Mining Trends of Library Usage

Yana Momchilova Mileva · Valentin Dallmeier · Martin Burger · Andreas Zeller
Saarland University, Saarbrücken, Germany
{mileva, dallmeier, mburger, zeller}@cs.uni-saarland.de

ABSTRACT

A library is available in multiple versions. Which one should I use? Has it been widely adopted already? Was it a good decision to switch to the newest version? We have mined hundreds of open-source projects for their library dependencies, and determined global trends in library usage. This *wisdom of the crowds* can be helpful for developers when deciding when to use which version of a library—by helping them avoid pitfalls experienced by other developers, and by showing important emerging trends in library usage.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability

General Terms

Measurement, Reliability

1. INTRODUCTION

Most of today's software projects heavily depend on the usage of external libraries. Each of those libraries comes in different versions. One would assume that it is wise to always rely on the latest library version, as it is the version that is currently the most refined one, well-structured and bug-free. However, in practice, things are different.

In this paper, we leverage the *wisdom of the crowds* when it comes to the usage of individual library versions. We consider the choice of the majority to be the wise choice in regard to which library version should be used. To explore the choices made, we mined information from the history of 250 APACHE[1] projects and the external libraries they use, in order to answer the following question: *Which library versions are the most popular ones?* After introducing the concept in Sections 2 and 4, we present three different approaches that analyze the popularity of versions from different angles:

[1]http://www.apache.org/

- In Section 5, we present *usage trends* of different library versions throughout several years and give a visual means for following the evolution of a library version usage.
- Section 6 analyzes the usage of the available library versions *at the present moment.*
- Section 7 discusses a reliability measure for new library versions—namely, the number of times developers have *switched back* from a specific version.

Those three approaches give different information about how different versions of a library are used. Such information can be valuable for both the libraries users as well as for the library developers.

As a result of our analysis, we have developed a tool called AKTARI (Section 8) that assists library users is selecting which library version to use: "Based on the global usage of log4j 1.2.15 it is not recommendable to use this version."

2. POPULARITY OF LIBRARY VERSIONS

The decision to use a specific library version usually depends on factors like reliability, functionality, usability, documentation, quality and compatibility. All of those factors come into play when a user decides whether to use a specific library version or not. By analyzing the choices made by software developers with respect to the usage of library versions, we estimate which are the most popular ones and thus the ones recommended for usage.

Having such information available can be valuable for two groups of developers:

Library users. Library users can highly profit from knowing how frequently a particular library version is used by the majority. Suppose, for example, that many people have switched back from a particular version—due to a bug in it, for example. Then, warning potential new users of this version can help them avoid facing the same issues again. Users can thus save time and improve the quality of their software product.

Library developers. Up to now, library developers did not have any other means for getting user feedback except direct communication with the users. With our approach, they can evaluate how successful a particular version is and thus improve the future development of their library. Our technique thus gives library developers a means of collecting indirect feedback from users to provide better service.

3. EARLY ADOPTERS VS. LATE FOLLOWERS

Like with any other product, it takes time until the majority of the users starts using it. According to Rogers' theory of *Diffusion of innovation* [6] the adoption of an innovation follows an *S* curve and there are several categories of innovation adopters. These include:

Early adopters (called *innovators* by Rogers) are those users who immediately start using a new technology and who are willing to take the associated risks.

Late followers (called *late majority* by Rogers) are those users who adopt a new technology after the majority of the users' community has already done so.

When it comes to library versions usage, users can be classified in a similar way—those who start using the new library version immediately after it is out and those who prefer to wait and see if it is safe to switch. Both approaches have their positives and negatives and every user has reasons for such behavior, such as the need for new functionality (early adopters) or security (late followers).

Whatever the reasons behind the decision to switch, our approach can give valuable information. The recommendations made by AKTARI most likely won't influence early adopters, as they know about the risks and benefits of being the first ones to use a new version and their reasons for switching are not risk-driven. However, our approach can assist late followers in making an informed decision about which library version to use.

4. ANALYZING THE USAGE OF LIBRARY VERSIONS

In this paper we focus on the analysis of JAVA programs. More specifically, we focus on programs that are managed by MAVEN[2], a widely adopted project management tool for JAVA projects. In MAVEN, library dependencies are stored explicitely in meta information files, which makes it easy to extract and analyze version usage.

To analyze the global usage of library versions, we used 250 real-life open-source projects from the APACHE foundation, one of the largest and most popular repositories for open-source JAVA projects. In total, we analyzed the usage of 450 different external libraries versions per month over the period of two years.

In a project managed by MAVEN, meta data describing the project data is represented by MAVEN's *Project Object Model* (POM). MAVEN relies on the presence of a central repository that stores different versions of commonly used libraries.

Since its first release in 2002, MAVEN has become the leading open-source project management tool for JAVA projects. The actual number of projects using MAVEN cannot be measured directly. However, in September 2008, the central MAVEN repository was hit over 250 million times [5]. Usually, a MAVEN installation checks the central repository only once per day, which implies that MAVEN is actively used by a large number of developers.

In MAVEN, required libraries are described as dependencies in an XML file called *pom.xml*, a descriptive declaration of a POM. Library usage information is stored in `<dependency>` elements. These elements list all the libraries that the project depends on. Each element has three mandatory children: a *group id* (company, team, organization, or other group), an *artifact id* (unique id under group id that represents a single dependency), and a *version* (specific release). These three components uniquely identify a specific version of a library. Thus, `<dependency>` elements define all required libraries *unambiguously*, including their version number. Figure 1 shows an excerpt from a *pom.xml* file.

[2]http://maven.apache.org/

```
<project>
    <dependencies>
        <dependency>
            <groupId>servlet-api</groupId>
            <artifactId>javax.servlet</artifactId>
            <version>2.3</version>
        </dependency>
    </dependencies>
</project>
```

Figure 1: Declaring a dependency to the *servlet-api* library in a MAVEN Project Object Model

To analyze the usage of library versions, we collected dependencies for all of the 250 APACHE projects on a monthly basis over a period of two years. For each month, we determined the library usage information for all dependencies referenced by at least one project.

5. LIBRARY VERSION USAGE TRENDS

Suppose you are head of a team developing a library. How can you make sure that you are providing the best service for its users? In order to determine the evolution of the usage of libraries and their versions, we mined the entire archive history of the aforementioned 250 APACHE open-source projects for the period January 2007 to January 2009 (excluding).

As a first example, consider the usage trend of the *junit* library for this period. In Figure 2, one can see that version *3.8.1*, which is the oldest version of this library used in that period of time, remained the most popular and widely used one. It was even more popular that the latest *4.4* version. From a user's point of view, we investigated the reason behind this behavior and found out that there was a big API

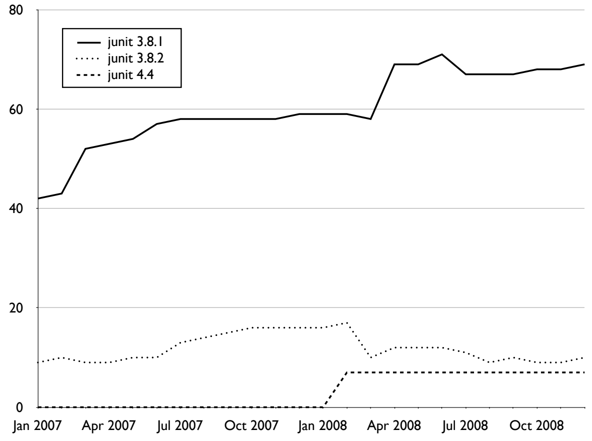


Figure 2: Usage trends of the *junit* library

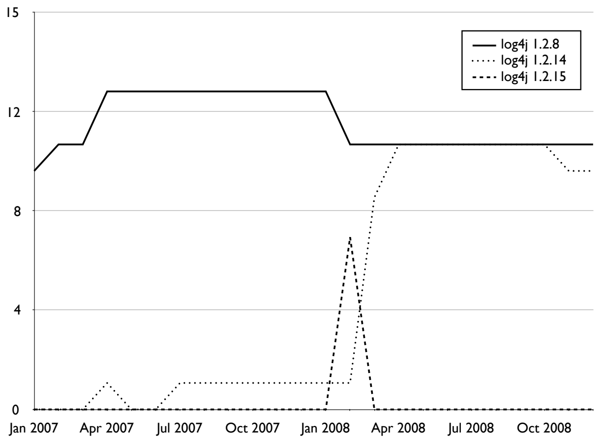


Figure 3: Usage trends of the *log4j* library

change from *3.x* to *4.x*. There are examples of projects' problem reports where it is stated that switching will be "lots of work" (e.g. *Gentoo bug entry 129773*). Also due to the change in the API there was an issue with backwards-compatibility—we found examples of users who decided to stick to the older versions as the *4.x* ones had compatibility issues with the *ant* library. Also, the *4.x* versions required a newer *jdk* version; the developers were concerned that this might be a reason for their clients to not use their product.

Figure 3 shows another example of usage trends of the different versions of the *log4j* library. The usage of *log4j 1.2.8* was at its highest point in mid-2007. What strikes however is the usage of the *log4j 1.2.15* version: At the moment it was released, there was a peak in its usage history and then a fast and sudden drop shortly afterwards. This was due to a bug in its implementation (see *Apache bug entry 43304*). Projects decided to switch to the new version (indicated by a drop in the usage of the *1.2.8*), but after discovering the bug they switched back (drop in *1.2.15*) and switched to the closest earlier version (increase in *1.2.14*). The fact in this case is that *1.2.15* was rejected by its users.

Those are only a few of the many examples of trends in library versions usage. For developers of libraries, such trends clarify the ways the users are using individual versions. As developers know what the differences between each version are, they can link the specific library version features to the library popularity and thus analyze the users' needs.

Trends in library usage are a method for displaying the preferences of the users.

6. MOST POPULAR VERSIONS

When a software developer decides to use a library, she has to decide which of the available versions is the best one to use. As this choice is made at a certain fixed moment in time, the recommendation should also be based on the usage at this particular moment in time.

To identify the most popular and thus recommendable library versions for usage in January 2009, we mined the 250 projects and their library dependencies for January 2009. Some of our results are depicted in Table 1.

To measure the popularity of a particular version, we consider the number of current usages of the version the user wants to switch from and the number of current usages of the version the user wants to switch to. For example, for a developer using *derby 10.1* that wants to switch to *derby 10.2*, we would recommend *not* to do so, as in only *#derby 10.2*/(*#derby 10.2* + *#derby 10.1*) = 1/(1+6) = **14%** of the cases version *10.2* is used. We again investigated the reasons behind this usage behavior and found a commit message stating that the developers will stick to version *10.1*, "until TranQL can handle 10.2", which reveals a compatibility problem in the newer *10.2* version. Developers should be warned about such issues with the versions in order to make a better informed choice as of which version is recommendable for them to use.

Table 1: Usage of library versions for January 2009

Library name and version	Times used
junit 3.8.1	60
junit 3.8.2	9
junit 4.4	7
log4j 1.2.8	10
log4j 1.2.14	9
log4j 1.2.15	0
servlet-api 2.3	4
servlet-api 2.5	1
derby 10.1	6
derby 10.2	1

Knowledge about popular versions helps developers in deciding which versions to choose.

Of course, every developer may have individual reasons when and why to switch to a new version. However, if a large majority of developers uses a specific library version (e.g. *junit 3.8.1*) this information should be taken into account.

7. SWITCHING BACK TO EARLIER VERSIONS

When the developers of a software project switch from an old library version to a newer one, they usually do so either because the old version had problems that were fixed in the new one or because the new one offered more and/or better functionality. On the other hand, there are users who prefer to wait before they switch—such that once they switch, they will not have problems with a defective library version. However, identifying when it is safe to switch is a difficult task. Here again, the vote of the majority comes into play.

For projects that migrate early to a new library version, it might be that they *migrate back* to an older version. In most cases, the reason for switching back is that the new version has some issues that make it unusable for the specific needs of the project. If this is the case, the end user should be *warned* about such library versions and should avoid switching to them.

Again, we have mined the same 250 APACHE projects and their history (January 2007–January 2009). However, this time we were interested in the number of times people switched back from a particular library version.

Table 2 presents the cases of the *junit*, *log4j*, *servlet-api*

Table 2: Switching back to older library versions for the period January 2007–January 2009

Library	# usages	# switched back	%
junit 3.8.1	1501	0	0%
junit 3.8.2	293	1	<1%
junit 4.4	84	0	0%
log4j 1.2.8	269	3	2%
log4j 1.2.14	114	0	0%
log4j 1.2.15	**7**	**4**	**57%**
servlet-api 2.3	182	0	0%
servlet-api 2.5	10	1	10%
derby 10.1	147	0	0%
derby 10.2	31	0	0%

and *derby* libraries (these libraries are among the top most widely used libraries in our set of projects for the specified period). The first column in the Table gives the library name and version. The second column shows the number of times a particular library version was used in this period. The third column shows how frequently this specific library version was discarded, and the developers switched back to an *older* version in the same period. The fourth column gives the percentage of times a particular library version was switched back from.

Switching back and forth between versions is very time consuming and can introduce bugs in the code (if, for example, the library API has changed, the project code also has to be changed). That is why developers usually do not switch back from a particular library version once they have switched to it—unless it really has a problem. As one can see, the most popular version of *junit* is *3.8.1.* No project ever switched back from using this version, thus indicating that this is a very good version of *junit* to use. The *servlet-api 2.5* version, for example, was switched back from in 10% of the cases. We found bug reports pointing to a problem the *2.5* version has with *Tomcat 5.5* (e.g. *Grails bug entry 2053*). The case that strikes the most, however, is the *log4j 1.2.15* version, as in no less than *57%* of the cases people switched back from it. We investigated this case and found out that the reason why so many users decided to switch back is a bug in this version that prohibited its usage for all MAVEN projects that depended on *log4j*, but did not depend on the *java mail* and *jms* libraries. The problem with this library version is also indicated in Figure 3 and Table 1, and was thus detected by all of our techniques.

Finding reverts to previously used versions can show the library developers how big the impact of a library issue is. It can also show the library users how reliable a specific library version is and thus give them yet another indication if they should switch to it.

The number of times a library version was switched back from is a strong indicator of the quality of the library.

8. RECOMMENDATIONS WITH AKTARI

We have presented three approaches for analyzing the usage of library versions. All of them give information that is complementing the others and should thus be used together.

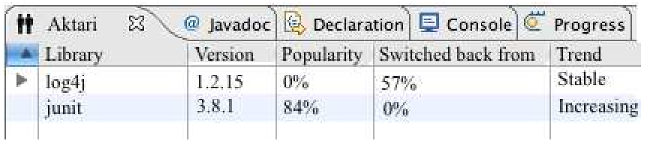
Combining usage trends with times users switched back from a library gives an indication to the library developers what design mistakes they made and also how big the impact of a bug in their library is.

When giving recommendations as to which library to use, one should take into account as many factors as possible. For new projects, backwards-compatibility for their clients is not an issue—in this case considering only the popularity at a certain moment in time might be misleading, as it takes into account all kinds of factors, including the age of the projects. However, adding information about how many times a library was switched from will give a better recommendation, since it also considers if a library is bug-free and thus the developers of a new project will be able to choose the newest and most reliable version.

On the other hand, considering only the popularity of a version disregarding the emerging increasing and decreasing trends might also be misleading. This is why we believe that all of the presented approaches bring different information and combined they are a powerful tool for library versions usage recommendations.

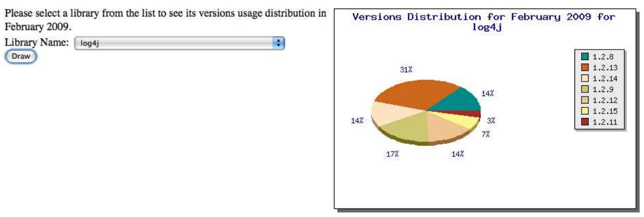
We have combined and integrated our library analysis techniques into a tool called AKTARI[3] that comes in two forms—an Eclipse plug-in and a Web tool.

The plug-in assists library users in selecting the most recommendable, according to the majority of users, library version. It detects which versions are being used by the project and gives information regarding the global usage of these versions (see Figure 4—tool's planned design).



Library	Version	Popularity	Switched back from	Trend
log4j	1.2.15	0%	57%	Stable
junit	3.8.1	84%	0%	Increasing

Figure 4: AKTARI—Eclipse plug-in design



Please select a library from the list to see its versions usage distribution in February 2009.

Library Name: log4j

Figure 5: AKTARI—web-tool

The Web tool (Figure 5) is available for the library developers who want to check the usage trend of their library. It offers diagrams (like the ones in Figure 2) as well as Pie charts that represent each of the three analysis techniques described, and thus assists developers in analyzing usage, success and popularity of their library. It is available at
`http://www.st.cs.uni-saarland.de/softevo/aktari.php`[4]

9. THREATS TO VALIDITY

As any empirical study, this study has limitations that must be considered when interpreting its results. We identified the following threats to validity.

[3]"Aktari" is the Swahili word for "crowd".

[4]Publicly available starting June 7th, 2009.

The number of projects may affect the outcome. It is possible that the addition or the removal of a particular project from the set of analyzed projects might influence the results. However, we have mined hundreds of projects and we believe that this possibility is small.

Results might not hold for non-MAVEN projects. The advantage of MAVEN is that it eases dependency management. However, it does not impose restrictions on which version of a library can be used. Also, the libraries used in a project depend on the scope of the project and not on its management tools. We are therefore convinced that our results are also valid for projects that do not use MAVEN.

The implementation may have errors. A final source of threats is that our implementation could contain errors that affect the outcome. To control these threats we did a careful cross-checking of the data and the results to eliminate mistakes in the best possible way.

10. RELATED WORK

A lot of related work has been done to support developers in adjusting their code to a new version of a library.

SpotWeb [7] is a tool that crawls open source repositories to mine frequent usage patterns for libraries. These patterns are then presented to a developer that wants to start using a library. In contrast to this work, our approach tries to suggest when a developer should switch to a new version of a library.

Another tool that aims at making the process of switching versions easier is CatchUp! [3]. It is a plugin for Eclipse that records refactoring operations applied when switching to a new library. Recorded refactorings can then be replayed for clients that also want to switch to that version.

Dagenais and Robillard [1] use a partial program analysis technique to suggest replacements for calls to methods that are no longer present in the new version of a library. The presented tool gathers suggestions by mining the version history of the library and is based on the assumption that changes to replace a deleted method happen in the same change set.

Holmes and Walker [4] analyze library's API popularity and based on the usage frequency of the API elements direct the library users to using the popular ones.

To the best of our knowledge, our approach is the first that tries to recommend or dissuade from switching library *versions* based on global usage history.

11. CONCLUSION AND FUTURE WORK

There are many different libraries available and each of them exists in many different versions. Thus, choosing the right version to use is a potential problem for many developers.

This paper presents an approach to supporting developers in this decision. The technique is based on the popular vote of the majority. The more people use a particular version, the higher its usage is recommended.

The results presented can also serve library developers when trying to evaluate the quality of their projects. Seeing the popular vote of the majority clearly shows the opinion of users regarding a particular library version.

Generally, we believe that trends in software, as accumulated over the histories of hundreds of projects, can result in recommendations that are immediately useful. We want to further explore the potential of such wisdom of the crowds and have already identified a number of ways this work can be extended and improved:

Identify Early Adopters and Late Followers. In order to be able to give recommendations that are more project-specific we plan to differentiate between early adopters and late followers projects and give them library usage recommendations accordingly.

Include Bug Fix Information. In order to further advocate the case of switching to a new version, we plan to include bug data: ”In version *3.2* 50 bugs were fixed and 10 of them relate to your project. Switching to version *3.2* will fix existing issues in your code.”

Cost vs. Benefit of Switching. In order to assess the cost of switching, we plan to use static and dynamic program analysis techniques and to compare the source and bytecode before and after the switch [2, 8].

Acknowledgments: Yana Mileva is funded by Microsoft Research Cambridge Lab. Valentin Dallmeier is supported by the Saarland Graduate School of Computer Science. The authors thank Andrzej Wasylkowski for his comments on the paper as well as the anonymous reviewers for their feedback.

12. REFERENCES

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

[2] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA 2006: ICSE Workshop on Dynamic Analysis*, May 2006.

[3] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[4] R. Holmes and R. J. Walker. Informing Eclipse API production and consumption. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 70–74, New York, NY, USA, 2007. ACM.

[5] S. M. Kerner. Apache Maven Goes Commercial. `http://www.serverwatch.com/news/article.php/3784681`, November 2008.

[6] E. M. Rogers. *Diffusion of Innovations*. The Free Press, 1962.

[7] S. Thummalapenta and T. Xie. Spotweb: detecting framework hotspots via mining open source repositories on the web. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 109–112. ACM, 2008.

[8] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–44, September 2007.