Understanding When to Adopt a Library: A Case Study on ASF Projects

Akinori Ihara, Daiki Fujibayashi, Hirohiko Suwa, Raula Gaikovina Kula, and Kenichi Matsumoto

Nara Institute of Science and Technology
{akinori-i, fujibayashi.daiki.eq3, h-suwa, matumoto}@is.naist.jp
Osaka University
raula-k@ist.osaka-u.ac.jp

Abstract. Software libraries are widely used by both industrial and open source client projects. Ideally, a client user of a library should adopt the latest version that the library project releases. However, sometimes the latest version is not better than a previous version. This is because the latest version may include additional developer effort to test and integrate all changed features. In this study, our main goal is to better understand the relationship between adoption of library versions and its release cycle. Specifically, we conducted an empirical study of release cycles for 23 libraries and how they were adopted by 415 Apache Software Foundation (ASF) client projects. Our findings show that software projects are quicker to update earlier rapid-release libraries compared to library projects with a longer release cycle. Moreover, results suggest that software projects are more likely to adopt the latest version of a rapid-release library compared to libraries with a longer release cycles.

1 Introduction

A software library is a collection of reusable programs, used by both industrial and open software client projects to help achieve shorter development cycles and higher quality software [8]. Many of these libraries are open source software and are readily available through online repositories such as the GitHub[1] repository. To incorporate bug fixes and new features, open source library projects often release newer and improved versions of their libraries. Based on user feedback, libraries evolve faster to reach the market, making it difficult for client projects to keep up with the latest version.

Ideally, a client user of a library should adopt the latest version of that library. Therefore, it is recommended that a client project should upgrade their library version as soon as possible. However, the latest version is not always better than previous versions [5, 9], as adoption of the latest version may include additional developer efforts to test and integrate changed features [7, 10, 13]. Developers of client projects may be especially wary of library projects that

[1] https://github.com

follow a rapid-release style of development, since such library projects are known to delay bug fixes [12]. Recent studies investigated the dependency relationships between evolving software systems and their libraries [5, 6, 15]. These tools makes it possible for developers to clarify and visualize these dependencies and aim to guide developers who are selecting possible candidate libraries for an upgrade.

In this study, our main goal is to better understand the relationship between the adoption of library versions and the library release cycle. Specifically, we conducted an empirical study of the release cycle of 23 libraries and how they were adopted by 415 Apache Software Foundation (ASF) client projects. These 23 libraries were used by over 50 software projects of our target ASF client projects. To guide our research, we address the following two research questions:
RQ1: Does the release cycle of a library project influence when it is adopted by a client project?
Recent studies [7, 10, 13] have found that open source software often has many issues soon after its release. Often these libraries are reactive in fixing issues based on user feedback. In other words, these software may be harmful in the early period after the release. Therefore, for RQ1, we would like to understand the effect of client project adoption on shorter release cycles.
RQ2: Does the release cycle of a library project influence whether the latest version is adopted by a client project?
Recent studies have shown that the newest version of a library is not always adopted by many client projects. For example, client projects may decide not to adopt the latest version to avoid untested bugs, especially if the library project has a shorter release cycle. Therefore, for RQ2, we would like to understand the effect of adopting the latest client project on shorter release cycles.

Our findings show that software projects are quicker to update earlier rapid-release libraries compared to library projects with a longer release cycle. Moreover, results suggest that software projects are more likely to adopt the latest version of a rapid-release library compared to libraries with a longer release cycles.

2 Background and Definitions

2.1 Motivation

Related work such as Almering et. al [1], Goel et. al [3] and Yamada et. al [14] all investigate when a software is ready to be used. These works use the Software Reliability Growth Model (SRGM) of the software evolution process to grasp the process of the convergence of defects discovered in software as the *'growth curve of the S-Shape (Sigmond curve)'*. Similarly, Mileva et. al [9] evaluated a library by its library usage by clients.

Building on this work, we conducted an exploratory investigation of when developers adopted versions of a library. Fig. 1 shows the release date (broken lines) for the library `log4j` and the number of ASF projects (solid lines) which have adopted the new library version in a time series. The figure shows users of

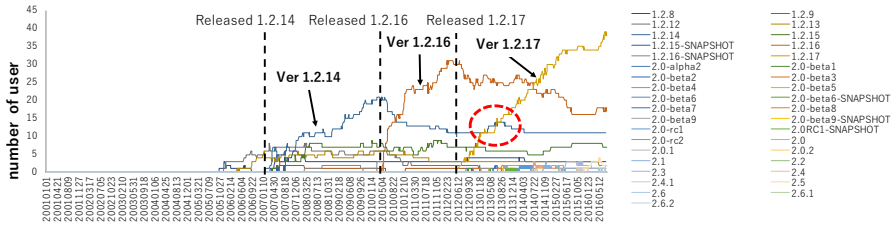

Fig. 1. Adoption trends based on client usage

the popular `log4j` library, mined from 797 software projects. From this work, we highlight two points: (1) library adoption is not organized, with no clear patterns of migration and (2) in many cases the latest version is not always selected as the default option. For instance, in Fig. 1 we can see `ver.1.2.14` is still being used by some client projects (red dotted circle), even though the latest version is `ver.1.2.17`.

In this paper, we define the *"release cycle"* as the time until a new version is released. As a cycle, usually a project will have a fixed release timing from as quickly as 1 day to a span of across several years. Due to agile development trends, we assume that the release cycles may become faster. For instance, the Google Chrome project and the Mozilla Firefox project are working on rapid release to develop a new version in 6 weeks [4]. A rapid-release cycle is beneficial in that it can fix a bug and make a new component quickly. Sometimes, these projects can be reactive in bug fixing, for example, projects can get feedback from users soon after their release [8]. However, this rapid release style creates an influx of releases, which is likely to further confuse users on when to adopt a new version. Therefore, our motivation is to investigate when and how software projects adopt a new library relative to their release.

2.2 Library Adoption and Release Timings

Fig. 2 describes the evolution and adoption of a library during different release cycles. We use this figure to explain how we measure the timing of adoption relative to each release, including the relative definition of the latest release. This example shows a project S and two libraries (A, B). Library A has released versions $A1$ and $A2$, with $A2$ being the latest version. Similarly, Library B has released versions $B1$, $B2$, $B3$ and $B4$, with $B4$ being the latest version.

This example also shows library adoption. Specifically, we see that project version $S3$ imports the library $A1$, which is not the latest version at this point in time. This is because at this time, $A2$ was available for selection. $S3$ also imports library $B3$, which is the latest version at this time. However, we see that in the near future B4 will be released.

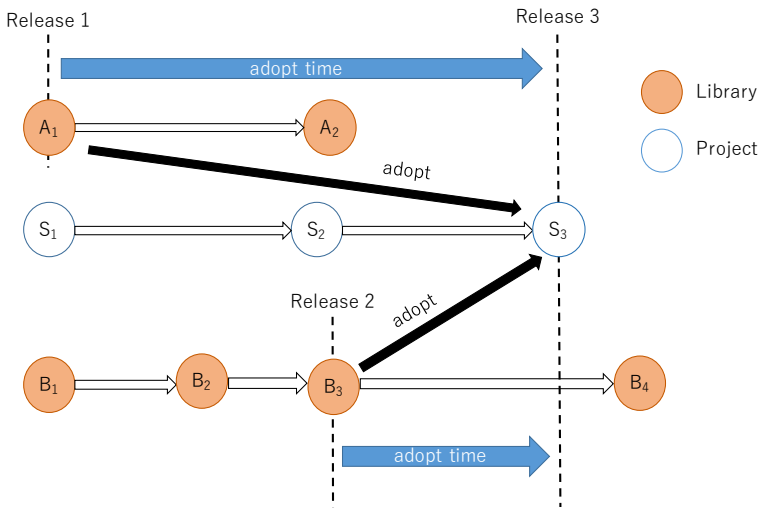

Fig. 2. Release cycle and adoption period

3 Empirical Study

Adopting the latest versions has the added benefits of new features, but adapting the latest version may also risk having untested bugs or removed features. Therefore, the goal of this study study is to understand the impact of the release cycle on the developers decision whether to wait for the next library release or quickly adopt the latest version.

3.1 Data Preparation

Table 1 shows the top 23 of 4,815 libraries which the 415 software projects used. In total, these 23 libraries were used by over 50 software projects. These libraries were originally extracted from 415 projects of 797 ASF projects which are using MAVEN dependency tool on July 21, 2016. To analyze the library adoption and release timings, we extracted histories of library dependency information. Our dataset comprises of JAVA programs managed by the MAVEN dependency tool. MAVEN stores explicitly in meta information files (POM.xml). The meta information contains the libraries' names and the version number in which the software is adopted. By tracking the history of the POM.xml in a version control system, for any software, we can know when and which library version has been adopted.

3.2 Clustering Libraries by Release Cycle

In order to evaluate the impact of the release cycle, the rank and grouping of libraries based on their release cycles is needed. Hence, for each library, we

Table 1. Ranking of library users

Rank	Library	Num	Rank	Library	Num
1	junit	305	16	easymock	67
2	commons-logging	167	17	jackson-mapper-asl	60
3	log4j	153	18	commons-cli	55
4	slf4j-api	145	19	jackson-core-asl	53
5	commons-lang	130	20	mail	53
6	commons-io	122	21	velocity	52
7	slf4j-log4j12	109	22	jcl-over-slf4j	52
8	servlet-api	99	23	mockito-all	52
9	commons-collections	98			
10	commons-codec	96	....	.....	....
11	commons-httpclient	83			
12	guava	81	....	.....	....
13	ant	73			
14	xercesImpl	69	....	.....	....
15	jetty-server	68	4815	axis2-transports	1

compute and assign a [2] variable importance score for each library. We then use the *Scott-Knott test* [11] to group libraries into statistically distinct ranks according to their release periods. The *Scott-Knott test* is a statistical multi-comparison procedure based on cluster analysis. The *Scott-Knott test* sorts the percentage of release periods for the different libraries. Then, it groups the factors into two different ranks that are separated based on their mean values (i.e., the mean value of the percentage of release periods for each library). If the two groups are statistically distinct, then the *Scott-Knott test* runs recursively to further find new groups, otherwise the factors are put in the same group. The final result of the *Scott-Knott test* is a grouping of factors into statistically distinct ranks.

Table 2 shows the 6 categories (ie., C1, ..., C6) in which each of the 23 studied libraries were categorized. Based on these 6 groupings and the dataset, we are now able to address the research questions in our results.

4 Results

RQ1: Does the release cycle of a library project influence when it is adopted by a client project?

To answer RQ1, we use the clustered libraries groupings to compare release and adoption times. As a result, we make the following observations:

Observation1—All top frequent libraries are not released in one year. The boxplot in Fig. 3 shows the distribution of the periods between releases in each library. The libraries are sorted by the number of adopted software projects. While some library projects (e.g.,`jetty-server`, `jackson-mapper-asl`, `mockito-all`) often release new versions in one year, other library projects (e.g., `commons-cli`, `servlet.api`, `commons-logging`) often release new versions after more than one year. In particular, releases for the `commons-cli` project were

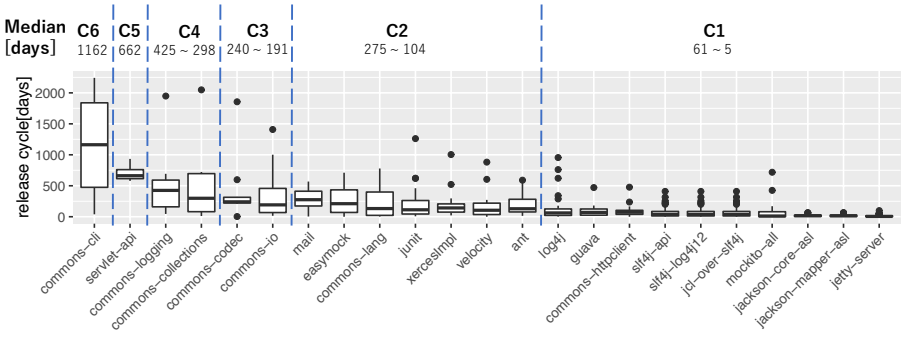

Fig. 3. The release cycle of each library by boxplot. The target libraries are sorted by clustering (broken lines) from C1 to C6. The top figure shows the clustering number and the median of the release cycle days.

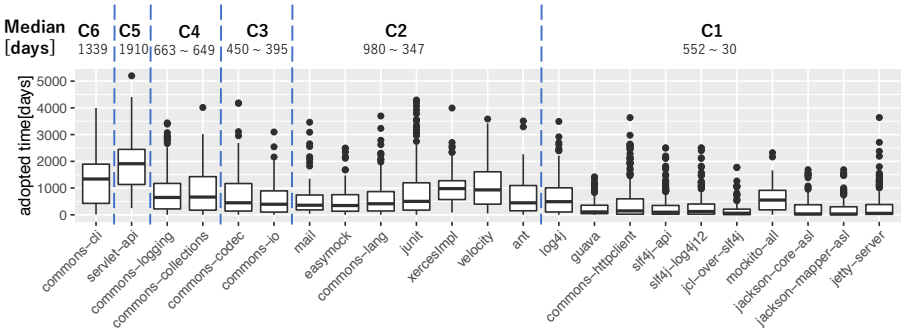

Fig. 4. The boxplot shows the adoption time of each library. The target libraries are sorted by clustering (broken lines) from C1 to C6.The top figure shows the clustering number and the median of the adoption time [days].

delayed for a consideration time.

Observation2—While older and established projects often release new versions after more than one year, beginner projects often release new versions in three months. Through our analysis, we found the different features between quick-release projects and late-release projects. Table 2 shows the working period with GitHub for each library project. Traditional projects that have worked for 10 years often release new versions after more than one year.

Observation3—While software projects have adopted the quick-release libraries soon after the release, they have not adopted the late-release libraries as quickly. The boxplot in Fig. 4 shows the distribution of the adopted periods for our target projects in each library. We found that software projects have adopted the quick-release libraries (sixth group). In other

Table 2. Clustering by library release cycle and each library start of the release date

Library	Date	Library	Date
Cluster 6		**Cluster 1**	
commons-cli	Nov.6,2002	log4j	May.1,2002
Cluster 5		guava	Sep.15,2009
servlet-api	Sep.25,2001	commons-httpclient	Aug.31,2001
Cluster 4		slf4j-api	Mar.8,2006
commons-logging	Aug.13,2002	slf4j-log4j12	Mar.8,2006
commons-collections	Apr.2,2002	jcl-over-slf4j	Mar.8,2006
Cluster 3		mockito-all	Feb.28,2008
commons-codec	Apr.25,2003	jackson-core-asl	Jan.14,2009
commons-io	Jul.2,2007	jackson-mapper-asl	Jan.14,2009
Cluster 2		xercesImpl	Mar.29,2009
mail	Feb.22,2000		
easymock	Aug.8,2001		
commons-lang	Nov.25,2002		
junit	Dec.3,2000		
xercesImpl	Jan.30,2002		
velocity	Jul.7,2002		
ant	Jul.19,2000		

words, they often adopt new versions soon after their release. On the other hand, software projects have adopted the late-release libraries (1st-2nd). This means that they do not adopt new versions quickly after the release.

In the group of the quick-release cycle, the adopted time of the `mockito-all` library is longer than the other libraries. To understand the reason, we analyzed software projects which adopted the `mockito-all` library. As we can see in Fig. 3, there are some outliers for `mockito-all`. Those are some versions which took a long time to release a new version. In particularly, version 1.9.0 was released approximately 1 year after releasing version 1.8.5. Also, version 1.10.0 was released approximately 2 years after releasing version 1.9.5. While waiting for the version 1.10.0, many software projects started using the `mockito-all` library just before releasing the version. In addition, although the `Velocity` library was adopted in a comparatively quick-release project, most software projects adopted the `Velocity` library a relatively long time after the release. The results show that many projects still started adopting the `Velocity` library after the project released the newest version on November 29th, 2010. Therefore, to answer RQ1, we find that:

> Software projects are more quickly updated than rapid release libraries compared to library projects with a longer release cycle.

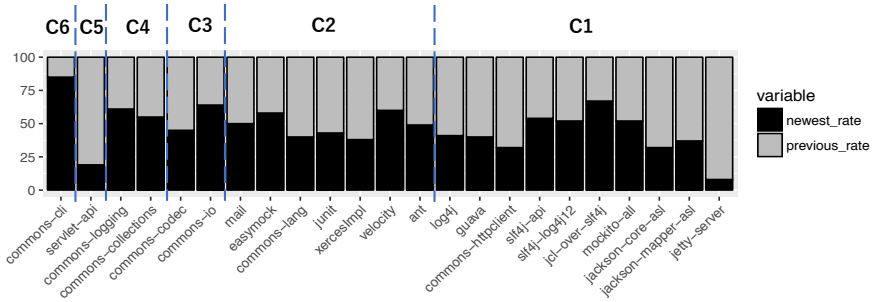

Fig. 5. Figure showing the adoption status rate of each library. The black bar means the adoption rate of newest version. The gray bar means the adoption rate of previous version. As showing by rate, the vertical axis means 100%

RQ2: Does the release cycle of a library project influence whether the latest version is adopted by a client project?

To answer RQ2, we use the clustered libraries groupings to investigate whether the latest version of a library was adopted. As a result, we make the following observations:

Observation4—Software projects do not always adopt new library versions in their projects.

Fig. 5 shows the percentage of the newest or previous versions which software projects adopted from each library. The black and gray bars show the newest adopted version and the previous version applied to the software projects. We found 8%-85% of software projects adopted the new library versions. The `commons-cil` library often adopted the newest version to the software. On the other hand, the `jetty-server` library was often adopted the previous version to the software.

Observation5—While the quick-release library often adopts the newest version to the software, the late-release library often adopts the previous version to the software.

85% of the `commons-cli` library changes were applied to the newest version. This library project has released only 4 new versions during our target period (16 years). This number of releases is fewer than for the other library projects. Furthermore, one of the versions contained a new feature and maintenance bug fixes. The other two versions contained dozens of bug fixes. From this analysis, the project just maintained the initial a stable version.

On the other hand, only 19% of the `servlet.api` library changes were applied to the newest version although it is a late-release library. This library project has released only 7 new versions during our target period. To understand this strange result, we analyzed the history of applying the library. We found that version 2.5 is the majority even if the project released newer versions.

92% of the `jetty-server` library changes were applied to the previous version. This library project has released 267 new versions with most release intervals ranging from 0 to 20 days during our target period. 267 new versions show a clear contradiction to the `commons-cli` library and *servlet-api* library. Furthermore, we found that version 6.1.26 is the majority, even if the project released newer versions. In sum, to answer RQ2, we find that:

> Software projects are more likely to adopt the latest version of a rapid-release library compared to a library with a longer release cycles.

5 Conclusions and Future Work

In this study, we revealed the relationship between the release cycle and the time it takes to adopt a library. Our results suggests that the shorter the release cycle, the shorter the time to be adopted, and that the rapid-release library will be adopted faster even in the same release cycle. Also, for libraries with majority versions, it is difficult to adopt the latest version. We find that it is especially difficult to generalize the reason for adopting a previous version. We think that the reasons are clarified by analysis of the released version. In detail, we believe that reasons will be clarified by analyzing the number of bug fixes and the number of added functions to the released version. These factors are important when selecting a library although there are still many challenges in finding other factors.

In this study, we considered the adoption situation only by the adoption time and whether the version is the newest adoption or a previous adoption of the OSS library. We confirmed that the version was adopted, but we did not also consider the state after adoption. When downgrading a version, we think that the reason should be extracted from the commit log. Further research is needed to confirm this. Also, there are cases where users changed to a version whose adoption was skipped or a library with the same function. Future work will include how to analyze these cases and to clarify what influence what this has on library selection.

Acknowledgments

This work was supported by the JSPS Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers: Interdisciplinary Global Networks for Accelerating Theory and Practice in Software Ecosystem and the Grant-in-Aid for Young Scientists (B) (No. 16K16037).

References

1. Almering, V., van Genuchten, M., Cloudt, G., Sonnemans, P.: Using software reliability growth models in practice. In: IEEE Software, pp. 82–88 (2007)
2. Breiman, L.: Machine Learning. Kluwer Academic Publishers (2001)
3. Goel, A.L.: Software reliability models: Assumptions, limitations, and applicability. IEEE Transaction on Softwware Engineering **11**(12), 1411–1423 (1985)
4. Khomh, F., Dhaliwal, T., Zou, Y., Adams, B.: Do faster releases improve software quality?: An empirical case study of mozilla firefox. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, pp. 179–188 (2012)
5. Kula, R.G., German, D., Ishio, T., Inoue, K.: Trusting a library: A study of the latency to adopt the latest maven release. In: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, pp. 520–524 (2015)
6. Kula, R.G., Roover, C.D., German, D., Ishio, T., Inoue, K.: Visualizing the evolution of systems and their library dependencies. In: Proceedings of the 2014 Second IEEE Working Conference on Software Visualization, pp. 127–136 (2014)
7. Mäntylä, M.V., Adams, B., Khomh, F., Engström, E., Petersen, K.: On rapid releases and software testing: A case study and a semi-systematic literature review. Empirical Software Engineering **20**(5), 1384–1425 (2015)
8. McCarey, F., Ó Cinnéide, M., Kushmerick, N.: Knowledge reuse for software reuse. Web Intelligence and Agent Systems **6**(1), 59–81 (2008)
9. Mileva, Y.M., Dallmeier, V., Burger, M., Zeller, A.: Mining trends of library usage. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol), pp. 57–62 (2009)
10. Plate, H., E. Ponta, S.: Impact assessment for vulnerabilities in open-source software libraries. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), pp. 411–420 (2015)
11. Scott, A.J., Knott, M.: A Cluster Analysis Method for Grouping Means in the Analysis of Variance, vol. 30. International Biometric Society (1974)
12. Tosin Daniel Oyetoyan, D.S.C., Thurmann-Nielsen, C.: A decision support system to refactor class cycles. In: 2015 IEEE 31st International Conference on Software Maintenance and Evolution (ICSME) (2015)
13. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad. In: Proceedings of the 37th International Conference on Software Engineering (ICSE), pp. 403–414 (2015)
14. Yamada, S., Ohba, M., Osaki, S.: S-shaped reliability growth modeling for software error detection. In: Trans. Reliability, pp. 475–484 (1983)
15. Yano, Y., Kula, R.G., Ishio, T., Inoue, K.: Verxcombo: An interactive data visualization of popular library version combinations. In: Proceedings of the IEEE 23rd International Conference on Program Comprehension, pp. 291–294 (2015)