:    allVersions ← all versions of L
3:    for all version L_y ∈ allVersions do
4:      possibleUpdates ← all possible updates
5:      {U⟨L_y, L_{y+1}⟩ | L_{y+1} ∈ allVersions,
6:      L_{y+1} newer than L_y}
7:      for all {U⟨L_y, L_{y+1}⟩ ∈ possibleUpdates do
8:        ess(L_y, L_{y+1}) ← calcEditScriptSize(L_y, L_{y+1})
9:        for all change type c ∈ changeTypes do
10:         nrChanges(c, L_y, L_{y+1}) ← |{c|c ∈ ΔL_{y,y+1}}|
11:       end for
12:     end for
13:   end for
14: end for
15:
16: function calcEditScriptSize(L_y,L_{y+1})
17:   editScriptSize_{L_{y,y+1}} ← 0
18:   for each java file ∈ L_y do
19:     f_{y+1} ← find match for f_y in L_{y+1}
20:     editScript_{f_{y,y+1}} ← calculate Δf_{y,y+1}
21:     editScriptSize_{L_{y,y+1}} += |editScript_{f_{y,y+1}}|
22:   end for
23:   return editScriptSize_{L_{y,y+1}}
24: end function
```

Algorithm 2 formally describes our approach to obtain edit script size data. The procedure to obtain all possible update pairs (lines 1–7) is similar to Algorithm 1. The algorithm calculates the edit script size and the number of breaking changes for all library updates.

To calculate the edit script size (lines 16–24), the following steps are taken. For each java file in L_y, the corresponding next version of the file is found in L_{y+1} (line 19). The edit script to convert f_y into f_{y+1} is calculated (line 20), and the size of this edit script is added to the total edit script size of $\langle L_y, L_{y+1}\rangle$ (line 21). This data serves as dependent variable in the regression model of rework estimation. Finally, the number of times the 10 different

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

update types occur in $\Delta L_{y,y+1}$ is calculated and stored (line 10). These numbers serve as the independent variables in our regression model.

ChangeDistiller was used to calculate edit scripts in $\Delta L_{y,y+1}$ (Fluri et al., 2007). Fig. 3 shows an example of two pieces of code and the steps as determined by ChangeDistiller to convert the first version of the method into the second one. ChangeDistiller detects that the statement `int x = 1;` (line 2) is updated with a new value of 2. Also, it detects that the `if`-statement on line 4 of version 1 is deleted, and the statement `x---` (line 5) is moved. Altogether, the size of the edit script to convert the first version into the second is 5: one update, two delete, one insert and one move operation.

ChangeDistiller works on the level of individual source files, but was adapted to work on the level of jar files. This can be seen in lines 22–27 of Algorithm 2. For each two versions of a java source file, ChangeDistiller calculates the edit script to convert the first version into the second. In our approach, we see each jar file as a collection of java files. Each java file in the jar file is iterated and the corresponding next version of that file is found in L_{y+1}. the length of the edit script to convert f_y into f_{y+1} is added to the total edit script size for the jar file. To match versions of files, filenames that matched directly are considered to be two versions of the same file (for instance, two files with a filename ending in `java/src/foo/bar/Bar.java` are considered direct matches). Files that did not have a direct counterpart in the other version, meaning they were deleted, added, or moved, were matched using a token-based similarity algorithm similar as used by ChangeDistiller itself. When two file pairs exceeded the default token-based similarity threshold of 0.8, these files where considered to be moved. Our adaptation of ChangeDistiller returns a single number that represents the length of the edit script to convert S_x into S_{x+1}. For each update in the Maven repository, this number is stored in our database.

4.5. Obtaining release intervals and dependencies

To calculate release intervals, we collect upload dates for each jar file in the Maven Central Repository. Upload dates were obtained for 129,183 out of 144,934 (89.1%) of libraries. A small number of libraries have the same date as release date (November 11th, 2005), which is suspected to be a default value, and these were left out of the analysis.

4.6. Obtaining deprecation patterns

For API developers, the Java language offers the possibility to warn about future incompatibilities by means of the "`@Deprecated`" annotation.[20] By marking old methods as deprecated, backward compatibility is retained while still providing library users with a signal to stop using that method. In `semver`, the use of such annotations is required, before methods are actually removed. To detect deprecation tags, we scan the source code for the text "`@Deprecated`". By building an abstract syntax tree by using the Java Development Tools Core library,[21] we match the deprecation tags to update types from Section 4.3 to make it possible to distinguish between different types of deprecation patterns.

In the next sections, we answer each of our research questions.

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

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

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

#	*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	3269	2150	Yes
4	MAJOR.MINOR-prerelease	2.0-beta1	16,115	10,756	No
5	MAJOR.MINOR.PATCH-pre.	2.0.1-beta1	12,674	8939	No
6	Other versioning scheme	2.0.1.5.4	10,930	8307	No
		Total	129,138	91,731	

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

Breaking changes		
#	*Change type*	*Frequency*
1	Method has been removed (MR)	177,480
2	Class has been removed (CR)	168,743
3	Field has been removed (FR)	126,334
4	Parameter type change (PTC)	69,335
5	Method return type change (MRC)	54,742
6	Interface has been removed (IR)	46,852
7	Number of arguments changed (NPC)	42,286
8	Method added to interface (MAI)	28,833
9	Field type change (FTC)	27,306
10	Field removed, previously constant (CFR)	12,979
11	Removed from the list of superclasses	9429
12	Field is now final	9351
13	Accessibility of method has been decreased	6520
14	Accessibility of field has been weakened	6381
15	Method is now final	5641
16	Abstract method has been added	2532
17	Added final modifier	1260
18	Field is now static	726
19	Added abstract modifier	564
20	Field is now non-static	509
Non-breaking changes		
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	9202

5. RQ1: application of semantic versioning

We first investigate different version string patterns that can be found in our repository. After this, we determine how many major, minor and patch releases actually occur in our dataset, and differences between these update types in terms of release cycle and average number of breaking changes.

5.1. 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 table shows that most libraries follow the version string pattern as prescribed by semantic versioning, which enables automated analysis of adherence to this standard as performed in this paper.

The first three versioning schemes correspond to actual `semver` releases, whereas the remaining ones correspond to

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

Update type	Contains at least 1 breaking change				*Total*
	Yes	*%*	*No*	*%*	
Major	4268	35.8%	7624	64.2%	11,892
Minor	10,690	35.7%	19,267	64.3%	29,957
Patch	9239	23.8%	29,501	76.2%	38,740
Total	24,197	30.0%	56,392	70.0%	80,589

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

Type	*#Breaking*		*#Non-break.*		*Edit script*		*Days*	
	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

prereleases. Since prereleases 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)[22] 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 (6442 out of 22,205 libraries, 29%).

This shows that most libraries follow a version string pattern that is compatible with semantic versioning guidelines, even though these guidelines may not have been followed intentionally.

5.2. Breaking and non-breaking changes

In total, 126,070 update pairs $\langle L_y, L_{y+1}\rangle$ have been extracted from the Maven repository. Out of all these potential updates, 48,143 pairs contain an L_y that is actually used by an S_x. Out of these 48,143 pairs, 3260 pairs actually contain breaking changes (6.8%).

Table 2 shows the top 20 breaking changes and top 10 non-breaking changes in the Maven repository as detected by Clirr. The breaking changes in these table are obtained from the 126,070 potential updates $\langle L_y, L_{y+1}\rangle$. 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. 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 best practice 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 best practices such as encoded in `semver` are not adhered to in practice with respect to breaking changes. The total number of updates in Table 3 (80,589) differs from the total number of pairs in Table 1 (91,731) because of missing or corrupt jar files, which have a correct version string but cannot be analyzed by Clirr.

We can thus conclude that breaking changes are common, even in non-major releases.

5.3. 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. Although there does seem to be some respect for semantic versioning principles in the sense that minor and patch releases introduce fewer breaking changes (around half as many as the major releases), 27 and 30 breaking changes 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.[23]

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.

Major releases are generally smaller in terms of work performed than minor releases, and are released faster than minor releases. Major releases contain less breaking changes on average than minor releases.

5.4. Median analysis

To find out how the number of days since the previous release relates to the update type of the release, we perform a quantile regression that shows the median number of days that an update in each category approximately takes. Since the data is highly skewed, we perform a bootstrap to resample from the skewed distributions, which provides normal distributions. To further prevent the influence of extreme outliers, we estimate the median number of days instead of the average number of days per group.

Table 5 shows the result of the analysis. Practically, the table shows us that major releases are released at a median number of days of 42. Minor releases are released at a median number of days

[22] 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).

[23] 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.

Table 5
ANOVA analysis to compare the number of breaking changes and the churn in major, minor and patch releases.

Release type	*Median coeff.*	*Bootstr. std. error*	*p-Value*	*95% C.I.*
Minor	10	1.319	0.000	7.416 to 12.584
Patch	−3	1.353	0.027	−5.652 to −0.348
Constant (major)	42	1.128	0.000	39.50 to 44.50

Table 6
The types of changes detected. Frequency = the number of times this change type occurred in an update, *#Errors* = The number of errors this update type caused in all S_x, *#E/F* = the average number of errors per breaking change, *#sys* = The number of distinct S_x that contain errors because of this update, *#uniq* = The number of different updates of L_y that contain this change.

#	*Type*	*Frequency*	*#Errors*	*#E/F*	*#sys*	*#uniq*
1	MR	177,480	1,524,498	8.59	8328	960
2	CR	168,743	1,645,518	9.75	3983	505
3	FR	126,334	4,143,723	32.80	8028	960
4	PTC	69,335	956,314	13.79	5357	547
5	RTC	54,742	288,939	5.28	4478	433
6	IR	46,852	95,250	2.03	1657	130
7	NPC	42,286	533,741	12.62	5701	713
8	MAI	28,833	126,427	4.38	4746	562
9	FTC	27,306	1,233,095	45.16	4324	485
10	CFR	12,979	677,234	52.18	3354	317
	Total	595,158	11,139,014	18.72		

of 42 + 10 = 52, and patch releases take a median of 42 − 3 = 39 days to be released.

This shows that minor releases tend to take longer to be released than major releases. An ANOVA analysis based on averages (n = 58,763, F = 0) gives 79 days for major, 84 days for minor and 61 days for patch releases, also showing that minor releases tend to take longer on average to be released than major releases. A possible explanation is that a major release contains less rework that takes a large development effort but instead mainly contains changes to the interface instead of rework effort in the entire library, which would take more time. An alternative explanation is that development on major releases started on a separate branch earlier than the update dates in our data shows.

To answer **RQ1**: *The version string conventions as prescribed by semantic versioning are generally followed in the Maven repository. However, breaking changes are widespread, even in non-major releases. Surprisingly, on average minor releases contain more changes and take longer to release than major releases.*

6. RQ2: breaking changes and errors

To answer **RQ2**: "What is the impact of breaking changes in terms of compilation errors?", we investigate the number of breaking changes and the relationship with compilation errors in this section.

Table 6 shows overview statistics for the 10 different types of breaking changes detected by applying Algorithm 1 to the entire Maven repository.

The table shows the number of breaking changes and the number of compilation errors these changes cause. For instance, class removals occur 168,743 times and cause a total of 1,645,518 compilation errors when applying the algorithm to the entire repository. The most frequently occurring breaking change is the method removal, occurring 177,480 times in the repository and causing 1,645,518 compilation errors in total. For method removals, there are 3,983 unique jar files that contain compilation errors caused by breaking changes in 505 unique jar files. Another type of frequently occurring breaking change is the class removal, which appears 126,334 times in our dataset and causes 1,645,518 errors.

Table 7
Regression analysis to estimate the relationship between breaking changes and errors.

Dependent variable	ln(*NE*)			
Number of observations	2269			
R^2	0.8879			
Model *p*-Value	0.0000			
Independent	*Coeff.*	*Std. err*	*p*	*95% C.I.*
ln(*NBC*)	1.683	0.133	0	1.657–1.709

The average number of errors per breaking change is also shown in Table 6. It shows that a constant field removal (*CFR*) has the highest average number of errors per change: 52.18. Furthermore, field type changes (45.16), field removals (32.8) and parameter type changes (13.79) cause a relatively large number of compilation errors as compared to other change types. On average, a breaking change causes 18.72 errors.

Applying all possible library updates and collecting all compilation errors gives a total of 595,158 breaking changes of the 10 most occurring change types and a total of 11,139,014 compilation errors because of these changes. This thus demonstrates that breaking changes are a real problem in the Maven repository, since they cause a large number of compilation errors which would need to be fixed before a newer version of a library can be used.

6.1. The relationship between breaking changes and errors

To further investigate the relationship between breaking changes and the number of errors caused by these changes, we calculate the correlation between these properties. The Spearman rank correlation between the number of breaking changes in $\Delta L_{y,y+1}$ and the number of errors in S_x caused by these changes is 0.65 (p = 0), indicating a significant positive relationship between breaking changes and compilation errors caused by these changes, as expected.

To investigate further how many errors each breaking change introduces, we perform the following regression analysis:

$$\ln(NE)_i = \beta_1 \ln(NBC)_i + \varepsilon_i$$

with *NE* being the number of errors in S_x and *NBC* being the number of breaking changes in $\Delta L_{y,y+1}$. We do not estimate a constant since each error must be caused by a breaking change. Both *NE* and *NBC* are log-transformed because the data is lognormally distributed. The results can be found in Table 7. The model is highly significant with a p-Value of 0 and an adjusted R^2 of 88.79%. The estimated slope coefficient of NBC is 1.683, indicating that if the number of breaking changes increases by 1%, the number of errors is expected to increase by 1.683%.

Table 8
ANOVA analysis to compare the average edit script size in library updates in the entire Maven repository and library updates with breaking changes in dependencies.

Dataset	μ	σ	*Freq.*
2106 systems with breaking changes	0.657	4.055	2106
Entire Maven repository	0.376	3.500	24,565
	SS	*df*	*MS*
Between groups	40.99	1	40.99
Within groups	4154.3	26,669	0.156
Total	4195.30	26,670	0.157

We can thus conclude that breaking changes cause a significant amount of compilation errors in client systems.

6.2. Average amount of work with and without breaking changes

To further investigate the relationship between edit script size and breaking changes in libraries, we calculate the mean edit script size per method for library updates with and without breaking changes. We use the 3260 library updates which contain breaking changes as described in Section 4, but due to missing data, only 2106 systems can be used in this analysis. We denote the average edit script size in this set as (μ_{bc}), which we compare to the average edit script size in the entire Maven repository regardless of breaking changes, denoted as (μ_{maven}). The edit script size is divided by the number of methods in L_{y+1} to correct for the effect of library size. We compare these means to find out if the amount of work in library updates with breaking changes is comparable to the amount of work performed in general.

There are three possibilities:

1. $\mu_{bc} < \mu_{maven}$: A library update containing breaking changes contains less work as compared the work done in the average library release. This may be caused by the fact that fixing breaking changes requires rework in the library itself, as shown in Table 12, which may interfere with other work performed in that update.
2. $\mu_{bc} \approx \mu_{maven}$: The average amount of work done in library updates which include breaking changes is not significantly different from work done in releases in general.
3. $\mu_{bc} > \mu_{maven}$: A developer performs more work in a library update that contains breaking changes than in library releases in general: breaking changes are more frequently introduced in bigger updates.

To compare the means between these two groups, we perform an ANOVA analysis, of which the results are shown in Table 8.

The analysis is significant with $F = 12.16$ and a p-Value of 0, indicating that there exists a significant difference in the amount of work performed in library updates with breaking changes and library updates in general. The analysis contains 24,565 libraries from the Maven repository and 2106 libraries from the rework estimation analysis we performed in Section 8.1. The mean edit script size *per method* of the Maven repository group is 0.376 and the mean for the 2106 systems is 0.657. This means that for two systems with 100 methods, the edit script size for a system with breaking changes in library updates will be 65.7 and the edit script size for a library update in general will be 37.6, which is a difference of approximately 75%. The ANOVA analysis indicates that there exists statistical support for the third scenario, $\mu_{bc} > \mu_{maven}$, which means the average edit script size per method tends to be larger for library updates with breaking changes than for library updates in general. This means that breaking changes occur in library updates where a relatively large amount of code is changed. This could indicate that developers pay less attention to backward compatibility when they work on a large library update.

To answer **RQ2**: *Breaking changes have a significant impact in terms of compilation errors in client systems.*

7. RQ3: semantic versioning adherence over time

In this section, we answer **RQ3**: "Has the adherence to semantic versioning principles increased over time?" To find this out, 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 Fig. 4.

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 **RQ3**: *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.*

8. RQ4: update behavior

In this section, we answer **RQ4**: "How are dependencies actually updated in practice, what are typical properties of new library releases, and do these properties influence the speed with which dependencies get updated?".

The key reason to investigate breaking changes is that they complicate upgrading a library to its latest version. To 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 9 shows the number of updates of different types of S and L in the Maven repository. When a system S is updated, a library dependency L can be updated as well to a major, a minor, or a patch version. When looking at each horizontal row in the table, it 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).

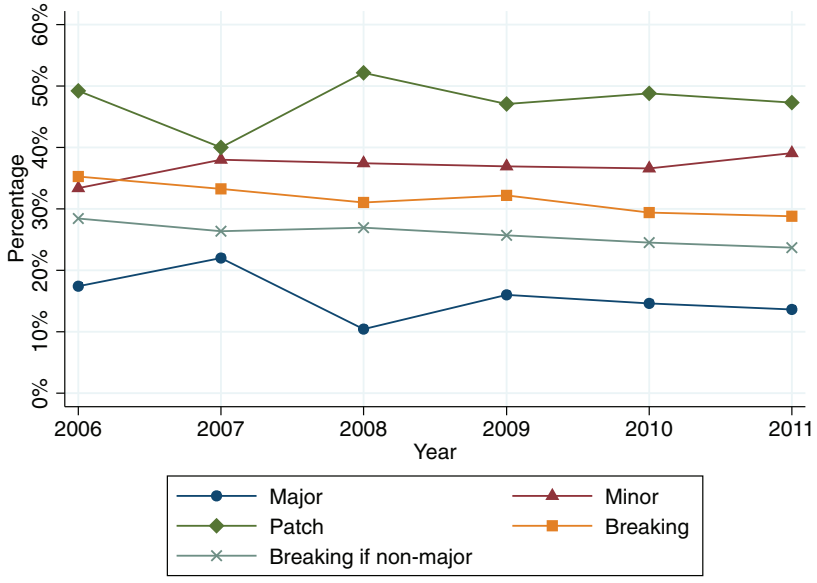

Fig. 4. The percentage of major, minor, patch, breaking, and breaking if non-major releases through time.

Table 9
The number of updates of different types of S and simultaneous updates of dependency *L*.

Update S	*Update L*			*Total*
	Major	*Minor*	*Patch*	
Major	543	189	82	814
Minor	651	791	227	1669
Patch	150	54	297	501
Total	1344	1034	606	2984

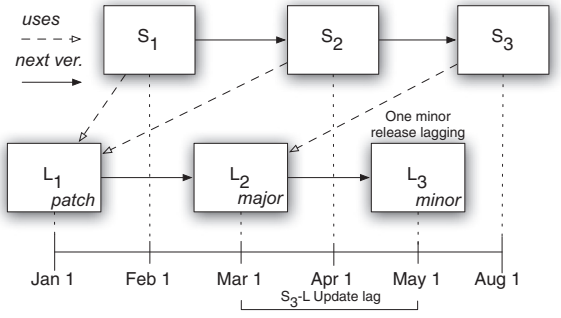

Fig. 5. An example of a timeline with a system *S* updating library *L*.

To further investigate update behavior of dependencies, we calculate the number of versions of *L* that *S* lags behind, as illustrated in Fig. 5. 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.

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.

Table 10
Percentiles for the number of major, minor and patch dependency versions lagging.

	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

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 10 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

Table 11
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.

	Breaking changes	Edit script size	Changes
Major versions lagging	0.0772	−0.0701	−0.0465
Minor versions lagging	0.1440	0.1272	−0.0434
Patch versions lagging	0.0190	0.0199	0.3824

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 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 11. 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 numbers indicate that, in general, people are more reluctant to update major, minor and patch releases with a larger number of breaking changes, but the effects are very small. Alternatively, one could argue that people tend to ignore breaking changes and changed functionality in new versions of dependencies, perhaps because they do not even *know* a priori whether a release introduces breaking changes. Thus, there exists a lag in these dependencies, regardless of breaking changes or changed functionality.

The correlation between the edit script size and the number of major versions lagging is even negative with a value of −0.0701, which indicates that major library versions with a larger amount of new or changed functionality are generally included slightly faster than releases with less changed or new functionality. The correlation between the number of breaking changes and the edit script size and the number of patch versions lagging is negligible with values of 0.0190 and 0.0199, with significant p-Values.

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.

8.1. Breaking changes and edit script size

To further investigate update behavior on the library side, we perform a regression analysis, linking the edit script size of an update to different types of breaking changes. This analysis shows what amount of work is typically performed in a new release of a library and what edit script size is associated with different breaking changes.

From the data acquired through Algorithm 2, we estimate the influence of each breaking change type in $\Delta L_{y,y+1}$ by including the number of occurrences of each type as independent variables. The dependent variable is the size of the edit script of $\Delta L_{y,y+1}$. Table 12 shows the results of this regression, which is based on the 3260 pairs containing breaking changes as described in Section 4. The actual number of observations is only 2447 due to the exclusion of observations with missing data.

As can be seen in Table 12, the model as a whole is highly significant ($p = 0$) and has an adjusted R^2 of 58.68%, indicating that more than 58% of the variability in the edit script size between L_y and L_{y+1} is explained by the 10 different change types in the model. The model shows that all variables are significant at the 95% confidence interval, indicating that the all variables contribute significantly to the total edit script size in $\Delta L_{y,y+1}$. The coefficients in the model indicate the size of the performed rework in terms of tree edit operations to update a library from L_y to L_{y+1}. For instance, the change type method removal (*MR*) has a coefficient of 2.415, indicating that a method removal in $\Delta L_{y,y+1}$ takes 2.415 edit script operations, on average.

As the table shows, all 10 breaking change types are associated with a significant edit script size, but some changes have a larger coefficient than others. For instance, a class removal and an interface removal only represent an edit script size of 0.539 and 0.684, respectively. This could be explained through the average size of classes or interfaces that are removed, which could be smaller than the average class. The constant of 5.0 indicates that the average library update which contains breaking changes has a "base level" average of 5 edit script lines.

As an example of the expected edit script size in a library update, consider a library which removes a class with 10 methods and two private fields in its next version. The predicted edit script size would then be $5.001 + 1 * 0.539 + 5 * 2.415 + 2 * 0.818 = 19.251$. The constant of 5 indicates that a library change without any of the included change types takes an edit script size of 5, on average.

Comparing the standardized coefficients (*beta*) for each of the 10 change types, it can be seen that the method removal (*MR*) and the parameter type change (*PTC*) have the largest influence on the total edit script size, with a *beta* of 0.346 and 0.208, respectively. Field removals, class removals and field type changes turn out to have relatively little influence on the total edit script size, with *beta*'s of 0.059, 0.069, and 0.049, respectively. The constant field removal *CFR* correlates too much with other change types and is therefore excluded automatically from the regression.

To answer **RQ4**: *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. Method removals and parameter type changes are two changes which are typically associated with the largest changes in library code.*

9. RQ5: library characteristics associated with large impact

In this section, we answer **RQ5**: "Which library characteristics are shared by libraries which frequently introduce a large number of breaking changes, and as a result, cause compilation errors?"

To assess which library characteristics cause a large number of compilation errors in dependent systems, we investigate the correlation of breaking changes and errors with two library properties: the maturity and the size of a library.

We use the index of a release (any release, major, minor or patch) as a proxy for the maturity of a library, starting with 1 from the oldest release. We assume that the more releases a certain library had before the current release, the more mature it is. Alternative measures, such as the number of days since the first release, were considered inferior since a library can have a single release and another release 2 years later, which would indicate a mature library. The size of a library is measured as the number of methods in a library.

These properties are investigated for the following reason. We expect that the size of a library increases as the library matures. For this reason, the number of methods and the release index are

Table 12
Regression analysis on the edit script size and different change types in libraries.

Dependent variable	$ess(L_{y,y+1})$					
Number of observations	2447					
R^2	58.83%					
Adjusted R^2	58.68%					
Model p-Value	0					
Indep.	*#*	*Coeff.*	*Std. err*	*beta*	*p*	*95% C.I.*
Constant	0	5.001	1.096	–	0	2.851–7.151
MR	1	2.415	0.110	0.346	0	2.200–2.630
CR	2	0.539	0.109	0.069	0	0.325–0.753
FR	3	0.818	0.187	0.059	0	0.451–1.184
PTC	4	1.921	0.141	0.208	0	1.646–2.197
RTC	5	2.021	0.221	0.141	0	1.587–2.454
IR	6	0.684	0.218	0.043	0	0.256–1.113
NPC	7	2.734	0.191	0.204	0	2.360–3.108
MAI	8	2.534	0.193	0.178	0	2.156–2.913
FTC	9	1.239	0.367	0.049	0	0.518–1.960
CFR	10	*Omitted due to collinearity*				

Table 13
Spearman rank correlations between the number of breaking changes, number of errors, number of methods, and the release index of a library.

		0.65 ($p = 0$)	
		#Breaking changes	#Errors
0.0278 ($p = 0$)	# of methods	0.3291 ($p = 0$)	0.3392 ($p = 0$)
	Release index	−0.015 ($p = 0.153$)	0.1078 ($p = 0$)

expected to be positively correlated. We also expect that it becomes increasingly hard for library developers to maintain backward compatibility as the maturity of a library increases, simply because the library has a larger interface that can be broken. Therefore, the correlation between the maturity and the number of breaking changes in a release is expected to be positive as well. The number of compilation errors is expected to have comparable correlations with these two properties, because of the direct relationship between breaking changes and the number of errors caused by these changes.

Spearman rank correlation coefficients of these properties can be found in Table 13. There is a correlation of 0.3291 between the number of methods in a library and the number of breaking changes in that library, meaning that bigger libraries indeed tend to introduce more breaking changes.

The correlation between the release index and the number of methods in that library turns out to be only marginally positive with a value of 0.0278, meaning that there is practically no correlation between these two properties: most libraries do not seem to grow in the number of methods through time. This is contrary to our expectation. There is a negligible correlation between the number of breaking changes and the release index, indicating that libraries do not introduce more breaking changes when they become more mature, which is also contrary to our expectation. The correlation between the number of errors and the number of methods is also positive, indicating that larger libraries cause more compilation errors.

To answer **RQ5**: *Bigger libraries tend to introduce more breaking changes and errors. Libraries do not grow when they become more mature, on average, and more mature libraries do not introduce more breaking changes.*

10. RQ6: deprecation patterns

In this section, we answer **RQ6**: "How are deprecation tags applied to methods in the Maven repository?"

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."*[24] 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.

Table 14 and 15 show different correct and incorrect deprecation patterns. The columns with headers *v1* to *v4* contain possible deprecation patterns in a subsequent major, minor, minor and major release, respectively. 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 14 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

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

Table 14
Correct method deprecation patterns. @d = deprecated tag, *c* = correct, *i* = interesting; pr = private; pu = public; – = method deleted.

Correct patterns								
#	*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 @d m5	pu @d m5	–	y	y	0	0

Table 15
Incorrect method deprecation patterns. @d = deprecated tag, *c* = correct, *i* = interesting; pr = private; pu = public; – = method deleted.

Incorrect patterns								
#	*v1 (maj.)*	*v2 (min.)*	*v3 (min.)*	*v4 (maj.)*	*c*	*i*	*Freq.*	*%*
6	pu m6	pu m6	–	–	n	y	86,449	33.03
7	pu m7	pu @d m7	–	–	n	y	0	0
8	pu m8	pu m8	pu m8	pu @d m8	n	y	0	0
9	pu m9	pu @d m9	pu m9	pu m9	n	y	16	0.01

methods. Case 5 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.

Table 15 shows examples of incorrect deprecation patterns. Case 6 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 7 deprecates the method, but deletes it in a *minor* release, which would not be correct. This case does not occur. Case 8 declares the method deprecated in a major release, which would also be incorrect (and which does not occur). 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 they are supposed to be used. 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 proper way to deprecate and delete methods. However, the pattern was not found in the entire Maven repository.

To answer **RQ6**: *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. Deprecation tags are* ***never*** *applied correctly in the Maven repository as described by semantic versioning.*

11. RQ7: dispersion estimation

In this section, we answer **RQ7**: "What is the impact of breaking changes in terms of the spread of errors caused by these changes?"

11.1. Explanation

When fixing compilation errors caused by a library update, not only the number of errors is relevant in the estimation of total rework, but the dispersion of these errors across different places in the code is relevant as well. We expect that a change that causes 10 errors inside a single file is easier to fix than a change that causes 10 errors in 10 different files, since the code and context of each file has to be understood separately before its errors can be fixed. Fixing errors in multiple different files is therefore expected to take more time than fixing errors inside a single file.

For instance, a software library that is typically used in a limited set of places in code, such as JUnit or a database library, is expected to cause less work to change since code is localized in a small set of places. In contrast, a change in a base class library of Java itself, such as in the `String` class, is expected to have a large impact because its usage is so pervasive: there is a large chance that a class uses an instance of a `String`.

Fig. 6 shows the concept underlying this analysis. As can be seen in this figure, L_{y+1} is used by three different libraries, S_{x1}, S_{x2} and S_{x3}. Library S_{x1} calls a method that was changed in $\Delta L_{y,y+1}$, and as a result, 2 compilation errors in one distinct method are introduced. Library S_{x2} calls a method that was added, but this does not introduce any errors.

Library S_{x3} calls a method that was removed, and therefore, 3 errors in 2 distinct methods are introduced. One error in S_{x3} is not directly related to the call to L_{y+1} but is a cascading error, caused by the other method that directly calls L_{y+1}. Overall, there are 3 distinct methods that are impacted because of the update of L_y to L_{y+1}.

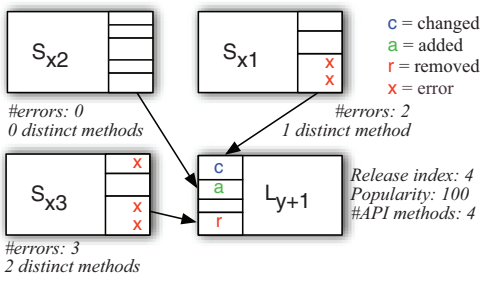

Fig. 6. Example of a library L_{y+1} and three libraries (S_{x1}, S_{x2}, S_{x3}) that use it. The update from L_y to L_{y+1} contains 3 changes to interface methods: a method header was changed (c), a method was removed (r), and a method was added (a).

We perform a regression analysis to test factors influencing the dispersion of errors in different libraries S_x with changes in a dependency L_y. As a measure of the dispersion of compilation errors across systems, we count the number of distinct files of all S_x that use L_y and which contain one or more compilation error after the update to L_{y+1}.

11.2. Explanation of independent variables

We include 3 independent variables in our analysis, which are expected to explain a part of the variability in error dispersion across systems. These factors are the release index of L_y, the number of methods in L_y and the relative usage frequency of L_y in the Maven repository. The rationale for inclusion of these independent variables is as follows. The number of methods in L_y is a measure of the size of that library. We expect that a larger size of L_y causes methods of this library to be used at more separate places in S_x, because it is expected to contain more separate pieces of functionality.

We include the release index as independent variable to measure maturity for the same reason it was included in the analysis in Section 10: we expect that libraries tend to increase the amount of functionality they provide over time as the library matures, which would lead to a larger scattering of errors across S_x. With the inclusion of both the release index and library size as independent variables, we can test for the influence of both independent variables separately, while keeping the others constant.

We include the popularity as independent variable to correct for usage frequency of libraries, which is defined as the number of libraries that use L_y divided by the total number of libraries in the repository (144,934). When a library L_y is used more frequently, it will cause more errors in systems using it simply because it is used more. This independent variable corrects for that effect.

Note that these three variables are all properties of L_{y+1}, and not of S_x, although the errors appear in S_x. We include properties of L_{y+1} in this analysis since we expect that the error dispersion in S_x is, to a degree, a property of the library causing the errors. Since error dispersion is also expected to be influenced by properties of S_x itself, we do not expect that the included properties are fully able to explain all variability in the model, which would lead to a relatively low R^2.

We include the natural logarithm of the number of distinct files and the number of methods in L_y because data analysis shows that the natural logarithm of these variables has a linear relationship.

11.3. Regression results

Table 16 shows the result of this analysis.

The model has an adjusted R^2 of 0.1333, which is significant with a p-Value of 0.[25] The release index and the natural logarithm of the number of methods in L_y are significant but the popularity of L_y is not. The *beta* coefficients show that the size of L_y is the most important factor in explaining the dispersion of errors across S_x, with a beta of 0.381. The release index of a library has a coefficient that is significant but practically not significantly different from 0.

This means that, after correcting for library popularity and maturity, the number of methods in a library is a significant predictor for the dispersion of errors in client systems. This could be explained by our theory that larger libraries contain more separate pieces of functionality, which have a bigger chance of ending up at different places in client systems.

To answer **RQ7**: *The size of a library tends to increase the dispersion of errors in client systems.*

12. 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.

12.1. Signaling interface instability

Our results show that developers do not tend to follow the best practices encoded in `semver`, even though the used versioning schemes suggest a semantic pattern. If developers would adhere completely to `semver` 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. Possible explanations are that library developers are not aware of existing semantic versioning practices, they are not aware that they have introduced breaking changes, they do not expect that the changes they make have actual impact on client systems, or they simply do not care. Either way, we argue that the principles set out by `semver` should be followed by every developer of software libraries, or any piece of software of which the interface is used by other developers.

In our opinion, 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

[25] For an explanation of this low R^2, see Section 13.7.

Table 16
Regression analysis to analyze factors associated with a large error dispersion.

Dependent variable	ln(#distinct files with error in S_x)				
Number of observations	3690				
R^2	0.1354				
Adjusted R^2	0.1333				
Model p-Value	0.0000				
Independents	*Coeff.*	*beta*	*Std. err*	*p-Value*	*95% C.I.*
Release index (L_y)	−0.007	−0.075	0.003	0.006	−0.013 to −0.002
ln(nr. methods (L_y))	0.313	0.381	0.022	0	0.269 to 0.357
Popularity (L_y)	−47.68	−0.019	67.26	0.478	−179.6 to 84.27
Constant	−0.262	–	0.131	0.045	−0.519 to −0.006

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.

The concept of an interface can be seen as a set of mathematical functions, which are by definition immutable. Daily software engineering practice, however, is different, since changes to these functions are common. By using semantic versioning principles, interface consumers are signaled that the normal expectation that they might have regarding the stability of interface functions does not hold in a particular case.

Another explanation for the lack of discipline in interface versioning 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.

12.2. Other versioning standards

Semantic versioning is not the only standard for versioning software libraries. For instance, the OSGi alliance has released their own semantic versioning manifesto[26] and contains comparable guidelines as the ones in `semver`. Furthermore, there exist several alternative versioning approaches,[27] 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.

[26] http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf.
[27] http://en.wikipedia.org/wiki/Software_versioning.

12.3. Actual usage frequencies

In our research, we do not take into account the difference between internal and non-internal packages. The number of internal packages as compared to the number of non-internal packages is negligible, and are therefore not expected to influence our analyses in a material way.

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, semantic versioning principles generally do 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. The same is true for breaking changes in internal and non-internal packages.

Future work could investigate the difference in the occurrence of breaking changes in functionality that is actually used by other developers and breaking changes in internal packages. Also, the adherence to `semver` in libraries that use the OSGi framework could be investigated. We expect that the adherence to `semver` is higher in packages that use OSGi since OSGi provides an additional layer of visibility which would prevent counting breaking changes in internal packages.

12.4. 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.

12.5. 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.

12.6. Edit scripts as rework measure

The edit script size to convert a library into one of its next versions was chosen to represent the performed rework in that update. It was preferred over other measurements of rework, such as the difference between the number of lines of code (LOC) between two library versions, because we consider it to be the most detailed representation of changes available. Differences in LOC have the problem that when the contents of a method or file completely changes but the LOC stays exactly the same, no difference is detected. Differencing as used by version control systems was also considered to be inaccurate for the purpose of this analysis since it is sensitive to irrelevant changes in whitespace and comments.

12.7. Other applications of change impact analysis

The approach to inject breaking changes as described in this paper can be used to perform general change impact analysis as well, without breaking changes or libraries. For instance, the Eclipse compiler could also be used in a similar way to perform a "mutation analysis" of public interfaces, where random changes are injected in the public API to determine how systems using that API would react. This is similar to the work performed in this paper, but would not be limited to breaking changes that actually occur. The data as obtained by our approach could also be used to construct a "profile" for a library, which would indicate the expected amount and spread of errors when updating a certain library. We expect that different libraries will have different profiles and some libraries are easier to update than others, partly due to design and partly due to the problem the library solves.

13. Threats to validity

13.1. Transitive closure of update pairs

A potential issue of internal validity of our approach is the calculation of the transitive closure of all future versions of each library. This was done for three reasons. First, in practice, developers can update to any later version of a library from their current version and can skip versions "in between". Second, the number of breaking changes between, for example, version 1 and version 3 of a library is not the sum of the breaking changes between version pairs 1–2 and 2–3. This also gives rise to the need to consider each update pair separately. Third, the regression analyses as performed in this paper are not influenced by the larger amount of data since, statistically, estimated coefficients will not be influenced by the possible duplication in the data. We use robust regression and robust standard errors to mitigate any risk rising from data duplication, which are common statistical methods to deal with this type of problem (Andersen, 2008).

13.2. Error counting

Another issue of internal validity is the "masking" of compilation errors, which happens when compiling Java code with Eclipse and the JDT. If a package import cannot be found, for instance, the compiler will never reach compilation errors further down the file because it stops compiling. It is unknown how many times this happens in our dataset. It would mean, however, that the true amount of rework to fix "masking" compilation errors would be underestimated using our approach since fixing these errors would reveal new, previously unreported compilation errors which would have to be fixed in turn. On the other hand, collections of compilation errors can be manifestations of a change in the same object, such as a removed method, field or parameter. Fixing such errors would be faster than fixing the same amount of unrelated errors, for instance with a global search-and-replace action. More research is needed to assess the strength of over- or underestimation of the rework effect due to these two reasons.

13.3. Release dates

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 (2321, 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.

13.4. Version strings

We only investigated the changes in subsequent library versions which both have a "proper" version string, i.e. a specified major and minor release number. When a prerelease string was included in the version number, no analysis was performed on the number of breaking changes since `semver` does not state whether prereleases can contain breaking changes. This does not introduce a bias in our study since we want to test whether libraries that *do* have a proper versioning scheme adhere to `semver`.

Not all subsequent versions of methods could be recognized while scanning for the deprecation patterns in Section 10. Library versions were parsed separately, leaving the problem that different objects representing the same method in different versions should be connected with each other. For performance reasons, this was done by text matching of method names and the number of parameters. Overloaded methods with the same number of parameters were not taken into account in this analysis. Future work could further investigate whether deprecation patterns are different for methods with overloaded versions with the same number of parameters.

13.5. Deprecation tags

The low number of deprecation tags detected in the Maven repository is surprising. To make sure all deprecation tags were recognized, we scanned these tags in two different ways. First, a textual search was performed to search for literal occurrences of the string "`@Deprecated`". Second, when a deprecated tag was found in a library, the complete library was parsed 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.

13.6. Selection of independent variables

With respect to construct validity, we chose a set of independent variables in our analysis, such as maturity, size, and the popularity of a library as variables that could influence rework effort and dispersion of errors across systems. We did not have the goal to create an exhaustive list of all possible variables that could influence rework effort and dispersion. There are possibly other variables at work that influence rework and dispersion besides the ones investigated in this paper, which could significantly alter our analysis and conclusions. The restriction to investigate only the 10 most frequently occurring change types could also influence our conclusions. In future work, the influence of other variables can be investigated further.

13.7. Small R^2

The regression analysis in Table 12 has an R^2 of 58.68%, while the regression analysis in Table 16 has an R^2 of only 13.33%. Concerns may rise about the limited explanatory power of these models. The first model only includes 10 most frequently occurring breaking changes, which means that any change that cannot be explained by the independent variables is not taken into account. In other words, 40% of the edit script is explained by changes that cannot be related to the 10 breaking change types. This is expected since it is likely that there will be a large amount of changes which are not associated with any breaking change, such as methods that have a changed implementation without changing their method headers.

The other model has an even lower R^2, but this is expected given the setup of the experiment. The model incorporates factors associated with L_y to explain the dispersion of errors in S_x. The most important factors that would explain this dispersion can likely be found in S_x. For instance, a modular design of S_x with better encapsulation could mean that the calls to a certain library, for instance a database library, are located in a single class. Measuring these factors, however, are not the goal of the model. The p-Value of the model (0) indicates that the model is highly significant. Because of the low R^2 of the model, the model should only be used for explanatory purposes, and not for prediction.

13.8. External validity and generalizability

Our findings are based on an exploration of semantic versioning principles in the Maven repository. It is unknown whether the results can be reproduced in other software repositories mentioned before, such as NuGet, OSGi bundles, Ruby gems,[28] or, for example, the GitHub repository. We have already seen that NuGet has a different approach to update dependencies than Maven, but how often this actually introduces breaking changes with compilation errors is unknown. As mentioned before, other guidelines similar to `semver` have been released, so adherence to these guidelines can be investigated in a similar way as done in this paper. Further research is needed to determine whether the patterns as found in this paper hold in (industrial) software systems instead of open-source software libraries.

14. Conclusion

In this paper, we have looked at semantic versioning principles as adopted by over 100,000 open source libraries distributed through Maven Central. We investigated to what degree the semantic versioning scheme as used by library developers provide library users with signals about breaking changes in that release. Semantic versioning provides developers with a clear set of rules regarding the use of major, minor and patch version numbers, and we have tested these rules on our dataset.

Our findings are as follows:

- The introduction of breaking changes is widespread: Around one third of all releases introduce at least one breaking change. We see little difference between major and minor releases with regards to the number of breaking changes: One third of the major as well as one third of the minor releases introduce at least one breaking change (RQ1).
- Breaking changes have a significant impact on client libraries by introducing compilation errors that need to be fixed before a library upgrade can be performed (RQ2).
- The number of breaking changes in non-major releases has only decreased marginally over time (RQ3).
- Updates of dependencies to major releases are most often performed in major library updates, and similarly for updates of minor and patch releases and dependencies. Major releases of dependencies tend to take longer to be upgraded than minor or patch releases. There exists a small influence of the number of backward incompatibilities and of the amount of change in new versions on this lag (RQ4).
- Bigger libraries tend to introduce more breaking changes and errors. Libraries do not grow when they become more mature, on average, and more mature libraries do not introduce more breaking changes (RQ5).
- 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 (RQ6).
- The size of a client library can influence the dispersion of errors in client systems when breaking changes are introduced in the library (RQ7).

We can conclude that in general, developers spend little effort to communicate backward incompatibilities or deprecated methods in releases to users of their libraries. This manifests itself through a large number of breaking changes in major releases and also becomes visible in the unstructured way of labeling and removing deprecated methods. This leads to a significant number of compilation errors in client systems using these libraries.

Although one can argue that not all developers may be aware of semantic versioning principles in specific, we have assumed that most developers are aware of the intent of these principles: providing information about the amount of work done in a release and providing information about the stability of the interface of the library.

We have demonstrated what the impact can be when developers ignore backward compatibility. We therefore argue that semantic versioning principles should be embraced more widely by the developer community.

References

Andersen, R., 2008. Modern Methods for Robust Regression. SAGE Publications, Inc.
Badri, L., Badri, M., St-Yves, D., 2005. Supporting predictive change impact analysis: a control call graph based technique. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference, APSEC'05.
Balaban, I., Tip, F., Fuhrer, R., 2005. Refactoring support for class library migration. SIGPLAN Not. 40 (10), 265–279.
Bauml, J., Brada, P., 2009. 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, pp. 428–435.
Cossette, B.E., Walker, R.J., 2012. 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. ACM, New York, NY, USA, pp. 55:1–55:11.

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

Dagenais, B., Robillard, M.P., 2008. Recommending adaptive changes for framework evolution. In: Proceedings of the 30th International Conference on Software Engineering, pp. 481–490.
Davies, J., German, D.M., Godfrey, M.W., Hindle, A., 2011. Software bertillonage: finding the provenance of an entity. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 183–192.
Davies, J., Germán, D.M., Godfrey, M.W., Hindle, A., 2013. Software bertillonage – determining the provenance of software development artifacts. Empir. Softw. Eng. 18 (6), 1195–1237.
Dietrich, J., Jezek, K., Brada, P., 2014. Broken promises: an empirical study into evolution problems in Java programs caused by library upgrades. In: Proceedings of International Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE. IEEE, pp. 64–73.
Dig, D., Comertoglu, C., Marinov, D., Johnson, R., 2006. Automated detection of refactorings in evolving components. In: Proceedings of the 20th European Conference on Object-Oriented Programming, pp. 404–428.
Dig, D., Johnson, R., 2006. How do APIs evolve? A story of refactoring: research articles. J. Softw. Maint. Evol. 18 (2), 83–107.
Duraes, J.A., Madeira, H.S., 2006. Emulation of software faults: a field data study and a practical approach. IEEE Trans. Softw. Eng. 32 (11), 849–867.
Fluri, B., Wuersch, M., Pinzger, M., Gall, H., 2007. Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. 33 (11), 725–743.
Henkel, J., Diwan, A., 2005. CatchUp!: capturing and replaying refactorings to support API evolution. In: Proceedings of the 27th International Conference on Software Engineering, pp. 274–283.
Kapur, P., Cossette, B., Walker, R.J., 2010. Refactoring references for library migration. SIGPLAN Not. 45 (10), 726–738.
Lehnert, S., 2011. A Review of Software Change Impact Analysis. Technical Report. Ilmenau University of Technology, Department of Software Systems/Process Informatics.
McDonnell, T., Ray, B., Kim, M., 2013. An empirical study of API stability and adoption in the android ecosystem. Proceedings of 2013 IEEE International Conference on Software Maintenance, pp. 70–79.
Ossher, J., Sajnani, H., Lopes, C., 2012. Astra: bottom–up construction of structured artifact repositories. In: Proceedings of 2012 19th Working Conference on Reverse Engineering, WCRE, pp. 41–50.
Raemaekers, S., Deursen, A.v., Visser, J., 2013. The maven repository dataset of metrics, changes, and dependencies. In: Proceedings of the 10th Working Conference on Mining Software Repositories, pp. 221–224.
Ren, X., Ryder, B.G., Stoerzer, M., Tip, F., 2005. Chianti: a change impact analysis tool for java programs. In: Proceedings of International Conference on Software Engineering, ICSE. ACM, pp. 664–665.
Robbes, R., Lungu, M., Röthlisberger, D., 2012. How do developers react to API deprecation?: The case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 56:1–56:11.
Romano, D., Pinzger, M., 2012. Analyzing the evolution of web services using fine–grained changes. In: Proceedings of International Conference on Web Services, ICWS, pp. 392–399.
Şavga, I., Rudolf, M., 2007. Refactoring-based support for binary compatibility in evolving frameworks. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 175–184.
Tempero, E., Bierman, G., Noble, J., Parkinson, M., 2008. From Java to UpgradeJ: an empirical study. In: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades, pp. 1:1–1:5.
Teyton, C., Falleri, J.-R., Palyart, M., Blanc, X., 2014. A Study of Library Migration in Java Software Journal of Software: Evolution and Process 26 (11), 1030–1052.
Xing, Z., Stroulia, E., 2007. API-evolution support with Diff-CatchUp. IEEE Trans. Softw. Eng. 33 (12), 818–836.
Zhou, Y., Wursch, M., Giger, E., Gall, H., Lu, J., 2008. A bayesian network based approach for change coupling prediction. In: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE'08, pp. 27–36.
Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S., 2005. Mining version histories to guide software changes. IEEE Trans. Softw. Eng. 31 (6), 429–445.