How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community

Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, Arnaud Thiefaine

▶ **To cite this version:**

Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, Arnaud Thiefaine. How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community. 29th Asia-Pacific Software Engineering Conference (APSEC 2022), Dec 2022, Virtual, Japan. hal-03853493

HAL Id: hal-03853493
https://hal.science/hal-03853493

Submitted on 15 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community

Oleksandr Zaitsev*†, Stéphane Ducasse†, Nicolas Anquetil†, and Arnaud Thiefaine*
*Arolla, Paris, France
{oleksandr.zaitsev,arnaud.thiefaine}@arolla.fr
†Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
{stephane.ducasse,nicolas.anquetil}@inria.fr

***Abstract*—The evolution of software libraries is a process that requires a joint effort of two groups of developers: the *library developers* who prepare the release and *client developers* who need to update their applications to the new versions. To build better tools that support both library and client developers throughout the evolution, we need to understand what problems they face and how they react to those problems. In this paper, we present the result of two surveys: one for library developers and one for client developers. Our surveys involved developers from two industrial companies and an open-source community. We assess (1) how they perceive the impact of library evolution and (2) what is the support that library developers can provide to their clients. By approaching those questions from the perspectives of library and client developers, we try to assess how challenging library update is for each of those groups and how motivated they are to overcome those challenges.**

I. Introduction

Modern software depends on many reusable components such as libraries, frameworks, microservices[1], etc. [1]. For simplicity, in this work, we will refer to all reusable components as *libraries* and all software that depends on them as *client software*. Like any other software, libraries evolve [2], [3], [4] and release new versions. As a result, client software that depends on those libraries must also change accordingly to be compatible with the new versions. This process is called the *library update*[2], it involves two groups of developers: those who maintain the libraries (we call them *library developers*) and those who depend on them (we call them *client developers*). The efficient library evolution requires the joint effort of both groups: the effort of client developers to update their dependencies to the latest versions and the effort of library developers to make the process of update simple and clear.

In recent years, many approaches were proposed to support developers in the task of library update by building automated tools. Some of those approaches are designed for client developers: they allow them to infer the modifications that need to be applied to their systems either by calculating the textual and structural similarity between the source code of the two versions of the library [5], [6] or by mining frequent method call replacements either in library itself [7], [8] or in other client systems that were already updated [9], [10]. Other approaches help library developers support their clients by annotating changed functions with transformation rules [11], [12] or recording the refactorings that were performed on the library to later "replay" them on client systems [13].

To build better tools for library update and efficiently automate parts of this process, it is important to understand the behavior of both library and client developers, the practices that they adopt, the problems that they encounter. Several empirical studies were performed, based either on the analysis of source code [14], [15], [16], [17], [18], [19], [20], [21] or surveying the developers [17], [22], [23], [24]. However, those studies either focus only on one group of developers (library or client) or ask questions about specific breaking changes found in the code. Also, to the best of our knowledge, no previous study focused on the type of help that library developers can provide to their clients.

In this work, we present the results of two surveys: one of library developers and another one of client developers. Participants came from different backgrounds (open-source or industry) and different projects, the questions were general and not related to specific libraries or issues. The focus of our study is the perception of the impact of library evolution from different perspectives and the type of support that library developers can provide to help their clients. The main contributions of this work are: (1) we perform two surveys: of library and client developers from different communities; (2) we are the first ones to ask client developers about the type of support that they need; (3) we confirm the results of previous studies [21], [23], [24] on the motivation of library developers to introduce breaking changes and their perception of impact.

The rest of this paper is structured as follows: In Section II, we discuss the state of the art. In Section III, we discuss the methodology of this study and list the research questions. In Section IV, we describe the population of developers who were selected for this survey. In the following two sections, we present the results from both of our surveys: first the library developer survey in Section V and then the client developer survey in Section VI. Finally, in Section VIII we present the conclusions.

[1]Microservice forms a library of reusable code of sorts, with the REST API being analogous to the API of public methods of a library.

[2]Not to be confused with library migration — the process of changing a dependency from one library to a *different* library.

TABLE I: Empirical studies of library evolution characterized by two criteria: is it based on a developer survey, on source code analysys, or both; does it explore the client developer perspective, the library developer perspective, or both

Paper			Method	Perspective
Robbes *et al.*	2012	[19]	code an.	client dev.
Jezek *et al.*	2015	[16]	code an.	both
Hora *et al.*	2015	[14]	code an.	client dev.
Bogart *et al.*	2016	[22]	dev. survey	both
Sawant *et al.*	2016	[20]	code an.	client dev.
Xavier *et al.*	2017	[21]	code an.	client dev.
Xavier *et al.*	2017	[24]	dev. survey	library dev.
Hora *et al.*	2018	[15]	code an.	client dev.
Kula *et al.*	2018	[17]	both	client dev.
Kula *et al.*	2018	[18]	code an.	library dev.
Brito *et al.*	2019	[23]	dev. survey	both
Our study			**dev. survey**	**both**

II. State of the Art

We start by discussing the existing empirical studies on library evolution and discuss their shortcomings. In Table I, we categorise the related studies using two criteria:

- *Developer survey or code analysis?* In literature, there are two main techniques for studying library evolution and its effect on clients:
  1) Analysing the source code of the evolving libraries and/or client systems; i.e., *"How do libraries change, and how do those changes propagate?"*
  2) Surveying the developers to explore the human side of library evolution; i.e. *"How do developers perceive this process?"*.
- *Client developer or library developer perspective?* Library evolution can be explored from the perspective of library developers or from the perspective of their clients. Some studies (like this one) explore both sides.

a) Studies that analyse source code: There have been multiple studies of the ripple effect caused by breaking changes and how it propagates through client systems. Robbes *et al.,* [19] studied the reaction of clients in Pharo ecosystem to the deprecation of API elements in their dependencies. They conclude that many clients do not react to library deprecations and when they do react, they often do not apply adaptations to the entire project at once, leaving it inconsistent. Sawant *et al.,* [20] performed a partial replication of this study on Java projects, considering almost 10 times more client systems. They arrive to the same conclusions as the ones derived from Pharo ecosystem. Hora *et al.,* [14], [15] extended the study of Robbes *et al.,* [19] by exploring the changes in non-deprecated API elements. They report that half of the analysed API changes are missed deprecation opportunities and suggest that recommender tools can be built to help library developers introduce those deprecations. They also observe that most API changes can be implemented as rules and suggest that those rules can be used to help client developers automatically update their code. Jezek *et al.,* [16] studied the evolution of Java libraries both from library and client perspectives. They report that breaking changes are very frequent (80% of version updates break compatibility), however, this causes few actual problems in real client systems. Xavier *et al.,* [21] performed a large-scale analysis of Java libraries and their clients to understand how frequent are breaking changes, how do they evolve over time, and how do they impact the clients. They discovered that on the median, 15% of all changes in a library are breaking changes and their frequency increases over time. However, they also report that most breaking changes do not have big impact on client systems and explain this with a hypothesis that library developers try not to break highly impactful API elements. Kula *et al.,* [17], [18] performed two studies: one on the client side and another one on the library side. In their first study they report that most client systems rarely update their libraries and 81.5% of client systems choose to remain with the older popular version of the library. In their second study, Kula *et al.,* [18] explored the evolution of libraries and found that while many breaking changes happen in non client-used API and non API classes, the client-used API are less likely to be broken.

b) Developer surveys: Bogart *et al.,* [22] performed a case study by interviewing developers of three software ecosystems to understand how developers make decisions about changes and document the practices that are used in those communities. In the same study where Kula *et al.,* [17] discovered that most client systems have outdated dependencies, they performed a survey of client developers and found that 69% of them are unaware of their vulnerable dependency. They also report that developers are reluctant to update because they perceive it as extra workload. Xavier *et al.,* [24] performed a survey of seven core developers from popular Java libraries. Based on this survey, they propose a list of five reasons that motivate breaking changes: library simplification, refactoring, bug fix, dependency changes, and project policy. They report that all surveyed developers are aware of the effect of breaking changes on clients. In their follow-up study, Brito *et al.,* [23] conducted a larger survey of 56 library developers through a firehouse interview (*i.e.,* contacting developers immediately when a breaking change is detected). According to their study, the most common reason for introducing breaking changes are the need to implement new features (32%), to simplify the API (29%), and to improve maintainability (24%). In the same study, Brito *et al.,* [23] also explored the actual impact of breaking changes on clients by analysing the questions published on StackOverflow. They report that 45% of all posts related to breaking changes are client developers asking how to overcome the negative effect of those changes on their code.

c) Shortcomings of the existing studies: Most surveys that we discussed above either targeted specific developers and asked them about breaking changes or vulnerable dependencies that were detected by authors [17], [23], [24], or analysed the questions related to breaking changes that were published on StackOverflow [23]. To the best of our knowledge, Bogart *et al.,* [22] were the only survey that

contacted developers of different projects and asked them general questions about the process of library evolution and the practices that they use both from the perspective of library and client developers (because most library developers are also clients of other libraries). However, that study still mostly focuses on library developer perspective. Also, no survey asked client developers about what makes the process of library update hard and what support do they expect from library developers to make this process easier.

III. Survey Setup

To address the shortcomings of the previous studies, we propose to conduct a survey of two groups: library and client developers. Each group should include developers from different projects and different backgrounds (open-source and industry). The survey must ask general questions about the experiences and practices of developers and not be restricted to specific cases of breaking changes.

In this work, we answer the following research questions:

a) Library Developer Survey:

- RQ.1 How do library developers perceive the impact of library evolution on their clients?
- RQ.2 Why do library developers introduce breaking changes?
- RQ.3 How motivated are library developers to support their clients?
- RQ.4 How do library developers help their clients to update?

b) Client Developer Survey:

- RQ.5 How do client developers perceive impact of library evolution on their systems?
- RQ.6 What makes library update easy and what makes it hard?

Considering the relatively small population size, we do not perform statistical tests and derive conclusions from simple descriptive statistics.

IV. Describing the Population

We conducted a survey of developers from two industrial companies (Arolla and Berger-Levrault) and Pharo open-source community. Both Arolla and Berger-Levrault are medium size software companies. The key difference is that Arolla is a consulting company where each developer works on different project and Berger-Levrault is an international software company with multiple teams where developers work on their projects together. We selected those companies for convenience: we are part of the Pharo community and by this have a privilege access to its members. We also had privilege access to some developers in both companies that participated in the survey.

a) Developers selection: For Pharo and Arolla, we sent emails to the mailing lists and messages to the official online forums of each community. For Berger Levrault, we contacted the director of research and asked him to spread the survey among his colleagues. In all three cases, participation was optional and all participants were informed that the survey would be anonymous. We asked every software developer (no matter what type of programming they do) to fill the client developer survey. As for the library developer survey, we explicitly asked it to be filled only by those developers who work on libraries, frameworks, microservices, or any tools that have API, several versions, and some users. Developers were free to answer both surveys. In general there is no matching between the libraries that client developers use and the ones that library developers maintain.

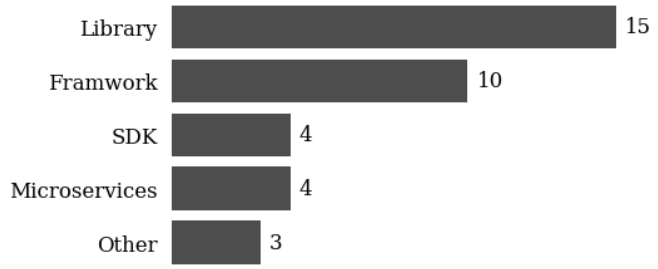

Fig. 1: The types of software developed by the library developers who took part in our study.

b) Library Developers: We received the answers of 18 library developers. 11 of them belong to Pharo open-source community, 4 are from Arolla, and one from Berger-Levrault.[3] Two participants decided not to specify the community to which they beong. We asked developers to evaluate their level of expertise as either *Absolute Novice* (0 dev.), *Beginner* (0 dev.), *Intermediate* (3 dev., 18%), *Advanced* (9 dev., 53%), or *Expert* (5 dev., 29%). Twelve developers work on open-source projects, 5 developers work on closed-source projects, and one developer works on both. We asked developers to specify the kind of software that they develop. Their answers can be seen in Figure 1. We also asked library developers to specify approximately how many client developers (users) they have. The estimates scattered among the following options: 1 to 10 clients (6 dev., 33%), 11 to 50 clients (5 dev., 28%), 51 to 100 clients (3 dev., 7%), 101 to 500 clients (4 dev., 22%). No library developer in our survey claimed to have more than 500 clients. As will be discussed in Section VII, we acknowledge that the relatively small size of libraries developed by the participants of our survey may pose a threat to external validity. But we also consider this to be a novelty because the surveys that only focus on largest or most popular open-source libraries are often biased by their size.

c) Client Developers: We received 37 answers from client developers, 22 of which were from Pharo community, 5 from Arolla, and 4 from Berger Levrault. Also, 4 developers preferred not to specify their affiliation, and other 2 developers listed other open-source communities that they belong to.

[3]It should be noted that getting library developers in companies is rarer than library users. This proportion is different in the context of open-source because many open-source projects are proposing libraries to other developers (whose code may often be closed source).

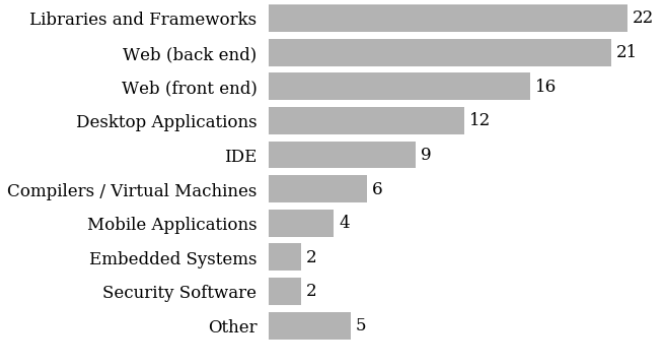

Fig. 2: The types of software developed by the client developers who took part in our study.

As can be seen in Figure 2, over selection covers a variety of domains by the types of projects that client developers work on. They range from library, frameworks, web, desktop applications but also tool development and even compiler ones. As we mentioned before, the developers in our study were free to participate in both surveys. This and the fact that most respondents come from the open-source community, may explain that 22 out of 37 client developers also develop libraries. We further discuss this in Section VII. We asked client developers to evaluate their level of expertise as either *Absolute Novice* (0 dev.), *Beginner* (2 dev., 6%), *Intermediate* (6 dev., 18%), *Advanced* (18 dev., 53%), or *Expert* (8 dev., 24%). As with library developers, we asked client developers to specify the kind of software that they develop. Their answers can be seen in Figure 2. We also asked client developers to specify approximately how many dependencies do they have. The most popular answers were 1 to 10 (15 dev., 42%) and 11 to 50 (16 dev., 44%). 4 developers said that they have between 51 and 100 dependencies and one developer claimed to have more than 100 dependencies.

V. Results of the Library Developer Survey

Now we present the results of the survey following the research questions mentioned before. To answer research questions that are rather general, we had to ask specific questions in our questionaires. That is why, each *research question* in our study is associated with one or more *survey questions*. In this section, we discuss the results of a library developer survey and in Section VI, we will present the results of a survey of client developer.

A. RQ.1 - How do library developers perceive the impact of library evolution on their clients?

To answer this research question, we asked two questions in our survey: one to assess their perception of the impact of breaking changes on client systems and one to understand how they estimate the time that clients need to update. The answers are presented in Figures 3 and 4.

As can be seen in the Figures, both questions were answered by all 18 library developers who took part in this survey. 9

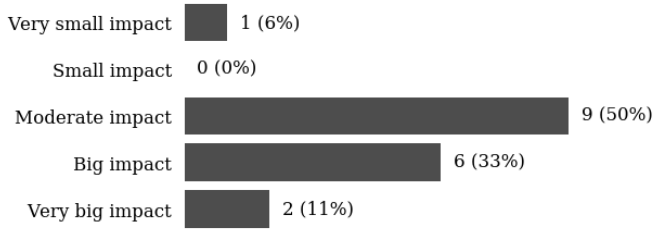

Fig. 3: Do you think that breaking changes in your releases have big impact on clients?

of them believe that the impact of breaking changes on their clients is *moderate*, 6 (33%) that it is *big*, and 2 (11%) that it is *very big*. One developer (6%) assessed the impact as *very small*.

As for the time that is needed to update the client systems to the new versions of their libraries, opinions of library developers are different. When presented with a five-option scale between *less than an hour* and *a week or more*, the estimates scattered among them with *less than an hour* being the most popular option, selected by 7 developers (39%), and *a week or more* being the least popular one, selected by one developer (6%). We hypothesise that different answers to this question are caused by different complexity of changes in library releases. Some library updates are simple and can be done in less than an hour. Others are hard and may require a week or more.

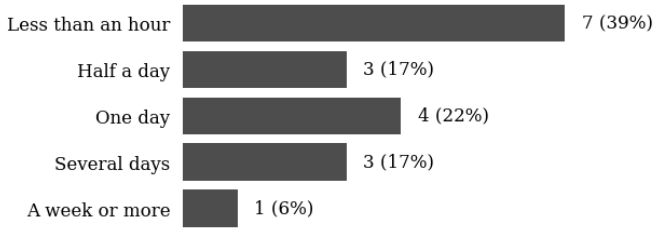

Fig. 4: How much time do you think client developers need to update to the new version of your library?

The answers to the second question about the time to update could have been influenced by *Deprewriter* [12] — the automatic deprecation update mechanism that is used by the Pharo community. It allows library developers to annotate method deprecations with transformation rules that will be used to automatically rewrite the call-sites in the client code. This mechanism considerably eases the update of deprecated methods. Since most of the library developers answering the survey were members of the Pharo community, this is a factor to take into account.

Summary:

- Most library developers in our survey agree that the impact of breaking changes on their clients is not small: 50% believe that the impact is moderate, 44% say that the impact is big or very big.
- Library developers had different opinions on how long it takes their clients to update. The most common estimate is less than an hour (39%) but some developers claim that it can take multiple days.

B. RQ.2 - Why do library developers introduce breaking changes?

This question has previously been answered by Brito *et al.,* [23] using the firehouse interview (sending an email to the developer as soon as the breaking change was detected). They reported that the most common reasons for introducing breaking changes are the need to implement new features (32%), API simplification (29%), and improving maintainability (24%). To verify those results, we asked a similar question in our survey of library developers. This was a checkbox question and the list of options was the same as the factors identified by Brito *et al.,* [23]. We also added an *"Other"* option and an optional open text field to identify additional factors. All 18 library developers answered this question. As can be seen in Figure 5, 12 developers identified *implementing a new feature* as a primary reason for introducing breaking changes; 11 developers identified *API simplification* and 11 developers selected *improving maintainability*; 6 developers selected *bug fixing*. This confirms the results that were previously reported by Brito *et al.,* [23]. Two developers also provided additional reasons: *research transfer* and *performance*.

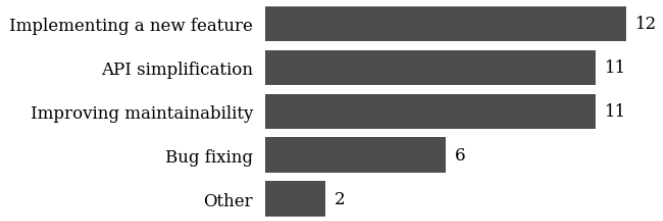

Fig. 5: What are your primary reasons for introducing breaking changes?

We also asked developers an open question: *"Were there specific scenarios when you had to sacrifice backward compatibility to introduce important changes? Can you describe them briefly?"*. We received 15 answers to this question. In addition to the four reasons listed above, 3 developers wrote that breaking changes are caused by *system redesign and architectural changes*, 2 developers mentioned *refactorings*. Also, 3 developers wrote that breaking changes in their APIs were caused by changes in their dependencies — a process that is known as the *ripple effect*, when a change in one library can impact other libraries that depend on it and propagate through the ecosystem from client to client. Two developers mentioned security issues related to authentication and one developer wrote that they break the API when they need to improve names or remove features that are not used.

Summary: The most common reasons for introducing breaking changes are the *implementation of new features* (12 dev. out of 18), *API simplification* (11 dev.), and *improving maintainability* (11 dev.), *bug fixing* was selected by 6 developers. This confirms the results of the previous study conducted by Brito *et al.,* [23]. Among other reasons mentioned by developers are *architectural changes*, *refactorings*, *security issues*, and *changes in other libraries*.

C. RQ.3 - How motivated are library developers to support their clients?

As we mentioned before, recently there were many studies that proposed automated tools to support developers in the process of library update. Some of those approaches are designed for client developers, while others are for library developers. For the second group of approaches, it is important to understand what is the motivation of library developers to support their clients.

To answer this question, we asked library developers two questions: *"How important is it for you to maintain backward compatibility?"* and *"How important is it for you to encourage clients to update to the latest version?"*. Those questions are different, because, as we seen in Section V-B, library developers have many good reasons to introduce breaking changes (and thus break compatibility) even if they want their clients to update easily.

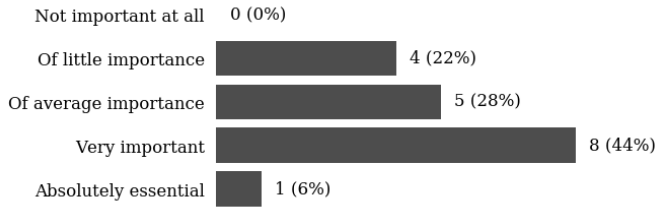

Fig. 6: How important is it for you to maintain backward compatibility?

In Figure 6, we present the answers to the first question. 4 developers (22%) specified that it is of little importance. For 5 developers (28%), backward compatibility is of average importance. 8 of the surveyed developers (44%) stated that it is very important for them, and for one developer it is absolutely essential to be backward compatible.

In Figure 7, we present the answers to the second question. Only one library developer specified that encouraging clients to update is of little importance. For 5 developers (28%) it

is of average importance, for 8 developers (44%) it is very important, and for 4 developers (22%) it is absolutely essential.

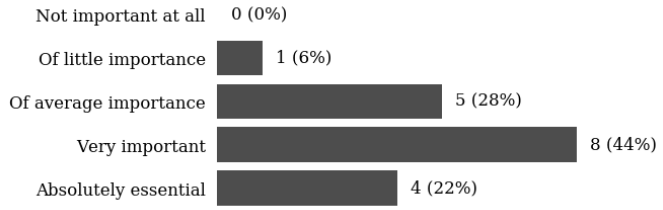

Fig. 7: How important is it for you to encourage clients to update to the latest version?

Summary:

- Maintaining backward compatibility is very important or absolutely essential for 50% of surveyed library developers.
- Encouraging clients to update is very important or absolutely essential for 66% of surveyed library developers.

D. RQ.4 - How do library developers help their clients to update?

To understand what library developers do to help their clients, we asked two questions in our survey. First, a checkbox question (multiple selection) intended to understand the software development practices adopted *before* the release: *"What software development practices do you use to improve the stability of your API?"*. Second, an open question about the practices *after* the release: *"When you are forced to break backward compatibility, what do you do to reduce the negative impact on users?"*.

The first question was answered by all 18 developers. We gave them three options: *weak coupling* (selected by 15 dev.), *abstraction layer* (13 dev.), and *microservice architecture* (5 dev.). In the *"Other"* section of this question, developers mentioned three more practices: *"design patterns"*, *"automatic transformation of deprecated methods"*, and *"independent software that checks the stability of the API and the continuity of its expected behaviour"*.

The open question was answered by 16 out of 18 surveyed developers. We analysed their responses and identified four practices that library developers use to reduce the negative effect of breaking changes on their clients:

- **Documentation** (8 dev.), including *migration guide* (4 dev.), *release notes* (1 dev.), and *deprecation comments* (1 dev.).
- **Deprecation** (4 dev.).
- **Automation** (4 dev.), including *rewriting deprecations* (2 dev.).
- **Communication** (3 dev.): including *communication before making the change* (1 dev.) and *live workshops to help clients update* (1 dev.).

Two developers explicitly said that they do nothing to help client developers. One developer, who mentioned that he/she uses the tools for automatic adaptation of source code, has also expressed discontent with such automation techniques: *"...usually, the devs prefers to see exactly how the code will be modified in the context of the application. When the change requieres actions on many part of the software, manually going through all the mofications by hand help to put again "in context" the impact of the modification"*.

Summary:

- Among the software development practices that improve the stability of API, the most popular were *weak coupling* (18 dev.) and *abstraction layer* (15 dev.). Developers also mention the *microservice architecture*, *design patterns*, and *automation tools* such as rewriting deprecations and API stability checks.
- Among the library developers who answered our survey, the most common practices to support clients after the introduction of breaking changes are: *documentation* (8 dev.), *deprecation* (4 dev.), *automation* (4 dev.), and *communication* (3 dev.).

VI. Results of the Client Developer Survey

In this section we present and discuss the results of the survey of client developers. As before, each research question is associated with one or more survey questions.

A. RQ.5 - How do client developers perceive impact of library evolution on their systems?

We asked the client developers two questions that mirror the questions that were answered by library developers and discussed in Section V-A. Each question was answered by 36 out of 37 client developers that took part in our survey.

First, we asked client developers to estimate, how much they are affected by the evolution of their dependencies. Notice that this question is slightly different from the library developer question in which we asked about the impact of breaking changes. As can be seen in Figure 8, the answers are almost equally distributed around the midpoint *somewhat affected* (14 dev.): 11 developers answered that they are *affected a little*, 10 developers — *significantly affected*, one developer claims to be *affected to a great extent*.

Then we asked client developers to estimate how long it usually takes them to update a dependency. As can be seen in Figure 9, the answers are scattered between *"less than an hour"* and *"several days"*. Only one developer answered *"one week or more"*. This trend is similar to the one we observed in Section V-A. Again, we hypothesise that the different answers are caused by different complexity of changes in libraries on which clients depend. Some libraries take less than an hour to update, others can take days.

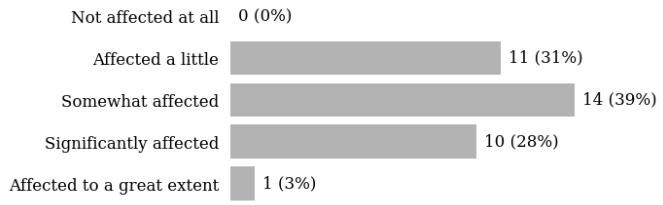

Fig. 8: How much are you affected by the evolution of your dependencies? (e.g., when one of your dependencies releases a new version or drops support for the old one)

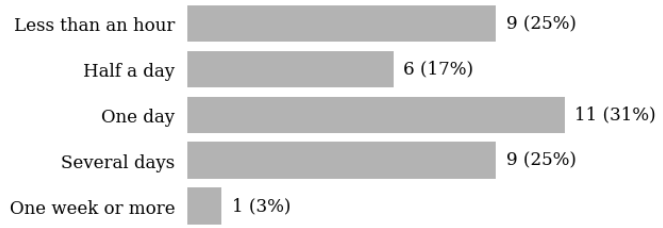

Fig. 9: How much time does it usually take you to update your dependencies?

Finally, we asked client developers to estimate how often they have to deal with the task of updating dependencies. As can be seen in Figure 10, out of 36 client developers who answered this question, 17 (47%) have to update their dependencies three times a year or more often, 10 (28%) have to do it twice a year. The other two options: once a year and less often each have 3 developers who selected them. Also, 3 developers stated that they do not update their dependencies regularly.

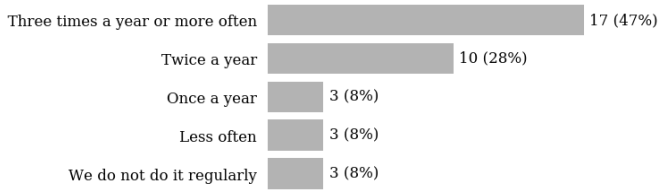

Fig. 10: Try to estimate how often do you have to deal with the task of updating dependencies

Summary:

- Most client developers in our study do not think that they are greatly affected by library evolution.
- The time required to update a library dependency can be different: from less than an hour to several days. Only one developer has claimed that it may take a week or more.
- 27 out of 36 client developers in our study have to update their dependencies at least twice a year; 17 developers do it three times a year or more often.

B. RQ.6 - What makes library update easy and what makes it hard?

To answer this research question, we asked client developers two open questions in the survey: *"When updating a dependency is easy, what makes it easy?"* and *"When updating a dependency is hard, what makes it hard?"*. Each question was answered by 34 out of 37 client developers who took part in the survey. We analysed their answers and summarise them in Tables II and III.

TABLE II: When updating a dependency is easy, what makes it easy? Second column (dev.) is the number of developers who mentioned this factor in an open question.

Factor	dev.
Documentation	15
Absence of breaking changes	11
Test coverage	6
Tool support	5
Deprecations	4
Simple breaking changes	4
Community support	3

TABLE III: When updating a dependency is hard, what makes it hard? (only factors that were mentioned by at least 2 developers).

Factor	dev.
Breaking changes	11
Absent or bad documentation	10
Indirect dependencies	7
Big changes to the API	7
Poor tests coverage	4
Removed functionality	3
Changed hooks or abs. classes	3
No community support	2
Behavioral changes	2

The most common factors that make library update easy are good *documentation* (mentioned by 15 developers, 44%) and the *absence of breaking changes* (11 dev., 32%). Other factors include *test coverage* of client code, *tool support* such as dependency managers and automated code rewriting, *deprecations* that are introduced before removing functionality, *simple breaking changes* such as method renaming, and *community support*. The most commonly mentioned factors that make library update hard are *breaking changes* (mentioned by

11 developers, 32%), *absent or bad documentation* (10 dev., 29%), *indirect dependencies* to other libraries that can result in version conflicts (7 dev., 21%), and *big changes to the API* that force clients to change the logic of how the library is used (7 dev., 21%). Other factors that were mentioned 2-4 times are *poor test coverage* of client code, *removed functionality*, *changed hooks and abstract classes*, *absence of community support*, and *behavioral changes*. There are also factors that were mentioned only once: strong coupling, expertise required, security issues, naming collisions with other libraries, absence of deprecations, bugs and compilation errors. We did not include those factors in Table III.

Summary: The two most important factors affecting the complexity of library update are breaking changes and documentation. 11 out of 37 client developers in our study mentioned that breaking changes make updating hard, and also 11 developers mentioned that their absence makes it easy; 15 developers mentioned that good documentation makes updating easy, and 10 developers mentioned that missing or misleading documentation makes it hard.

VII. Threats to Validity

We discuss the four types of validity threats that were presented by Wohlin *et al.,* [25]: internal, external, construct, and conclusion validity.

a) Internal Validity:

- We did not ask the participants to focus on a single project but rather to answer the questions based on their general experience. This makes it hard to further analyse the contexts that originate certain situations/decisions.
- Some questions in our survey are hard to answer generally. The answers can vary depending on the project that developer has in mind or specific situations on the client side. For example, when we asked how much time it takes for clients to update, the answer could depend on how much of the API was used by a particular client.
- The transforming deprecation mechanism of Pharo [12] may have affected how Pharo developers answered this survey.
- Inside the three mentioned communities, the surveys were public and participation was optional. This means that many library developers could also have answered the client developer survey. We can not verify this because the surveys were fully anonymous and we did not want to restrict participation to only one survey because each library developer might also be a client developer for other libraries.

b) External Validity:

- For convenience, in this study we surveyed developers from an open-source community that we (authors) are part of and two industrial companies that collaborate with our research group. The participation was optional, which means that the developers who responded to our call might be the ones who know us personally and have interest in our research. This might have introduced a bias.
- The library developers in our survey are responsible for libraries with no more than 500 clients. Those libraries can be considered small compared to the top-1000 libraries in NPM and Maven. This means that our study may not be representative of the large libraries and frameworks, but more focused on the smaller libraries that are also more common. Although, we list it as a threat, we also believe this to be a novelty of our study because most related works focus only on large libraries, which introduces a bias on their side.

c) Construct Validity:

- Some research questions in this survey can not be fully expressed with specific survey questions. For example, to measure the impact of library evolution on clients, we ask questions about the perception of this impact (too general) and another question about the time it takes to update (too specific).
- To save the time of our participants, most questions in our survey were not open but contained a list of options to choose from. This poses a threat to construct validity because developers were limited and biased by those options. To address this threat, we tried to provide a diverse list of options for each question, taking inspiration from the literature (*e.g.,* the options for RQ2 were the same as the ones identified by Brito *et al.,* [23]) or by discussing them with our colleague developers.
- Measuring the impact of breaking changes on clients may be seen as a "self-fulfilling prophecy": by definition they do have an impact. Whether this impact is experienced or not depends on the actual use of the library.

d) Conclusion Validity:

- The population size is relatively small. Our study involved 18 library developers and 37 client developers. Although it is comparable to other surveys in this field (*e.g.,* Bogart *et al.,* [22] surveyed 28 developers, Xavier *et al.,* [24] — 7 dev., Kula *et al.,* [18] — 16 dev., and Brito *et al.,* [23] — 102 dev.), this population might be too small to claim statistical significance.
- Considering the relatively small population size, we did not perform statistical tests and derived conclusions from simple descriptive statistics.

VIII. Conclusion

In this article, we presented the results of a first survey of the perception of library evolution by both client developers (who need to update their software to new versions of libraries they use) and library developers (who produce new versions of libraries). The survey involved software developers from two industrial companies: Arolla and Berger Levrault, and one open-source community: Pharo. We have confirmed the results of a previous study performed by Brito *et al.,* [23] who reported that the most common reasons for introducing

breaking changes are new features, API simplification, and maintainability improvement. We have also identified what makes library update easy and what makes it hard for clients, and the kind of support library developers can provide them. The survey results presented in this article can help better understand the process of library update from the perspective of library and client developers, identify the challenges that they face and what can be done to overcome those challenges.

REFERENCES

[1] M. T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio, "An industrial case study on reuse oriented development," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 283–292.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[3] M. Lehman, "Laws of software evolution revisited," in *European Workshop on Software Process Technology*. Berlin: Springer, 1996, pp. 108–124.

[4] T. Mens, J. F. Ramil, and M. W. Godfrey, "Analyzing the evolution of large-scale software: Issue overview," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 363–365, Nov. 2004.

[5] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 333–343.

[6] E. Xing, ZhenchangandStroulia, "API-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, pp. 818 – 836, 2007.

[7] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–35, 2011.

[8] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "Apievolutionminer: Keeping api evolution under control," in *Proceedings of the Software Evolution Week (CSMR-WCRE'14)*, 2014.

[9] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2008, pp. 471–480.

[10] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 192–201.

[11] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes." in *International Conference on Software Maintenance (ICSM)*, vol. 96, 1996, p. 359.

[12] S. Ducasse, G. Polito, O. Zaitsev, M. Denker, and P. Tesone, "Deprewriter: On the fly rewriting method deprecations," *JOT*, 2022.

[13] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *Proceedings International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 274–283.

[14] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the Pharo ecosystem case," in *International Conference on Software Maintenance (ICSM'15)*, 2015, pp. 251–260.

[15] A. Hora, R. Robbes, M. Tulio Valente, N. Anquetil, A. Etien, and S. Ducasse, "How do developers react to api evolution? a large-scale empirical study," *Software Quality Journal*, vol. 26, pp. 161–191, Mar. 2018.

[16] K. Jezek, J. Dietrich, and P. Brada, "How java apis break–an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[17] R. G. Kula, D. M. German, and A. O. andTakashi Ishio andKatsuro Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[18] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used apis," *Information and Software Technology*, vol. 93, pp. 186–199, 2018.

[19] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: The case of a smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering (FSE)*. New York, NY, USA: ACM, 2012, pp. 56:1–56:11.

[20] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular java APIs," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 400–410.

[21] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 138–147.

[22] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

[23] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in APIs," *Empirical Software Engineering*, pp. 1–35, 2019.

[24] L. Xavier, A. Hora, and M. T. Valente, "Why do we break APIs? first answers from developers," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 392–396.

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.