Which API Lifecycle Model is the Best for API Removal Management?

Dung-Feng Yu, Cheng-Ying Chang

Institute of Computer and
Communication Engineering,
National Cheng Kung University,
Tainan, Taiwan
Email:{dfyu, zhengying}@nature.ee.ncku.edu.tw

Hewijin Christine Jiau, Kuo-Feng Ssu

Department of Electrical
Engineering,
National Cheng Kung University,
Tainan, Taiwan
Email:{jiauhjc, ssu}@mail.ncku.edu.tw

***Abstract*—Frameworks and libraries are reused through application programming interfaces (APIs). Normally, APIs are assumed to be stable and serve as contracts between frameworks/libraries and client applications. However, in reality, APIs change over time. When these changes happen, API users must spend additional effort in migrating client applications. If the effort is too much to afford, the frameworks/libraries that API developers have built will lose market share. In order to reduce migration effort, API developers should manage API changes through API lifecycle. Before construction of API lifecycle, investigation of the following question is required: Which API lifecycle model is the best for API removal management? To answer this question, we first construct three API lifecycle models based on the observation of current practices, and then devise a set of metrics to assess those models using case studies. Assessment results conclude the best model is deprecation involved model which benefits API developers and users most with the least costs. Such model becomes the base for API developers to build API lifecycle which enables API developers to manage API changes, and reduces migration effort.**

Keywords–API lifecycle; API lifecycle model; API change; API removal management; software migration.

I. INTRODUCTION

Software reuse offers many benefits, such as acceleration of software development and reduction of the overall development cost [1]. Reusable software provides common functionalities through application programming interfaces (APIs). Normally, APIs are assumed to be stable and serve as contracts between reusable software and client software. But, in fact, as reusable software evolves to meet changing requirements and solve emerging bugs over time, APIs will inevitably and frequently change [2]. These changes will cause client software to fail and increase maintenance cost [3]. Therefore, API changes must be managed.

API developers manage API changes to web service using tools, such as *API Manager* [4], *Lifecycle Manager* [5], and *Oracle API Management* [6]. Each tool contains API lifecycle, which is represented as a set of specific stages and the transitions between them. Different tools contain different API lifecycles. As a result, API developers need to choose the tool that contains the most suitable API lifecycle. After API lifecycle is chosen, API of the web service must follow the lifecycle from its birth (i.e., API addition) to its death (i.e., API removal). Throughout API lifecycle, API developers can control API addition, removal, and other changes.

Unlike API changes to web service, API changes to frameworks and libraries are not well managed [3][7][8]. From previous study, API changes happen frequently across different versions of a frameworks/libraries and commonly across different frameworks/libraries [9]. These API changes often cause a large amount of effort in migrating client application [10]-[12]. As a result, API users will complain through communication channels, such as online forums or mailing lists. If API developers do not handle those specific complaints, the framework or library that they have built will lose market share because API users will choose other frameworks or libraries instead.

To reduce migration effort, we first investigate the origin of API changes and then propose an effective way to manage them. According to previous research [3][9][10], most API changes occur when API developers *directly make them* in the design improvement tasks, without considering the affected client applications. As a result, the most effective way to manage API changes of frameworks and libraries is to *plan them* based on API lifecycle. API lifecycle can enable API developers to make API changes according to predefined stage and transitions. It also guarantees that API developers consider the affected client applications when making API changes. However, there is no one-size-fits-all API lifecycle for all API developers. Hence, we investigate the best API lifecycle model, instead of the best API lifecycle.

In this work, API lifecycle model is an abstraction of API lifecycles, and is represented as a set of general stages and transitions between them. To choose the best API lifecycle model, we first construct three API lifecycle models according to the observation on current practices, and then analyze those models from the perspective of API removal management. This perspective is chosen because API removal management enables API developers to prevent unintentional API removals and therefore avoid causing client applications to fail. We assess the impact of API lifecycle models on API developers and API users through three case studies. The best model is determined by the assessment results.

Main contributions of this work are as follows: 1) We are the first to provide API developers with a solution to manage API changes. 2) The best model is the base for API developers to build a suitable API lifecycle. Such API lifecycle enables API developers to manage API changes and therefore avoid causing client applications to fail. As a result, migration effort will be reduced. 3) We devise a metric set that enables API developers to assess the impact of their API lifecycle on both API developers and API users.

A preliminary analysis is performed in Section II to introduce API lifecycle models and assess the impact on

 ISBN: 978-1-61208-590-6

both API developers and API users. Case studies of models are conducted in Section III, and then results are presented in Section IV. The best model is determined according to the results. Related work is discussed in Section V. Finally, conclusion and future work are provided in Section VI.

II. Preliminary Analysis

The goal of this preliminary analysis is to approximately assess the impact of API lifecycle models on API developers and API users from the perspective of API removal management. Therefore, three generic API lifecycle models are introduced, which are shown in Figure 1. The support for API removal management is discussed, and the positive and negative impacts are further analyzed in detail. Finally, the analysis result is summarized to provide an overview of API lifecycle models.

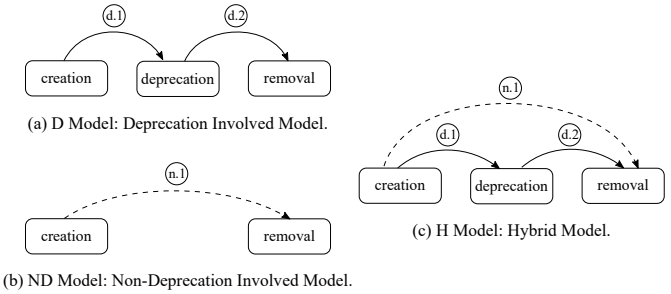

Figure 1. Three API Lifecycle Models.

A. D Model: Deprecation Involved Model

As shown in Figure 1(a), *Deprecation Involved Model* (*D Model*) contains three stages, including creation stage, deprecation stage, and removal stage. In D Model, API (e.g., A_D) enters creation stage when it is designed to provide functionality and exposed to API users. Through extensive usage, A_D might be found buggy or insufficient. Hence, API developers decide that A_D should be scheduled for removal because it is no longer recommended to use. To notify API users of the decision, API developers often label A_D 'deprecated' and provide the corresponding migration information (e.g., using other API instead) in API documents. After the notification, A_D enters deprecation stage from creation stage, and becomes a deprecated API. Meanwhile, API developers plan the future removal of the deprecated API. Once A_D is removed, it enters removal stage and ends its lifecycle.

D Model provides API developers full support for API removal management through the following ways. First, all API removals are planned before they occur. Therefore, API developers are able to control the timing of API removals. Second, migration problems caused by API removals (e.g., compilation errors occur when API users compile client applications and new version of frameworks/libraries) are mitigated through the provision of migration information. As a result, API developers have control over the impact of API removals on API users.

Although D Model provides full support for API removal management, it still has one negative impact on API developers. When maintaining deprecated APIs, API developers have to keep implementation of deprecated APIs in each new version of frameworks/libraries, and thus encounter the problem of code bloat. On the other hand, D Model has three positive impacts on API users. First, migration information enables API users to adapt client applications to API removals. Second, API users are informed of API removal schedules in advance through API documents, and thus they have sufficient time to adapt client applications. Third, the probability that API users have to adapt client applications is low because API developers often remove APIs only when necessary. This reduces the effort in migrating client applications.

B. ND Model: Non-Deprecation Involved Model

Figure 1(b) shows *Non-Deprecation Involved Model* (*ND Model*). ND Model contains creation and removal stages, but deprecation stage is excluded. As a result, the API following ND Model is always a non-deprecated API, and will never become a deprecated API. In ND Model, API (e.g., A_{ND}) enters creation stage when it is designed and exposed to API users. After A_{ND} is used, it might be found buggy or insufficient. Hence, the non-deprecated API, A_{ND}, will be directly removed in the new version of frameworks/libraries without planning. Such API is called *NR API* in this work, where *NR* stands for *Non-deprecated* and *Removed*. After A_{ND} is removed, it enters removal stage from creation stage. Such stage transition is denoted by the dotted arc in Figure 1(b).

ND model does not provide API developers with any support for API removal management. The reasons are listed as follows. First, all API removals occur without any planning. As a result, API developers are not able to control the timing of API removals. Second, migration problems will occur because of lacking of migration information in API document. API users need to find the alternative APIs by themselves to replace the removed APIs. Therefore, API developers do not have control over the impact of API removals on API users.

ND Model not only lacks the support for API removal management, but also has three negative impacts on API users. First, API users have difficulties in adapting client applications to API removals because no migration information is provided in API documents. Second, API users do not have sufficient time to adapt client applications because they are aware of API removals only after those API removals occur. Third, the probability that API users have to adapt client applications to API removals is high because such API removals occur without planning. Despite those negative impacts, ND Model has one positive impact on API developers: there is no deprecated API in ND Model, and thus API developers will not encounter the problem of code bloat.

C. H Model: Hybrid Model

As illustrated in Figure 1(c), *Hybrid Model* (*H Model*) is a hybrid of D Model and ND Model. In H Model, API in creation stage has two possible paths to removal stage. Path 1 includes two solid arcs in Figure 1(c), which is the same as D Model. Path 2 includes one dotted arc in Figure 1(c), which is the same as ND Model. The API which follows Path 1 becomes a deprecated API, and its removal is planned. The API which follows Path 2 becomes an NR API, and its removal is unplanned.

H Model provides partial support for API removal management. In H Model, both deprecated APIs and NR APIs exist. For deprecated APIs, API developers have control over the timing and impact of their removal. But, for NR APIs, API developers do not have any control. H Model has both

 ISBN: 978-1-61208-590-6

TABLE I. SUMMARY OF THE PRELIMINARY ANALYSIS RESULT.

Model	API removal management	Negative impacts	Positive impacts
D Model	Full support	1. Deprecated APIs cause API developers the problem of code bloat. (*cBloat*)	1. Migration information of deprecated APIs enables API users to adapt client applications to planned API removals. (*mInfo*) 2. Prior notification of planned API removals enables API users to have sufficient time to adapt client applications. (*aTime*) 3. The probability that API users have to adapt client applications to planned API removals is low. (*aPro*)
ND Model	No support	1. No migration information of NR APIs hinders API users from adapting client applications to unplanned API removals. (*mInfo*) 2. API users do not have sufficient time to adapt client applications due to the lack of prior notification of unplanned API removals. (*aTime*) 3. The probability that API users have to adapt client applications to NR APIs is high. (*aPro*)	1. NR APIs do not cause API developers the problem of code bloat. (*cBloat*)
H Model	Partial support	1. No migration information of NR APIs hinders API users from adapting client applications to API removals. (*mInfo*) 2. For unplanned API removals, the lack of their prior notification causes API users the problem of not having sufficient time to adapt client applications. (*aTime*) 3. For unplanned API removals, the probability that API users have to adapt client applications is high. (*aPro*) 4. Deprecated APIs cause API developers the problem of code bloat. (*cBloat*)	1. Migration information of deprecated APIs enables API users to adapt client applications to planned API removals. (*mInfo*) 2. For planned API removals, their prior notification enables API users to have sufficient time to adapt client applications. (*aTime*) 3. For planned API removals, the probability that API users have to adapt client applications is low. (*aPro*) 4. NR APIs do not cause API developers the problem of code bloat. (*cBloat*)

Note 1: The words in the parentheses are the abbreviations of impacts.
Note 2: The negative and positive impacts of H Model which are inherited from D Model are highlighted with the gray background, while those which are inherited from ND Model are shown with the white background.
Note 3: Planned API removals are the removals of deprecated APIs, while unplanned API removals are the removals of NR APIs.

positive and negative impacts. Some of them are inherited from D Model, and the others are inherited from ND Model. The whole list of those impacts is provided in Table I, which is introduced in the next section.

D. Summary

To have an overview of those API models, we summarize the analysis result in Table I. The analysis result reveals the following three types of information on the models: 1) degree of support for API removal management, 2) negative impacts, and 3) positive impacts. Furthermore, the impact categories, which are shown as index words in Table I, are described as follows:

1) *Code bloat (cBloat)*. It includes the impacts related to the problem of code bloat.
2) *Migration information (mInfo)*. It includes the impacts related to the provision of migration information.
3) *Adaptation time (aTime)*. It includes the impacts related to the time which API users have for software adaptation.
4) *Adaptation probability (aPro)*. It includes the impacts related to the probability that API users have to adapt client applications to planned or unplanned API removals.

III. CASE STUDIES

The goal of the case studies is to assess the positive and negative impacts of the models in four impact categories. Through the assessment, the best model will be concluded if it outperforms the others in the most impact categories.

A. Research Questions

To conclude the best model, we have to answer the following research questions:

1) RQ1: Regarding the impact category of code bloat, which model performs the best?
2) RQ2: Regarding the impact category of migration information, which model performs the best?
3) RQ3: Regarding the impact category of adaptation time, which model performs the best?
4) RQ4: Regarding the impact category of adaptation probability, which model performs the best?

To answer these research questions, we assess impacts of the models in four impact categories through a set of metrics. As shown in Table II, those metrics are organized according to targeted impact categories, and definitions are also provided. With these metrics, we can assess the impacts and answer the research questions.

B. Data Collection

Three subjects are chosen for data collection, and each one is the representative of a specific API lifecycle model. Those subjects are all medium-scaled open source projects with hundreds of Java classes. The duration of data collection for each subject is approximately three years. The subjects and the collected data are introduced as follows.

Subject for D Model: JFace with versions from 3.6 to 4.3. JFace is a popular user interface framework on Eclipse platform. The duration of the data collection is from June 2010 to June 2013. The collected data includes API documents and source code. From API documents, the deprecated APIs are extracted to measure the values of $PC_{d/all}$ and T_d. Besides, migration information for the deprecated APIs is investigated to enable the measurement of $PC_{dMI/d}$. The value of PB_p is measured through extracting APIs from API documents, detecting planned API removals, and confirming the occurrence of those planned API removals in the source code.

Subject for ND Model: JFreeChart with versions from 0.9.4 to 1.0.0. JFreeChart is a mature and widely used chart library. According to download statistics, it has been downloaded for more than three million times since its registration in SourceForge. The duration of the data collection is from October 2002 to December 2005. During data collection,

TABLE II. DEFINITION OF METRICS.

Targeted impact category	Metric	Definition
Code bloat	$PC_{d/all}$	In all APIs, the percentage of deprecated APIs.
	$PC_{nr/all}$	In all APIs, the percentage of NR APIs.
Migration information	$PC_{dMI/d}$	In all deprecated APIs, the percentage of deprecated APIs with migration information.
	$PC_{nrMI/nr}$	In all NR APIs, the percentage of NR APIs with migration information.
Adaptation time	T_d	The average time, measured in months, of a deprecated API from being announced to-be-removed to being removed. (i.e., the average time of a deprecated API staying in the deprecation stage)
	T_{nr}	The average time, measured in months, of an NR API from being announced to-be-removed to being removed.
Adaptation probability	PB_p	The probability of the occurrence of planned API removals.
	PB_u	The probability of the occurrence of unplanned API removals.

TABLE III. ASSESSMENT RESULTS FOR THE CODE BLOAT CATEGORY.

Model	$PC_{d/all}$, the metric for assessing negative impacts				$PC_{nr/all}$, the metric for assessing positive impacts			
	package	class	method	field	package	class	method	field
D	0.00%	4.01%	1.43%	2.69%	–	–	–	–
ND	–	–	–	–	6.81%	12.28%	13.19%	14.57%
H	0.00%	2.60%	1.81%	1.84%	0.00%	0.00%	0.02%	0.00%

Note 1: The label "–" means that the value of the metric is not available because the negative or positive impact of the corresponding model does not exist.
Note 2: The model which performs the best is highlighted with the gray background.

TABLE IV. ASSESSMENT RESULTS FOR THE MIGRATION INFORMATION CATEGORY.

Model	$PC_{nrMI/nr}$, the metric for assessing negative impacts				$PC_{dMI/d}$, the metric for assessing positive impacts			
	package	class	method	field	package	class	method	field
D	–	–	–	–	$*$	87.10%	86.39%	58.97%
ND	0.00%	0.52%	0.04%	0.00%	–	–	–	–
H	$*$	$*$	0.00%	$*$	$*$	88.48%	92.87%	71.56%

Note 1: The label "–" means that the value of the metric is not available because the negative or positive impact of the corresponding model does not exist.
Note 2: The label "$*$" means that the value of the metric is not available because of division by zero.
Note 3: The model which performs the best is highlighted with the gray background.

API documents are not provided. As a result, the collected data includes the source code, online forum, and jfreechart-commit mailing list, but no API documents. From the source code, packages, classes, methods, and fields with public or protected access modifiers are extracted as APIs. NR APIs and their removals are then detected to measure the values of $PC_{nr/all}$ and PB_u. In the online forum, migration information is identified to measure the value of $PC_{nrMI/nr}$. In jfreechart-commit mailing list, notification of the removals of NR APIs is also identified to measure the value of T_{nr}.

Subject for H Model: JFreeChart with versions from 1.0.2 to 1.0.13. The duration of the data collection is from August 2006 to April 2009. API documents are provided during data collection. Therefore, the collected data includes API documents, source code, online forum, and jfreechart-commit mailing list. In API documents, the deprecated APIs and their migration information are investigated to measure the values of $PC_{d/all}$, T_d, and $PC_{dMI/d}$. From API documents and source code, NR APIs, planned API removals, and unplanned API removals are detected to measure the values of $PC_{nr/all}$, PB_p, and PB_u. In the online forum, migration information of NR APIs is identified to measure the value of $PC_{nrMI/nr}$. In jfreechart-commit mailing list, notification of the removals of NR APIs is identified to measure the value of T_{nr}.

IV. RESULTS OF CASE STUDIES

In Section IV-A, results of case studies are presented to answer the four research questions. In Section IV-B, answers to those research questions are summarized to conclude the best model. Then, the cost-effectiveness of the best model is also discussed.

A. Answers to Research Questions

RQ1: Assessment of Impacts in the Code Bloat Category. The assessment results are summarized in Table III. The values of $PC_{d/all}$ show that little of APIs are deprecated APIs. Thus, the problem of code bloat caused by deprecated APIs in D Model is insignificant. On the other hand, NR APIs in ND Model not only avoid the problem of code bloat for API developers, but also reduce API developers' effort in maintaining API implementation by 6.81% to 14.57%. Regard to H Model, the values of $PC_{d/all}$ are between two models, but the values of $PC_{nr/all}$ are very low. So, the negative impact is medium but the positive is insignificant. In summary, ND Model is the best model because of the following two reasons. First, it only has the positive impact in the code bloat category. Second, it has the largest $PC_{nr/all}$ values, which means its positive impact is the most significant. As a result, ND Model performs the best regarding the code bloat category.

RQ2: Assessment of Impacts in the Migration Information Category. Table IV summarizes the assessment results. Both D Model and H Model have significant positive impacts with large $PC_{dMI/d}$ values. But, D Model is better than H Model because D Model has only positive impact. Unlike D Model, H Model has not only positive impact but also negative impact. Regarding the positive one, H Model has slightly larger positive impact than D Model. Regarding the negative one, the $PC_{nrMI/all}$ value of H Model reveals that none of NR APIs in the method level are provided with migration information. Unfortunately, lack of migration information hinders API users from migrating client applications. Two discussion topics in the online forum of JFreeChart have been found as the evidence. The first one is "Arrrgh! Lots of API changes again", in which API users state that "*..., these API changes really are not funny! It takes a lot of time, researching, looking in the new JFreeChart source code because there is no migration description.*" [13]. The second one is "API changes: undocumented (again)", in which API users state that "*Could documentation be improved and more information be provided to ease migration?*" [14]. With this evidence, it is concluded that the negative impact of H Model is significant enough to offset the positive impact of H Model. In summary, answer to RQ2 is that D Model performs the best regarding the migration information category.

RQ3: Assessment of Impacts in the Adaptation Time

TABLE V. ASSESSMENT RESULTS FOR THE ADAPTATION TIME CATEGORY.

Model	T_{nr}, the metric for assessing negative impacts	T_d, the metric for assessing positive impacts
D	–	24.01
ND	0.40	–
H	1.75	14.19

Note 1: The label "–" means that the value of the metric is not available because the negative or positive impact of the corresponding model does not exist.
Note 2: The model which performs the best is highlighted with the gray background.

TABLE VI. ASSESSMENT RESULTS FOR THE ADAPTATION PROBABILITY CATEGORY.

Model	PB_u, the metric for assessing negative impacts				PB_p, the metric for assessing positive impacts			
	package	class	method	field	package	class	method	field
D	–	–	–	–	0.00	0.00	0.00	0.00
ND	0.39	0.94	1.00	0.83	–	–	–	–
H	0.00	0.00	0.64	0.00	0.00	0.00	0.00	0.00

Note 1: The label "–" means that the value of the metric is not available because the negative or positive impact of the corresponding model does not exist.
Note 2: The model which performs the best is highlighted with the gray background.

Category. The positive and negative impact are assessed by T_d and T_{nr}. However, the subjects for D Model and H model do not contain the deprecated APIs which are removed. To measure the values, we assume that deprecated APIs in the subjects are finally removed in the last version. Because of the assumption, actual T_d values must larger than the measured T_d values. The assessment results are summarized in Table V. Compared with T_d values, values of T_{nr} are very small. It means that API users have only a short time to adapt client applications to unplanned API removals. Hence, the negative impact of ND Model and H Model is significant in those cases. Based on the assessment results, D Model is the best model because it has the most significant positive impact with the largest T_d value. Besides, D Model does not have negative impact. As a result, the answer to RQ3 is that D Model performs the best regarding the adaptation time category.

RQ4: Assessment of Impacts in the Adaptation Probability Category. Table VI summarizes the assessment results. Because deprecated APIs are not removed in our subjects, values of PB_p are all zero. It means the positive impact on API users is significant for D Model and H Model. On the other hand, the high values of PB_u for ND Model indicate that API users must adapt client applications to unplanned API removals. So, the negative impact of ND Model is significant. As a result, D Model is the best model because it has only positive impact and such positive impact is significant. Although H Model has positive impact with the same significance, it has some of negative impact assessed by PB_u. Hence, D Model is still better than H Model. In summary, the answer to RQ4 is that D Model performs the best regarding the adaptation probability category.

B. The Best Model

The answer to RQ1 is ND Model, while the answers to RQ2, RQ3, and RQ4 are D Model. As a result, ND Model outperforms the others only in the code bloat category, while D Model performs the best in the migration information, adaptation time, and adaptation probability categories. According to the summary, D Model is the best model because it outperforms the others in the most of categories.

The advantages of D Model are presented in Section II-A. According to four impact categories, the following discussion is to demonstrate that the cost of getting the advantage is low.

Code bloat category. The cost of maintaining deprecated APIs is low. As discussed the answer of RQ1 in Section IV-A, very few APIs in D Model are deprecated APIs. Hence, the cost of maintaining deprecated APIs is low.

Migration information category. The cost of providing migration information is low. Migration information is naturally derived because deprecated APIs are often planned to be replaced with new APIs. Besides, adding migration information to API documents requires little cost. As a result, the cost of providing migration information is low.

Adaptation time category. The cost of prior notification is low. The notification of API removals is often performed through 1) labelling an API 'deprecated' in API documents and 2) publishing updated API documents to API users. Because both of such costs are low, the cost of prior notification is low.

Adaptation probability category. The cost of planning API changes is low. Planning API removals requires designing new APIs and scheduling API removals. The former is the integral part of API design, which does not cause any additional cost. The latter needs little cost because API developers have to consider API removals for software release. Hence, the total cost of planning API changes is low.

V. RELATED WORK

Support for API removal adaptation. Chow and Notkin [15] proposed an approach for semi-automatic adaptation to API removals in libraries. Their approach required library developers to annotate API changes with a specific language, and the annotation was used by API users for API removal adaptation. The drawback of this approach was that library developers had to learn a new language. Perkins [16] developed a technique to replace calls to deprecated API methods with their method bodies. The assumption of the technique was that the method bodies contained appropriate replacement code. Many approaches [17][18] were developed to support API removal adaptation by recording and replaying refactorings with refactoring engines. In those approaches, API developers and API users were required to utilize the same refactoring engines so that recorded refactorings could be replayed by API users. An alternative solution to API removal adaptation was to develop matching techniques [2][19]-[21] for discovering replacement APIs, by which deprecated or removed APIs were replaced. Some of the techniques directly performed replacement without API users involvement, and thus the appropriateness of discovered replacement APIs was not guaranteed. On the contrary, the others provided a set of replacement API candidates from which API users could choose. But, API users had to spend additional effort in guaranteeing the appropriateness.

Although many approaches and techniques are developed to help API users with API removal adaptation, they all have limitations. Recently, API deprecation is a promising solution for adapting API changes. An empirical study of Hora et al. [10] indicate that the deprecation mechanism should be adopted. The study shows that API deprecation reaction is faster and larger compared with NR API reaction. Besides, the study of McDonnel et al. [22] indicate that client

applications need a longer adaptation time when APIs are evolving fast. Such adaptation time can be preserved through deprecation because it signals API users that which API ought to be avoided [23]. In addition, Brito et al. [24] argue that replacement messages with deprecated APIs facilitate API users to adapt APIs. This argument complies with the study of Ko et al. [25], which empirically indicates that migration information promotes the reaction to API evolution. However, all these previous studies focus on why API developers should adopt deprecation. How to apply deprecation has not been investigated. Therefore, we discover the best model embedded with deprecation to present how deprecation can be applied to API removal management. In the best model, API removals are planned in advance and early announced to API users. Moreover, API developers provide migration information, which contains replacement APIs. As a result, the appropriateness of replacement APIs is guaranteed because of the credible information source. In summary, the best model ensures the support for API removal adaptation.

VI. Conclusion and Future Work

The best model for API removal management is presented in this work. The characteristics of the best model include 1) planning of API removals, which prevents unintentional API removals and makes APIs stable, 2) provision of migration information, which reduces migration effort, and 3) prior notification of planned API removals, which preserves sufficient time to adapt client applications. The goal behind the best model is to benefit both API developers and API users, who are the major stakeholders in the ecosystem formed by frameworks/libraries and client applications. As a result, following the best model will make API developers design more stable APIs with planning, and API users will spend less effort in constructing and maintaining client applications.

While we conclude D Model is the best in two popular, mature, and open source systems of Eclipse, the selected subjects might not be representative in other domains, such as web framework. Web framework is widely adopted in different ways to build kinds of web apps, and is changing at an extremely rapid pace right now. For the purpose of preserving market share, web framework developers are forced to evolve the framework in time to catch up with the trend. API removals will happen more frequently compared with those observed subjects in this study. Therefore, to investigate the best API lifecycle model further within such context will be our future work.

References

[1] I. Sommerville, *Software Engineering*. Pearson Education Limited, 2010, vol. Ninth Edition ed.

[2] Z. Xing and E. Stroulia, "API-Evolution Support with Diff-CatchUp," *Journal of IEEE Transactions on Software Engineering*, vol. 33, no. 12, Dec. 2007, pp. 818–836.

[3] D. Dig and R. Johnson, "The Role of Refactorings in API Evolution," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM), 26–29 Sept. 2005, Budapest, Hungary*, 2005.

[4] "About API Manager," 2017, URL: https://docs.wso2.com/display /AM110/About+API+Manager [accessed: 2017-08-22].

[5] "Lifecycle Manager for APIs," 2015, URL: https://www.roguewave.com /products-services/akana/lifecycle-manager [accessed: 2017-08-22].

[6] "Oracle API Management," 2015, URL: http://www.oracle.com/us/ products/middleware/soa/api-management/overview/index.html [accessed: 2017-08-22].

[7] D. Dig and R. Johnson, "How do APIs Evolve? A Story of Refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, Mar. 2006, pp. 83–107.

[8] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Journal of Automated Software Engineering*, vol. 14, June 2007, pp. 215–259.

[9] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and Impact Analysis of API Breaking Changes: A Large-scale Study," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER), 20–24 Feb. 2017, Klagenfurt, Austria*, 2017, pp. 138–147.

[10] A. Hora et al., "How Do Developers React to API Evolution? A Large-scale Empirical Study," *Journal of Software Quality Journal*, 2016, pp. 1–31.

[11] W. Wu, F. Khomh, B. Adams, Y. G. Guhneuc, and G. Antoniol, "An Exploratory Study of API Changes and Usages based on Apache and Eclipse Ecosystems," *Journal of Empirical Software Engineering*, vol. 21, no. 6, Dec. 2016, pp. 2366–2412.

[12] G. Bavota et al., "The Impact of API Change and Fault-Proneness on The User Ratings of Android Apps," *Journal of IEEE Transactions on Software Engineering*, vol. 41, no. 4, Apr. 2015, pp. 384–407.

[13] "Arrrgh! Lots of API changes again!" 2003, URL: http://www.jfree.org /forum/viewtopic.php?f=3&t=5093 [accessed: 2017-08-22].

[14] "API changes: undocumented (again)," 2004, URL: http://www.jfree.org /forum/viewtopic.php?f=3&t=9265 [accessed: 2017-08-22].

[15] K. Chow and D. Notkin, "Semi-automatic Update of Applications in Response to Library Changes," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 4–8 Nov. 1996, Monterey, CA, USA*, 1996, pp. 359–368.

[16] J. H. Perkins, "Automatically Generating Refactorings to Support API Evolution," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE), 5–6 Sept. 2005, Lisbon, Portugal*, 2005, pp. 111–114.

[17] J. Henkel and A. Diwan, "CatchUp! Capturing and Replaying Refactorings to Support API Evolution," in *Proceedings of 27th International Conference on Software Engineering (ICSE), 15–21 May 2005, St. Louis, MO, USA*, 2005, pp. 274–283.

[18] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries," in *Proceedings of 30th International Conference on Software Engineering (ICSE), 10–18 May 2008, Leipzig, Germany*, 2008, pp. 441–450.

[19] M. Kim, D. Notkin, and D. Grossman, "Automatic Inference of Structural Changes for Matching across Program Versions," in *Proceedings of the 29th international conference on Software Engineering (ICSE), 20–26 May 2007, Minneapolis, MN, USA*, 2007, pp. 333–343.

[20] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "AURA: A Hybrid Approach to Identify Framework Evolution," in *Proceedings of 32nd ACM/IEEE International Conference on Software Engineering (ICSE), 1–8 May 2010, Cape Town, South Africa*, 2010, pp. 325–334.

[21] Z. Xing and E. Stroulia, "Identifying and Summarizing Systematic Code Changes via Rule Inference," *Journal of IEEE Transactions on Software Engineering*, vol. 39, no. 1, Jan. 2013, pp. 45–62.

[22] T. McDonnell, B. Ray, and M. Kim, "An Empirical Study of API Stability and Adoption in the Android Ecosystem," in *Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM), 22–28 Sept. 2013, Eindhoven, Netherlands*, 2013, pp. 70–79.

[23] J. Zhou and R. J. Walker, "API Deprecation: a Retrospective Analysis and Detection Method for Code Examples on the Web," in *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 13–18 Nov., 2016, Seattle, WA, USA*, 2016, pp. 266–277.

[24] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems," in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 14–18 Mar. 2016, Suita, Japan*, 2016, pp. 360–369.

[25] D. Ko et al., "API Document Quality for Resolving Deprecated APIs," in *Proceedings of 21st IEEE Asia-Pacific Software Engineering Conference (APSEC), 1–4 Dec., 2014, Jeju, South Korea*, 2014, pp. 27–30.

 ISBN: 978-1-61208-590-6