Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2014; **26**:1030–1052
Published online 31 July 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1660

A study of library migrations in Java

Cédric Teyton*,†, Jean-Rémy Falleri, Marc Palyart and Xavier Blanc

LaBRI, Bordeaux University, UMR 5800, F-33400 Talence, France

ABSTRACT

Software intensively depends on external libraries whose relevance may change during its life cycle. As a consequence, software developers must periodically reconsider the libraries they depend on, and must think about replacing them for more relevant ones. We refer to this practice as *library migration*. To find the best replacement for their library, they can rely on information over the Web, but they get quickly overwhelmed by the amount of data they gather. Making the right choice in this context constitutes the topic of our work. The solution we propose is to exhibit and mine the library migrations trends computed by performing a study of a large set of software projects. To perform this analysis, we have defined an automatic approach to compute library dependencies and a semi-automatic approach that identifies library migrations. Then, we propose a deep analysis of the library migration phenomena by performing a descriptive study of a large set of software projects stored on the Githubplatform. Second, based on our descriptive study, we propose a support to developers who want to migrate their libraries. The main result of our study is that recommendations of libraries can be inferred from the analysis of the migration trends. Copyright © 2014 John Wiley & Sons, Ltd.

Received 19 April 2013; Revised 20 May 2014; Accepted 10 June 2014

KEY WORDS: software evolution; software repositories; API analysis

1. INTRODUCTION

The use of software libraries for reusing code is a widespread approach [1]. Examples of such libraries are *junit* for unit testing or *log4j* for logging. The relevance of a particular library for a software project may change during the project life cycle. This change might be motivated by the fact that the library is no longer updated, that a competing library with better features is available or that the library is the cause of compilation issues. As a consequence, software developers must periodically reconsider the libraries they use. We refer to the task of replacing a given library by a different one having equivalent features as a *library migration*.

To the best of our knowledge, no existing study has been performed to deeply understand the library migration phenomenon. To what extent software projects perform third-party library migration? Are library migrations frequent? For which kinds of libraries are they performed and when? For what reasons? And so on. This paper aims to uncover these interrogations.

Furthermore, the migration of a library is well known to be a complex task because of the differences between the structure, the protocol and the terminology of the library in use, and the one chosen as a replacement. Existing research approaches tackle this issue by providing support for the update of the source code [2–4]. However, they do not provide any help for the developer that do not know to which library she should migrate to. In such situation, developers have to cope with the large amount of information they can gather by using web search

*Correspondence to: Cédric Teyton, LaBRI, Bordeaux University, UMR 5800, F-33400 Talence, France.
†E-mail: cteyton@labri.fr

engine. This potentially results in an information overload and thus the replacement of a library can be overwhelming.

The purpose of this paper is then threefold. First, we propose a generic approach to extract and identify library migration trends. Second, we design a deep analysis of the library migration phenomenon by performing a descriptive study of a large set of software projects. The intent of this analysis is to measure the number of dependencies between well-known third-party libraries and existing software projects as well as the frequency of their migrations. Third, based on our descriptive study, we propose a support to developers who want to replace a library they use by a new equivalent one. Rather than providing a set of recommendations, we exhibit the library migration trends depending on two main factors: the technical domain covered by the library and the date of the migrations.

In order to perform our analysis, we have designed an approach that identifies automatically library dependencies and semi-automatically library migrations. Regarding the identification of library dependencies, our approach is based on a static analysis of the source code. Regarding the identification of library migrations, it relies on the version control system (VCS) and requires a manual review to clean wrong observations. Our approach has been completely prototyped and currently only supports Java software projects.

This paper extends our previous work [5] in several directions. First of all, we have drastically improved our approach for identifying library dependencies and library migrations. Our past approach was based on Maven, a tool used to manage dependencies between software projects. Our past approach was therefore only able to analyze software projects managed with Maven. Our new approach is generic as it is based on a static analysis of source code (currently only Java) and on an observation of VCSs. It now supports any Java project. Secondly, we provide a deep analysis of the library migration phenomena that was not present in our previous paper. Such analysis shows the dependencies between well-known libraries and existing projects and measures how frequent are their migrations. Third, we have improved our migration trends by taking into account a second important factor that is the time of library migrations. In our past paper, migration trends were only computed according to the technical domain covered by the libraries. Now, they also exhibit the temporal aspect of migrations. Even though it is not statistically significant, we also present an analysis performed on VCSs and bug trackers to better understand the reasons behind library migrations.

The remainder of this paper is structured as follows. Section 2 first explains our approach for identifying library use and migrations. Then, Section 3 presents the descriptive study we performed on projects stored on GitHub hosting platform. Section 4 then introduces the migration trends we computed. Section 5 discusses the limitations of our approach. Section 6 presents the related work while Section 7 presents the future work and concludes.

2. IDENTIFYING LIBRARY DEPENDENCIES AND MIGRATIONS

In this section, we present the approach we propose to automatically identify library dependencies and semi automatically identify migrations. To that extent, we first introduce our model to represent software libraries, projects, library dependencies and library migrations. We finally present the static analysis process that identifies library dependencies and the algorithm that identifies library migrations.

2.1. Dependency model

The model we use to identify library dependencies and migrations is straightforward as it only contains the set of libraries taken into consideration, the set of analyzed software projects, their list of versions and, for each version, the associated set of library dependencies.

We assume that a library is an entity that has a unique name and that exports a set of symbols. The symbols of a library are syntactic elements that can be used by software projects that depend on it. We consider that a software project that uses a library must include at least one of the symbols of the library in its source code.

DOI: 10.1002/smr

Definition 1 (Third-party library)
Let L be the set of well known libraries. A library $l \in L$ has a name (unique in L) and a set of associated string symbols S_l. The symbols of a library correspond to the fully qualified names of the syntactic elements exported by the library. For the sake of simplicity, we consider that the intersection of two sets of symbols of two distinct libraries is empty (More formally, $\forall l_1, l_2 \in L, l_1 \neq l_2 \rightarrow S_{l_1} \cap S_{1_2} = \varnothing$). Moreover, we do not consider the versions of a library. The set of symbols of a library therefore includes all the syntactic elements that are defined in any version of that library. Finally, associated to L, we define the $library_L$ function that inputs a symbol s and that returns the library l in which it is defined. More formally, $library_L : String \rightarrow L \cup \varnothing$ such as $library_L(s) = l$ if and only if $s \in S_l$. $library_L(s) = \varnothing$ if and only if s is not defined by any of the libraries of L.

Let us illustrate our model with an example. We consider four libraries (*junit*, *testng*, *log4j*, and *slf4j*). Table I presents the symbols of these libraries. For the sake of simplicity, as it is a naming conventionin Java, we use the '*' char as a joker character. As a consequence, the symbols of a Java library correspond to names of the packages defined by the library.

Definition 2 (Software projects and Libraries)
Let P be the set of software projects. For the sake of simplicity, we consider that each project $p \in P$ has an associated totally ordered set $V_p \subset \mathbb{N}$ of versions. Versions are sorted chronologically according to their date. We consider that a project may depend on a library at one of its version if it makes use of at least one of the symbols of the library at that version. For a project $p \in P$ at version $i \in V_p$, we then define $dep_p(i) : V_p \rightarrow P(L)$ the set of its library dependencies.

Figure 1 provides an example of such evolution of dependencies within a project p. This project depends on two libraries, *junit* and *testng*.

A library migration occurs when a project replaces one of its library by a similar one. As the definition of similarity between libraries cannot be rigorously defined and will always stay unclear, we choose to consider that a candidate library migration is observed when a project stops to use one of its library and in the meantime starts to use another one.

Definition 3 (Candidate library migration)
We state that a project $p \in P$ may migrate from a library $s \in L$ to another library $t \in L$ during a period defined between two versions $i, j \in V_p$ if and only if (1) it depends on s and not on t at version $i \in V_p$

Table I. Example of symbols for a set of four well known libraries.

Library	Symbol
junit	org.junit.*
testng	org.testng.*
log4j	org.apache.log4j.*
slf4j	org.apache.slf4j.*

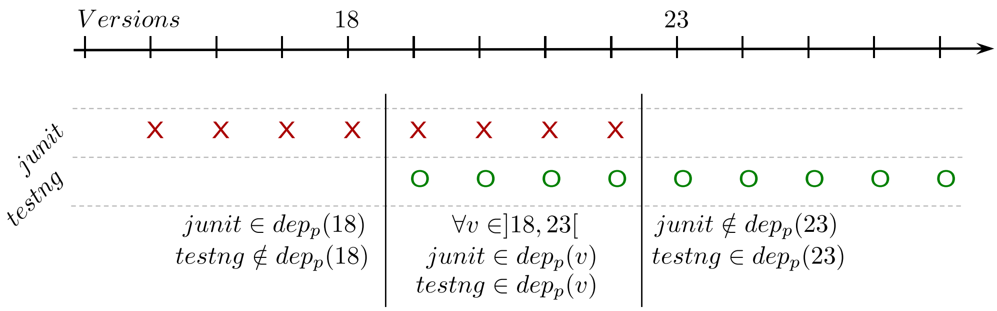


Figure 1. Example of a migration within a project p.

DOI: 10.1002/smr

(i.e. $s \in dep_p(i)$ and $t \notin dep_p(i)$), and (2) it depends on t and not on s at version j (i.e. $s \notin dep_p(j)$ and $t \in dep_p(j)$), and (3) it depends on both libraries during the migration (i.e. $\forall\, v \in]i, j[, s \in dep_p(v)$ and $t \in dep_p(v)$). A candidate library migration is therefore a tuple $(p, i, j, s, t) \in P \times \mathbb{N} \times \mathbb{N} \times L \times L$. Finally, given a set of projects P and a set of libraries L, we note M all the candidate migrations observed for all projects in P.

Regarding the example of the project p presented in the Figure 1, we identify one candidate migration: $(p, 18, 23, junit, testng)$. When observing a large set of projects, many candidate library migrations are observed. To better analyze which libraries are the targets or the sources of candidate migrations, we introduce the concept of a migration rule that identifies only the source and target library of a migration without any attention to the project or to the period.

Definition 4 (Candidate library migration rule)
A migration rule is a couple $(s, t) \in L^2$ such that there exists at least one project $p \in P$ that migrates from s to t during its life cycle. We note R the set of all library migration rules.

Regarding our example from the Figure 1, we have only one migration rule, $R = \{(junit, testng)\}$.

2.2. Automatic computation of library dependencies

The idea is now to extract the set of libraries used by a project regardless of its configuration management system (i.e., whether it uses systems like Maven or Gradle or none of them) with the intent to be as generic as possible. We first need to describe how the set of symbols S_l defined by a library l can be computed. We introduce a lightweight approach that is based on a static analysis of the source code of a library. To that extent, we consider that a library is at least defined by its unique name and by its set of source code files. For Java, this information is obtained from the jar files of well-known third-party libraries (by well-known we mean that the libraries are largely used by software projects and that their jar files are freely available).

Our approach then sequentially analyzes all the files of a library l and creates the corresponding set of symbols S_l. For Java, this consists of identifying all packages defined by the library and then returning their simplified representations that make use of the '*' char. For instance, the static analysis of the *testng* library will identify three packages ($org.testng$, $org.testng.annotations$, and $org.testng.xml$) and will therefore return only $org.testng.*$ as a simplified representation. Note that when several versions of a library exists, the operation is performed for each version and the results are finally merged in a single set.

Once all the symbols of the analyzed libraries are computed, our approach looks for symbol conflicts between each pair of distinct libraries. A symbol conflict occurs when two libraries define the same symbol. In Java, such a situation occurs in the two following cases:

- *library inclusion* When a library is deployed in several jar files and when one of these jar files corresponds to the full version of the library (l_{full}), whereas the others correspond to some smaller components (l_{comp}). As a consequence, all the symbols defined by the components are all included in the full version of the library. We then argue that only the full version should be considered as a library. The components should not be considered at all. As a consequence, when all the symbols of a library l_{comp} are included in another library l_{full} (i.e. $S_{l_{comp}} \subset S_{l_{full}}$), then we remove l_{comp} from the set of libraries taken into consideration. This is the case for instance with the *batik* library that provides the *batik-all* jar file that contains the full version and several other jar files that represent smaller parts of the *batik* library. We therefore consider only one *batik* library with all the defined symbols.
- *library reuse* When a library l_a uses another library l_b and modifies some of its parts. As a consequence, the jar file of the l_a library exports symbols that were initially defined by l_b. For the sake of simplicity, we choose to remove all the symbols that are redefined by a library. This is the case for example with the *junit* library that contains the symbol $org.hamcrest.core$, which was initially defined by the *hamcrest* library. We then choose to remove this symbol from *junit*.

It should be noted that symbol conflicts are resolved manually as the resolution strategy mainly depends on the programming language and on the habits of the developers of the libraries.

DOI: 10.1002/smr

Finally, once all the symbols of all the libraries have been identified and once all the conflicts have been resolved, the library extraction step can be applied on the source code of the analyzed software projects at any given version *i*. It assumes the existence of a search text program that can analyze the source code files of the software projects in order to detect the presence of symbols defined by any library. The library extraction maintains a set $dep_p(i)$ of dependent libraries for each software projects at each version *i*. This parser is applied sequentially on every files of the software projects and tries to match them with all the symbols defined by the libraries. Whenever a match occurs, the corresponding library is identified (using the function $library_L(s)$) and is added within the $dep_p(i)$ of the software project. At the end of the process, all $dep_p(i)$ of all software projects contain the dependent libraries used in the source code of the software projects.

2.3. *Semi automatic identification of library migrations*

To identify candidate migration rules, we propose an algorithm that detects if there are two libraries (*s* and *t*) that have been added and removed in a same period of time. Our algorithm (Algorithm 3) iterates through the versions of a project and, each time a dependency is removed, it creates a candidate migration for each target library that can be identified as a replacement (i.e. the ones that are added in the meantime).

Algorithm 1 Algorithm to automatically identify candidate library migrations

```
Require: $p$                                  ▷ a project
Require: $V_p$                                ▷ the set of versions of p
Require: $dep_p$                              ▷ the function that computes the dependencies
  Cand $\leftarrow \emptyset$                 ▷ the set of candidate migrations
  Dep $\leftarrow \emptyset$                  ▷ the list of active dependencies
  for all $i \in V_p$ do
      $L \leftarrow dep_p(i)$                 ▷ put dependencies at i into L
      for all $l \in L \setminus Dep$ do      ▷ for all dependencies that are added at i
          $append(Dep, l)$                    ▷ add them to the list of active dependencies
      end for
      for all $s \in Dep \setminus L$ do      ▷ for all dependencies s that have been removed at i
          for all $t \in tail(Dep, s) \cap L$ do ▷ take the dependencies still active that were added after s
              $Cand \leftarrow Cand + (s, t)$ ▷ update the set of candidate migrations
          end for
          $remove(Dep, s)$                    ▷ remove s from the list of active dependencies
      end for
  end for
```

The automatic identification of candidate migrations returns a large quantity of migrations in which there are many false positives. In our previous article, we proposed to use a data mining process to filter out false positive [5]. This process was not perfect even though we reached a reasonable precision, but the recall was significantly impacted. As our objective is to make a deep analysis of the migration phenomena and hence to identify all true migrations whatever the cost of the analysis. We therefore claim that it is worthwhile to dedicate efforts to manually rate the migrations in order to identify all the true migration rules.

Performing the manual analysis consists of looking at all the candidate migration rules that have been identified and to check whether they identify true or false migrations. We therefore ask to an expert developer to evaluate migration rules according to the similarity of their target and source libraries. If the expert considers that the source and target libraries of a candidate migration rule are similar (i.e. they offer similar services such as unit testing or XML parsing facilities), then we consider that the candidate migration rule is a true migration.

```
SELECT repository_url, count(repository_url) as pushes
FROM [githubarchive:github.timeline]
WHERE type="PushEvent"
AND repository_language="Java"
AND repository_fork="false"
AND repository_private="false"
GROUP BY repository_url
HAVING COUNT(*) >= 10
```

Figure 2. The query used to gather a set of 20 000 software projects.

3. DESCRIPTIVE STUDY

To better understand the phenomena of library dependency and migration, we conducted a descriptive study on Java open source software projects stored on the GitHub hosting platform. We have focused our study on the Java programming language for two reasons. First, it is a mature and widely used language, in which there exists thousands of libraries. Therefore, it is a perfect substrate to find library migrations. Second, there are many existing tools that make easy the static analysis of Java code, making the implementation of our approach easier. We chose software projects stored on GitHub for two reasons. First, GitHub stores a huge set of projects of different nature (small or large, young or old, etc.). Second, it relies on Git, a fast and distributed VCS that makes our implementation easier to design and faster to run.

In this section, we first introduce the data set of our study that is composed of a set of well-known libraries *L* and a large set of software projects *P*. Second, we describe the results of the extraction of library dependencies we performed on these projects. Third, we describe the results of the extraction of library migration.

3.1. Data set

Our descriptive study is based on a predefined set of libraries *L*. However, to the best of our knowledge, no such a set is currently defined. Moreover, as libraries are developed like any classical software projects, it is sometimes hard to distinguish a software library from a classical software system. We then arbitrarily decided to reuse a set of Java libraries we have computed in our previous work [5]. This set has been built by downloading the Jar files of libraries stored in the Maven Central repository. It is composed of 8795 jar files.

We then grouped these files according to the similarity of their names to discard any version information. For instance, we consider that *junit-4.8.1.jar* and *junit-4.8.2.jar* are part of the same *junit* library. After this operation, we obtained 3326 libraries. We then apply the process described in Section 2.2 and extracted the symbols from the 3326 jar files. Then, one person spent about 14 h to manually resolve the symbol conflicts. After this operation, we obtained a set *L* of 1189 libraries with their associated symbols.

To build our corpus of projects *P*, we decided to exclusive rely on the GitHub open-source platform. We used the data provided by the Github Archive[‡] in addition to the Google BigQuery service to gather a set of 20 000 software projects. Our goal was to have a large set of software projects to be able to observe library migrations.

Figure 2 presents the query we executed on Google BigQuery. We designed it to return us repositories that have at least 10 commits, that are not private and that are not forked repositories. Our intent was to have no empty projects and no duplicated projects that would clearly bias the results of our study. Note that the query was executed in October 2013.

In a second step, we discarded all the projects that have less than 100 Java LOC, less than 30 days of activity and use no library. Note that this last filter cannot be expressed with BigQuery. Our claim is that these projects are not interesting for our concern. During this step, we also removed repositories that URL did not respond any-more, probably because they were deleted from GitHub. Thanks to this

[‡] Http://www.githubarchive.org/

Table II. Corpus description.

	Deciles									
Metrics	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Versions	19	27	37	49	65	87	123	188	360	13 130
Java KLOC	0.7	1.2	1.8	2.6	3.8	5.6	8.6	15.2	35.9	31 611
Developers	1	1	1	2	2	3	3	5	8	855

Table III. Deciles of the population of projects according to the number of libraries they use.

	Deciles									
Population	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Smallest	1	1	2	2	3	4	5	6	8	27
Biggest	2	3	4	5	6	8	10	13	19	109
Total	1	2	2	3	4	5	6	8	13	109

second step, we obtained a set of 15 168 projects. It means that that 24% of the projects returned by Google BigQuery were either almost empty, or did not use any library, or were removed from GitHub.

We computed some statistics to better understand our corpus. In particular, we have made three measures to analyze the size of the projects in terms of number of versions, KLOC, and number of developers. Table II shows the deciles for these three measures. It shows that their distributions roughly follow a power law.

To better observe the influence of the size of a project on the library dependency and migration, we then decided to partition our corpus in two groups. One group gathers the biggest projects (named Biggest in the rest of this paper). It is composed of the union of 20% biggest projects respectively in term of number of versions, KLOC, and number of developers. In total, it is composed of 5959 projects (39.3 % of the corpus). The other projects compose the group of medium and small projects (named Smallest in the rest of this paper). It is composed of 9209 projects (60.7 % of the corpus).

4. LIBRARY DEPENDENCIES

To implement the approach of library dependency identification presented in the Section 2, we developed a prototype based on the Harmony framework [6]. Harmony is an infrastructure designed to ease the development of tools for software evolution analysis. Further, the computation of library dependencies is performed by a tool named SCANLIB we implemented for this study .[§] For each version of each project, the tool will be executed to textually search for any library symbol used by the project. This operation is quite fast as it only requires few seconds for each version.

Running our analysis on our corpus, we observed that 1018 of the 1189 libraries were used by the projects (85% of *L*). Such a result gives a high confidence to *L*, our arbitrarily chosen set of well-known libraries. Even though this is certainly not perfect because we are likely to miss some libraries, at least *L* includes most of the well-known used third party libraries (thanks to the nature of its construction).

We then analyzed the distribution of the used libraries among the projects. Table III shows the deciles of the number of used libraries for our two groups of projects (Biggest and Smallest). The Smallest projects use few libraries as 70% of them use five or less libraries. In the contrary, the Biggest projects use much more libraries as 40% use five or less libraries, meaning that 60% of them use more than six libraries. Further, 30% of them use more than 10 libraries.

Further, Table IV shows the deciles for the usage of libraries regarding our two groups of projects. The two distributions are quite similar. They show that 80% of the libraries are used by few projects

[§] Https://code.google.com/p/scanlib-java/

Table IV. Deciles of the population of libraries according to the number of client projects.

	Deciles									
Population	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Smallest	0.0	0.0	1.0	2.0	3.0	5.0	9.0	19.8	52.0	3566.0
Biggest	1.0	2.0	3.0	5.0	7.0	13.0	22.0	42.0	103.4	3610.0
Total	1.0	3.0	4.0	6.0	10.0	17.0	31.0	59.8	149.0	7176.0

(less than 59.8%, i.e. 0.4% of the projects *P*) whatever the size of the projects. Inversely, 10% of the libraries are used by more than 149 projects from our corpus. In other words, even though Biggest projects use more libraries, still few libraries are famous and widely used.

Further, to better understand which of the libraries are the most popular, Table V shows the 15 most used libraries in our corpus with their technical domain. In particular, we can observe the substantial popularity of *junit*, which appears in almost 50% of the projects. This library is a very mature framework for writing unit tests in Java software projects. Additionally, we notice that a set of 10 domains is covered in this ranking.

4.1. Extraction of library migrations

To implement the approach of library migrations identification presented in the Section 2, we developed a prototype based once again on the Harmony framework [6]. This prototype inputs a set of projects *P* and a set of libraries *L* and returns a set of candidate migrations grouped by migration rules.

We obtained a total of 28 052 migrations grouped in a set of 17 113 migration rules. As we described in 2, this set contains a large quantity of false positive migration rules that have to be manually eliminated. In our experiment, one Java expert (one of the authors) spent 2 days to manually check the 17 113 candidate migration rules and rated them either as correct or wrong. The process consisted of searching on the web to compare the description of libraries of each rule to check whether they have a similar purpose. At the end of the validation process, we obtained a total of 1198 migrations performed by 866 projects. One hundred sixty-four libraries were involved in these migrations. The 1198 migrations have been grouped into 329 migration rules.

Our manual step reveals that our approach has performed with a precision of 2%. Even if precision was clearly not part of our requirements for this study, such a bad result clearly shows that the automatic identification of library migrations is hard and that there is clearly a room for improvement. Further, it also shows that many libraries are added and removed during the life cycle of a software projects. However, only few cases of these additions and removal are library migrations.

Table V. The 15 most used libraries in our corpus.

Library	Clients	Biggest	Smallest	Category
junit	7176	3610	3566	Unit Testing
android	3672	1239	2433	Mobile Applications
xmlParserAPIs	3088	1968	1120	XML Processing
commons-lang	2558	1514	1044	Helper Utilities
slf4j	2248	1182	1066	Logging
spring	2064	1041	1023	J2EE
guava	1998	1134	864	Helper Utilities
httpclient	1940	963	977	Http Resources
httpcore	1905	942	963	Http Resources
log4j	1843	1084	759	Logging
commons-io	1794	1134	660	Input/Output Helper Utilities
org.json	1596	762	834	JSON Processing
mockito	1360	797	563	Testing
jackson	1287	732	555	JSON Processing
hamcrest	1287	772	505	Testing

Table VI. Ten most performed migration rules.

Source	Target	Score	Category
log4j	slf4j	95	Logging
commons-logging	slf4j	61	Logging
junit	testng	45	Unit testing
commons-httpclient	httpclient	33	Http resources
commons-httpclient	httpcore	32	Http resources
org.json	gson	28	JSON processing
testng	junit	27	Unit testing
org.json	jackson	26	JSON processing
hamcrest	fest	25	Testing
easymock	mockito	23	Testing
gson	jackson	22	JSON processing

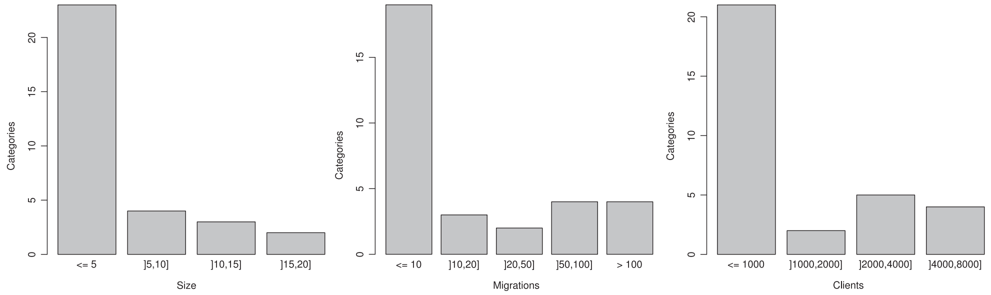


Figure 3. Distribution of the categories size (left), migrations (center) and client projects (right).

As a first result, our analysis shows that 866 projects on the 15 168 of *P* have performed a migration (5.57% of *P*). In particular, 593 belong to the Biggest projects (68.5%). This means that 9.95% of the Biggest projects have performed a migration. This confirmed one of the result of our previous paper that was that the practice of library migration is real but quite occasional.

To better understand the migrations, Table VI displays the 10 most observed rules in terms of number of migrations (whatever the size of the projects in which the migration have been performed). It is interesting to observe that the most observed migrations involve many of the most popular libraries as seen just earlier. The full list of migration rules is available on-line.[¶]

Finally, to better exhibit the categories of libraries, we decided to group together libraries that are connected by migration rules (directly or transitively). We reuse the concept of migration graph presented in the earlier version of our work [5]. The nodes of such graph are libraries that have been either source or target of at least one migration. A directed arc exists between two nodes if there is at least a migration between the two corresponding libraries. Connected components of the migration graphs represent categories of libraries. A category therefore identifies a set of libraries that address a same domain and where migrations have been observed between them. We call the size of a category the number of libraries it contains.

With our corpus, we obtained 32 categories of libraries. The distribution of the size, the number of migrations, and client projects for each category is shown in Figure 3. We can see that nine categories contain more than five libraries. Further, we can observe that 19 categories contain less than 10 migrations, which represents a small number of migrations. However, there are 10 categories where more than 10 migrations happened. Finally, we also note that 11 categories are used by more than 1000 projects from our corpus. By looking at Table VII, which displays the top 10 categories in terms of migrations, we observe that nine out of 10 are used by more than 2000 projects. Here, we find once again the domain of libraries introduced previously in Table V and in Table VI.

[¶] http://se.labri.fr/migrations/rules.html

Table VII. Top 10 categories in terms of migrations.

Category	Migrations	Size	Clients
Logging	238	4	4113
JSON	210	13	4116
Collections	128	14	4235
Testing utilities	106	8	2527
Http	100	5	2372
XML parsing	90	17	3726
Database	83	15	2259
Unit testing	72	2	7382
Reflection	43	8	2731
GUI	24	11	584

4.2. Synthesis

This descriptive study helps to better understand the phenomena of library dependency and migration. First, it shows that Biggest projects use more libraries than Smallest ones. Second, few libraries are widely used by all projects, whatever their size, while most of the libraries are seldom used. Regarding library migration, our descriptive study clearly shows that such a phenomena does exist. We report that Biggest projects are more prone to have performed a library migration than Smallest ones. Still, migrations occur within both categories of projects. Our study also shows that libraries can be gathered into categories. The study of these categories is therefore very useful to support the developers that want to migrate and who do not know to which library to migrate to, which is the intent of the next section.

5. MIGRATION TRENDS

In this section, we conduct a thorough analysis of the categories of migrations produced in Section 3. The goal is to highlight migration trends to help developers in a context of library migration. We first introduce the objectives for this analysis. Second, we present so-called library migration graphs. Third, we study the temporal evolution of the migration trends and demonstrate that they can be used as a complement of the migration graphs. We finally try to provide a taxonomy of the motivations that led developers to migrate.

5.1. Objectives

Our purpose is to help developers to choose a relevant target library when they consider to migrate one of their dependent library. In this context, the questions they certainly ask and we aim to answer are the following: is there a library that is massively adopted? Are there emerging libraries that start to be adopted? Are there libraries that are massively abandoned?

To rigorously answer these questions, we introduce two main instruments that aim to identify trends regarding library migration. The first instrument is what we call a migration graph. It shows the libraries of a category and highlights the migrations that did occur between them. The second instrument is what we call a migration evolution graph. It shows when migrations have been performed with the intent to quickly evaluate when migrations were performed.

Those two instruments are based on a quantitative measure of library migrations. In particular, we propose to measure what we call the *migration degree* as the difference between the in-degree migrations and the out-degree for each library in a category. The in-degree is the sum of the incoming migration flows, whereas the out-degree indicates the sum of the outgoing migration flows. A positive degree indicates that the library has undergone more arrivals than departures, whereas a negative degree indicates the opposite situation.

Moreover, we propose to categorize libraries depending on their migration degree. In particular, we propose the following patterns:

- *Best Challenger* is the library having the highest migration degree in a category
- *Challenger* is a library having a positive migration degree in a category
- *Breaking Bad* is a library having a negative migration degree in a category
- *Worst Breaking Bad* is the library having the lowest migration degree in a category

Our claim is that *challenger* libraries should be recommended as a target to migrate to, with a particular attention to pay to the *best challenger*. Inversely, the *breaking bad* libraries should not be recommended, especially the worst breaking bad.

Finally, associated with our two instruments, we attempt to extract the reasons and motivations behind the library migrations we observed. Our intent is to provide feedbacks from real projects to the developers who intend to replace their libraries.

The two first following sections describe the two instruments we propose along with some examples. The third section presents our approach to extract feedbacks related to migrations.

5.2. Library migration graph

5.2.1. Motivation. Our purpose is to highlight in a single visualization all the migrations of a category as well as the migration degree of each library. Using this, it becomes easy to identify all the *breaking bad* and *challenger* libraries of the category, and also the *best challenger* and *worst breaking bad* if they exist.

5.2.2. Methodology. We extended the migration graph proposed in the earlier version of our work [5]. In such graph, the nodes are libraries that have been either source or target of at least one migration rule. A directed arc exists between two nodes if there is at least a migration between the two nodes. The arcs are labeled with the score of the migration, that is, the number of times it has been observed. The width of the arcs also indicates this characteristic. Nodes with a positive degree are represented by a circle, the ones with a negative degree by a square. A round square is used to represent libraries having a migration degree of zero.

In addition, nodes are colored to represent the intensity of their migration degree. A strong red indicates the worst *breaking bad* of the category, whereas a strong green stands for the best *challenger*. The size of the nodes also reflects this parameter. Finally, the nodes are labeled with the name of the libraries as well as their overall migration degree.

We consider a *Sample* category containing three libraries, *X*, *Y*, and *Z*. The associated library migration graph is exposed in Figure 4. In this case, the library *X* appears as the only challenger, whereas *Y* is the *breaking bad* of the category and *Z* is neutral. We thus observe that the sizes of both nodes *X* and *Y* are higher than *Z*.

6. RESULTS

We now present three library migration graphs with the intent to characterize the libraries in a context of recommendation. We introduce three distinct categories where such characterization is possible. Note that for the sake of readability, the migration flows lesser than 3 are not displayed.

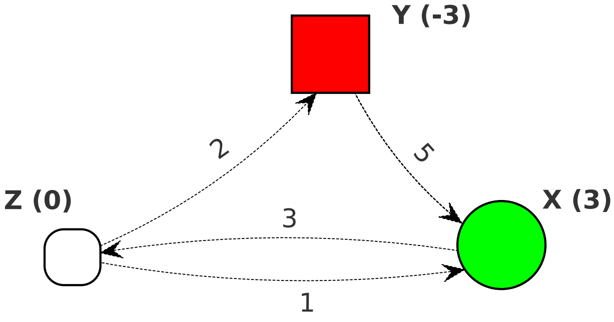


Figure 4. Migration graph for the Sample category.

The first category is *Logging* and contains four libraries used for the logging services. The associated migration graph is shown in Figure 5. We can observe that *slf4j* is clearly the *best challenger* from the category with a significant migration degree of 130, whereas *logback* is a promising challenger with a degree of 19. On the contrary, the two libraries *commons-logging* and *log4j* have both a substantial negative migration degree, consequently, they are strong *breaking bad* and would not be suggested as candidates for migrations.

The second category displays the 13 *Json* libraries that are used to process documents in *Json* format. We can report from the migration graph of this category in Figure 6 that *jackson* and *org.json* are, respectively, the *best challenger* and the *worst breaking bad*. Indeed, their degrees are, respectively, 53 and −50, which is significant. We then observe that *gson*, *fastjson*, and *flexjson* are the three next most important *challenger* libraries, and also that their gap with *jackson* is rather pronounced. Inversely, *json-lib* and *json-simple* should be discarded from the recommendation because of their negative degree. To conclude, we suggest four main *challenger* libraries for the *Json* category, with a particular attention to pay to *jackson*.

The last category is called *Testing* and gathers a list of libraries used during software testing. We expose the related migration graph in Figure 7. We distinguish out of the eight libraries of this category two strong *challenger* libraries and two *breaking bad* ones. The *best challenger* library is *mockito* with a degree of 33, whereas the other *challenger* is *fest* having a degree of 24. Thus, these two libraries are strongly

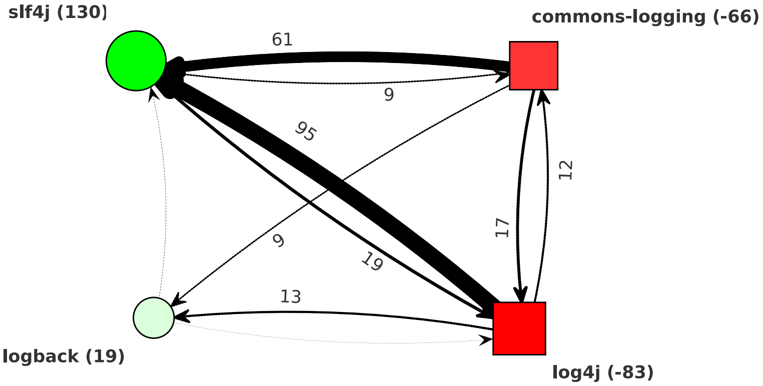


Figure 5. Migration graph for the *Logging* category.

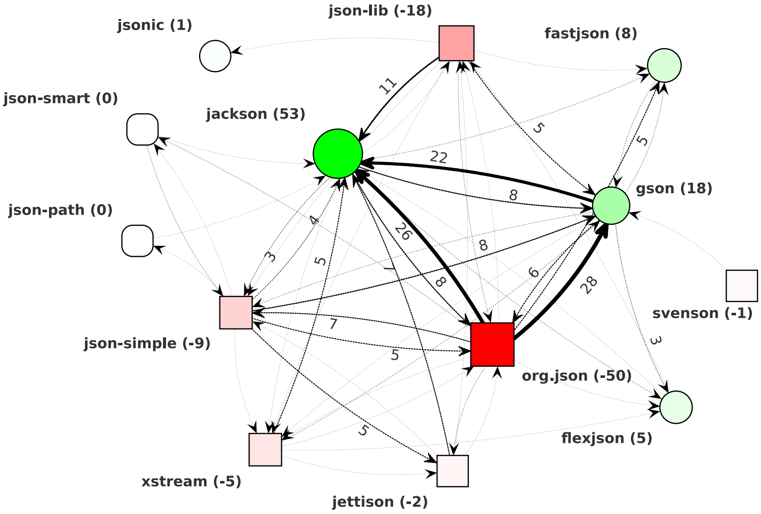


Figure 6. Migration graph for the *JSON* category.

DOI: 10.1002/smr

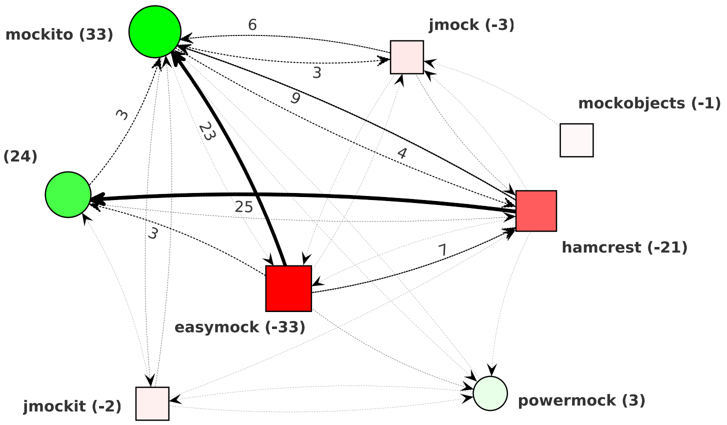


Figure 7. Migration graph for the *Testing* category.

recommended. This is not the case for *easymock*, which is the *worst breaking bad* and *hamcrest*. Indeed, they have an important negative migration degree of −33 and −21, respectively.

We claim the library migration graphs deliver relevant indicators to exhibit the migration flows within a category. In particular, it indicates the pronounced *challengers* and *breaking bad*. However, it should be noted that in categories that contain few migrations (less than 10), the characterization of libraries can be far from obvious.

7. MIGRATION EVOLUTION GRAPH

7.0.1. Motivation. In our opinion, library migration graphs may exhibit biased information about the current state of a library in terms of migrations. Such scenario is due to the time factor. For instance, if a library *X* has a migration degree of 5 and if we observe that it was the target of 15 migrations between 2010 and 2011, but since has been the departure of 10 migrations, we claim that the recommendation of this library is jeopardized. But the opposite situation may also happen. Consider a library *Y* with a migration degree of −5. If *Y* has been departed 10 times before 2011 but selected for migration five times since 2011, we think this recent trend is to its advantage. Consequently, studying the dates and the evolution of the trends is necessary to provide accurate recommendations.

The two questions we are interested to answer here are the following:

- Can we identify *challenger* libraries that have become *breaking bad* recently? (*collapsing* libraries)
- Can we identify *breaking bad* libraries that have become *challenger* recently? (*hopeful* libraries)

Finally, we also measure the impact of the time window for a study of library migration targeting developers recommendations. The idea is to evaluate how different are the recommendations performed using both a large time window and a small one (the current year) for a study of migrations. Similarly, we want to figure out whether studying a recent and small time window can be representative of the trends computed for several past few years. The question we are interested to answer here is

- Does the time window impact the library migration recommendations?

7.0.2. Methodology. We propose a so-called migration evolution graph that is computed as follows. First, it divides the period from 2010 until October 2013 into 4 years. We thus do not consider all the migrations found in our study, but still exclude only a minority of the rules as most of the migrations happened after 2010 (around 94%). For each year, it generates the migration degree for all the libraries of the categories. A black upper triangle indicates a positive degree, whereas a lower down triangle stands for a negative degree. Nothing is displayed in case the degree is equal to zero.

An illustration for the category *Sample* is displayed in Figure 8. Thanks to this chart, we report that the library *X* is *collapsing* because it has current degree of −3 in 2013, despite being a *challenger* on the

DOI: 10.1002/smr

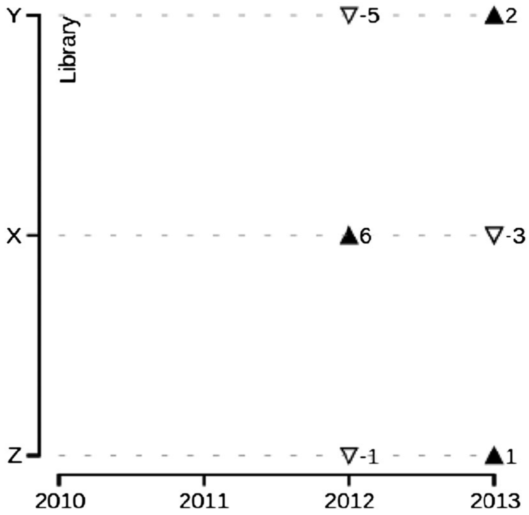


Figure 8. Evolution migration graph for the sample category.

overall period of the study. Inversely, *Y* is *hopeful* because of its positive degree of 2 compared with its overall degree of −3. Finally, *Z* is also an *hopeful* library with a positive degree of 1 over the year 2013.

In a second step, to evaluate the impact of the time window, we simply compute the top-3 *challengers* for a sample of categories and for the two following time windows: the period from 2010 until October 2013, and the year 2013 representing 10 months of data to analyze (and roughly 37% of the migrations from our corpus). In case the ranking highly differs, we will provide explanations to figure out the reasons of the divergence. In they are quite similar, we will say the trends of the current year are representative enough to generate recommendations.

7.0.3. Results 1: collapsing and hopeful patterns. We start by looking for *collapsing* and *hopeful* libraries at the current time of the study. We have reviewed each category and have collected three *collapsing* libraries and one *hopeful* scattered in four different categories. The four related migration evolution graphs are shown in Figure 9.

We report that *sqlite*, *json-smart*, and *jsoup* are the three collapsing libraries, whereas finally *reflections* is the *hopeful* library. We observe in Figure 9a that *sqlite* is currently a *challenger* but its recent trend on 2013 indicates a negative degree of −2, which is not to its advantage. In Figure 9b, we can see that *json-smart* has a degree of zero but can be considered as *breaking bad* because of its degree of −2 in 2013. Similarly, Figure 9c reports that the *jsoup* library features a degree −1 while having a positive degree of 4 during 2012. The collapse is rather low for this library but still exists. Finally, we show in Figure 9d that the *reflections* library records a degree of 3 in 2013, whereas it had a degree of −5 until this period. The trend is thus positive for this library.

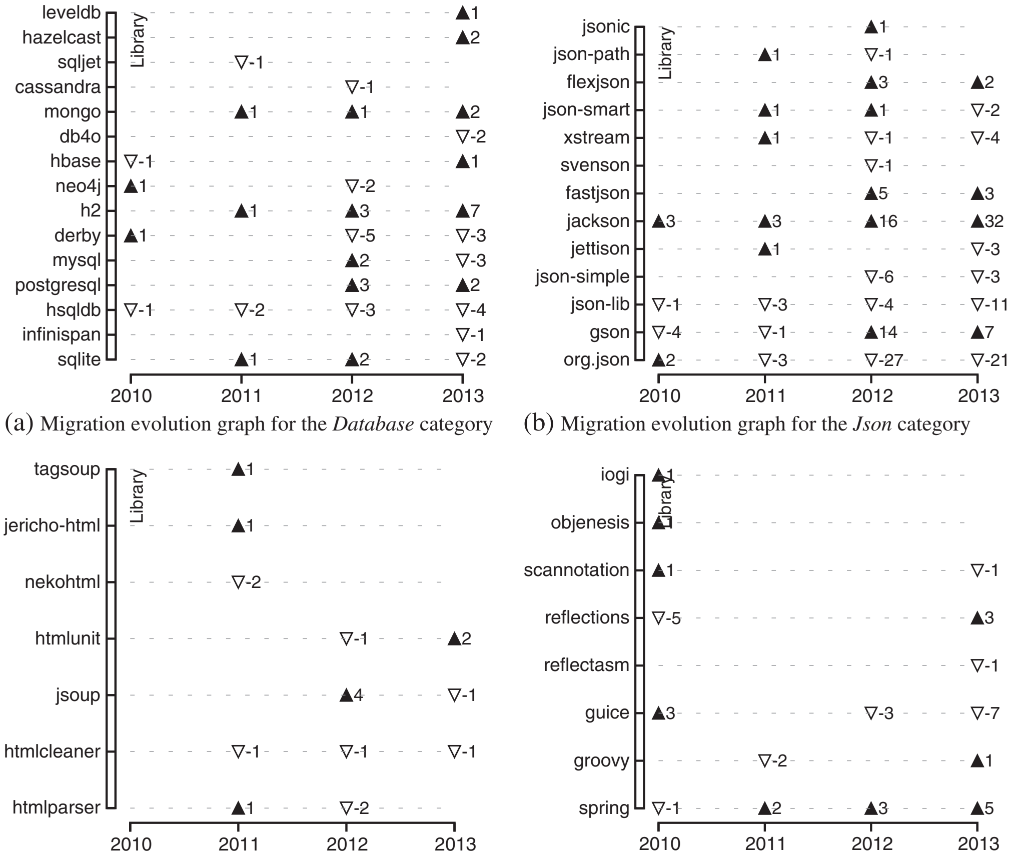
Thus, these two patterns of libraries are not frequent in practice but we claim their semantic provide indicators of interest for developers.

7.0.4. Results 2: understand the time window impact. To further evaluate the impact of the time window, we use the migration evolution graphs of five categories to compute the top three *challenger* in terms of migration degree for two following periods. First, a large time window from 2010 to October 2013, and second a window reduced to the year 2013. The comparison of the computed results is displayed in Table VIII.

We report the following observations. For the *Json* category, the results are entirely similar, so 2013 is a fair representation of the overall trends for this category. We notice a minor change for the *Testing* category, where in 2013, the obvious *best challenger* is *fest* with a migration degree of 17 compared with a degree of 8 for *mockito*. This is explained by the recent increase of this library, which may become the *best challenger* very soon for this category.

The rankings for the *Database* libraries are also similar, because the only difference is the appearance of *hazelcast* in third position, with an equal score compared with *mongo*. The variation is more important for

DOI: 10.1002/smr



(a) Migration evolution graph for the *Database* category

(b) Migration evolution graph for the *Json* category

(c) Migration evolution graph for the *Html* category

(d) Migration evolution graph for the *Reflect* category

Figure 9. Migration evolution graphs to identify *collapsing* and *hopeful* libraries.

Table VIII. Measure of the top three *challenger* libraries according to different time windows of analysis.

		Period			
		(2010–2013)		2013	
Category	Rank	Library	M. degree	Library	M. degree
	#1	jackson	54	jackson	32
Json	#2	gson	15	gson	7
	#3	fastjson	8	fastjson	3
	#1	mockito	29	**fest**	17
Testing	#2	fest	24	**mockito**	8
	#3	powermock	8	powermock	1
	#1	h2	11	h2	7
Database	#2	postgresql	5	postgresql	2
	#3	mongo	4	mongo/**hazelcast**	2
	#1	spring	9	spring	5
Reflect	#2	iogi/objenesis	1	**reflections**	3
	#3	—	—	**groovy**	1
	#1	jsoup	3	**htmlunit**	2
Html	#2	htmlunit	1	—	—
	#3	tagsoup/jericho	1	—	—

the *Reflect* category where we observe the *hopeful reflections* previously introduced. It enters to the second position while being a *breaking bad* during the first period. Finally, it is interesting to observe that only one *challenger* is computed for the *Html* category for the year 2013.

As a conclusion, we remark that the time window chosen to observe the migrations has a small impact on the recommendations that are produced. A reduced time window has the benefit to highlight the most trendy libraries such as *fest*, which has been considerably a target of migration in 2013 compared with *mockito*. However, as library migration is rare, a too small time window can diminish the amount of information computed. Moreover, it can be valuable for developers to understand how the evolution of migration trends over last years. Indeed, it may strengthen or tamper significant trends identified on a recent period (i.e., is this migration constant for 4 years or only over the current year). Therefore, we advocate the need to compute migrations on a medium-to-large time window in order to later tune it according to the developers wishes.

The migration graph, as well as the migration evolution graph, can certainly be improved because they display some redundant information, such as the migration degree. Thus, we could merge temporal information such as the current trend market of each library within the migration graph. However, these enhancements are let for future work.

7.1. Understand developers motivations

7.1.1. Motivation. We think that understanding why developers have migrated their libraries constitutes a relevant source of feedbacks. Thanks to such data, a developer may learn that a library X is not efficient for a given task, and Y should be preferred as it is more powerful for this task. Similarly, she could also learn that a library X is not well compatible with some environments such as modern mobile platforms. It could be also a problem related with the license of the library. Therefore, developers may consider seriously some of these messages if they are in a similar situation.

7.1.2. Methodology. To extract developers motivations, we adopted the following process. For all projects that performed any library migration $X \rightarrow Y$ from our corpus of 329 rules, we queried their commit logs containing either the keywords X or Y from their Git repositories and within their respective migration period. In addition, we textually searched in their issue trackers on GitHub any bug description or comment containing either the keywords X or Y. We then manually reviewed the collected messages for each category of libraries. We discarded so-called 'mood' messages, such as 'X was awful. Y is great.' or '*Switched to a nicer library*', that we observed in practice and that are not meaningful.

7.1.3. Results. We now examine whether developers can better understand the migration trends if people who completed the migration gave explanations about their choice. We collected a total of 1488 candidate messages that we manually reviewed and finally retained 26 messages. It turned out that 19 out of the 26 relevant messages belonged either to the *Loggers* or *Json* categories.

For the first category, the six collected messages are shown in Table IX. We classified two comments with a tag *Configuration* that is not related to the features but the configuration of the library. In two situations, developers mention issues related to dependencies and compilation on systems like Apache Maven. The remaining messages were classified as *Feature*, that is, it targets

Table IX. Reasons of migrations for the *Logging* category.

Migration	Message	Reason
slf4j → commons-logging	'üse commons-logging instead of slf4j to minimize dependencies'	Configuration
log4j → slf4j	'With *slf4j* you can use the following syntax, which I find easier to read: LOGGER.debug("TRUE : {} {} = {} ≥ {}", test, o1, o2, tol)'	Feature
log4j → slf4j	'This removes all compile-time dependencies on log4j'	Configuration
log4j → slf4j	'Migrate to slf4j so users can decide for themselves what kind of logger they are using'	Feature
log4j → slf4j	'It's good practice to use slf4j so people can use whatever logging implementation they like instead of forcing log4j'	Feature
log4j → slf4j	'The whole point of slf4j is to depend on the API only and let the user of the library choose which logging library to use'	Feature

the core features of the libraries. The significant adoption of *slf4j* is partly explained by its design, because it stands as a facade for several implementations of Java loggers such as *log4j* or *logback*, as mentioned by 3 messages. This library success owes a lot to this concept.

For the *Json* category, 13 messages were collected during our extraction and are exposed in Table X. We observe that opinions are sometimes divergent when it comes to compare efficiency and performances of some libraries. However, we can observe that *gson* and *jackson* are seen as the most powerful libraries with efficient features. However, some developers would rather prefer a smaller library to be embedded in their software. The standard *org.json* library faces some performance issues, which may explain its high rate of departures. We also notice a developer complaining of *json-lib*, which has been also observed as an important source of migrations. Finally, it is interesting to observe that one developer had issues *jackson* library on the Android platform. We would thus advice developers to check if there exists known problems of compatibility between this library and the Android platform.

To conclude, we claim that it is worth to understand the motivations of developers that completed a migration. However, only 26 messages were retained from the 1488 messages initially collected, meaning that developers are very cheap on explanations when they complete a migration, and thus tracking this information is far from trivial. We therefore claim it is difficult to extract patterns or trends of motivations for library migrations. Also, we suggest to developers to provide such information when they perform a recommendation as it would be valuable for other software developers.

8. THREATS TO VALIDITY

8.1. *Rules correctness*

One person manually rated each migration rule as either correct or wrong. However, we cannot ensure that all the results were actual, and thus the reviewer may have been wrong. However, if such scenario happened, we think it has a negligible impact on the final data presented in this paper.

Table X. Reasons of migrations for the *Json* category.

Migration	Message	Reason
gson → json-simple	‘now using json-simple instead of gson, since I couldn’t get gson to serialize UUID. Not a perfect library, but it works’	Feature
org.json → jackson	‘örg.json was just way too slow’	Performance
org.json → gson	‘Considering use to parse json files, since it is very simple and it will improve the code readability’	Feature
org.json → gson	‘Move from json.org to gson for conversion of java objects to json’	Feature
org.json → jackson	‘Üsing jackson as json interpreter for dealing with maps’	Feature
org.json → jackson	‘Jackson is 3x as fast, especially on larger lists’	Performance
org.json → jackson	‘Replaced jackson with org.json library for compatibility with android apps’	Feature
org.json → jackson	‘Üsing jackson streaming api for efficiency and scalability’	Feature
jackson → gson	‘Switched to gson. Much, much faster and more stable now’	Performance
gson → jackson	‘gson does not support generic dictionaries like iPhone library, whereas Jackson does’	Feature
json-lib → jackson	‘Replaced json-lib with jackson. jackson is much more flexible and under active development, and json-lib contains some serious issues’	Environment + bug
jackson → json-simple	‘Replace jackson by a json-simple in order to have a small library for Json usage ’	Configuration
json-simple → jackson	‘Even if json-simple doesn’t support streams, there are libraries that are really efficient when it comes to serialize and stream Json (jackson for example)’	Feature

DOI: 10.1002/smr

8.2. *Library set*

Even though the list of libraries *L* used to perform this study has a reasonable size, it only contains libraries that are managed by Maven. Moreover, this set does not take into account the versions of the libraries. Our approach therefore does not consider migrations across versions. Supporting such migrations would first require to identify all the versions of a library and second would require to be able to detect which version of a library a project depends on. These two issues are known to be still open [7].

8.3. *Corpus construction*

The corpus of projects selected for this case study has been built exclusively by querying the GitHub platform. Even though Section 3 presents the different characteristics of the projects, we did not use any rigorous sampling approach to establish the corpus. Thus, the results of our case study cannot be generalized to any existing Java software project. Also, the GitHub platform has been significantly used since 2010.[**] We could have considered platforms like Sourceforge or GoogleCode that contain older projects.

8.4. *Multiple migrations*

The approach proposed in this paper only computes migration rules of cardinality 1 : 1. We argue that rules of cardinality n : m may exist. For instance, when a new project takes over from a no longer maintained library and split the old one into two new ones. This scenario happened in practice. Indeed, the outdated *commons-httpclient* has been separated into two distinct but compatible elements, *httpcore*, and *httpclient*. Our algorithm is in theory extensible to handle such situations; however, it drastically increases the number of candidate migration rules generated.

8.5. *Categories construction*

In our approach, we form categories of libraries using connected components generated by the library migrations. Thus, we claim that each library can be replaced by another one from the same category. However, this technique has a limit that we encountered in this study. There exists libraries offering various set of features, like the Google *guava* library. The problem is that we can migrate from *guava* to libraries like Apache *commons-lang* and *commons-collections*. But in practice, it is not possible to migrate from *commons-lang* to *commons-collections*. The problem comes from *guava* which has been designed to include several domains of services within one library. To that purpose, we will have in a future work to improve how categories are built.

8.6. *Loopbacks and bounces*

Our study does not natively consider loopbacks and bounces. A loopback is a two-steps migration observed toward the life of a project. The project first switches from a library *source* to a library *target*, and later in time moves back to *source*. A *bounce* is a migration of type *x* to *y*, then *y* to *z*, with *x*, *y*, *z* belonging to a same category of libraries.

However, we performed a post analysis of our results to identify loopbacks and bounces. We identified 16 loopbacks distributed in eight different categories. In addition, 10 bounces through seven categories have been observed. Unfortunately, we could not find any reason that explained these choices, especially for loopbacks. We therefore estimate it is not worth to investigate further this aspect of library migrations as it happens very seldom.

The presentation of both our approach and study is henceforth completed. Section 6 next presents the related work before Section 7 concludes and opens the future perspectives.

[**] http://en.wikipedia.org/wiki/GitHub

DOI: 10.1002/smr

9. RELATED WORK

We divide the related work in four parts. We first describe existing research on the library migration problem and then discuss on the library update problem. Third, we present related work on general usage of libraries in software projects. We finally conclude by presenting existing work on the software categorization problem.

9.1. Library migration

During a library migration, the API-level challenge is to transform the source code so that it becomes compliant with the new library. Bartolomei *et al*. have addressed this problem and studied the design of Application Programming Interface (API) Wrappers, which are objects that adapt and delegate the previous source code instructions towards the new API [2, 3]. The mappings are manually identified and their concern is to design such wrapper in order to obtain a compliant version of the new source code. Our approach is useful for such a problem as it identifies which libraries are source and target of migrations. In a previous work, we proposed an approach to automatically extract similar functions from two independent libraries in a context of migration [4].

Library migration has also been used in the context where a library is available in two different programming language. In that direction, Zhong *et al*. proposed a Mining API Mapping approach that detects relations between two versions of such APIs [8]. The idea is to get client-code from the two versions and to build a transformation graph that represents the API-usage migration from one language to another. Zheng *et al*. propose a cross-library recommendation tool based on Web queries [9]. The idea is to inquire Web search engines and to mine results proposed from the query. One example of query could be ‘HashMap C#’ when looking for the equivalent for standard Java HashMap for C#. The results are computed one by one and candidates are ranked by relevance, mainly according to their frequency of appearance. For the moment, this work provides only preliminary results and queries proposed are coarse grained. Also, it strongly relies on Web search engines such as Google, and requires manual query writing, which can highly influence the results.

The library migration problem has also been studied in a context where the system used differs. Winter *et al*. shown that Java Libraries that run on the standard Java Virtual Machine (JVM) may be incompatible on JVM running on embedded systems [10]. They propose program transformation to eliminate parts of the API that are incompatible on such systems, like floating points operations or multi-threading. This approach is nonetheless not enough related to our context to provide elements of comparison.

9.2. Library update

The problem of updating a library has also been studied in the literature. A challenge with regard to library update is to provide relevant snippets of code source according to the programmer’s context. We distinguish two main techniques to that extent. The first one mines code that already performed an update. For instance, Schafer *et al*. [11] examined code instantiations of two versions of a framework. This code is included with the release as test or example code. Also, SemDiff [12] is a client–server connected to a framework source code repository that mines the changes and recommends modifications for a client migration.

The second variety of approach requires only internal code of two API versions and applies origin analysis techniques. Some promising results have been achieved in this area [13] [14]. Whatever the technique, our approach can be used as a massive source of data to get real library migrations and to get references of real projects that do have performed migrations. Such quantity of data could be used to validate the proposed approaches.

It should be noted that a recent study from Cossette performs a retroactive study on several library changes performed manually [15]. They listed the different changes and adaptations they had to make. They argue that existing automatic approaches such as the ones presented earlier are not satisfying enough to cover all the case of the library update problem. We claim that if the library update still faces challenges to be accomplished fully automatically, completing a library migration is even more difficult, therefore automatizing this operation is even more complex.

DOI: 10.1002/smr

9.3. General use of third-party libraries

Mileva *et al.* have observed the evolution over 2 years of the dependencies of 250 Apache projects managed with the build automation tool Maven [16]. They analyzed the Maven configuration files of these projects to mine usage of API and their versions. The study shows the usage trends of different versions of several libraries. This work points out interesting cases where clients switched back to a previous version of a library they are using. Little work has been done on third-party library recommendation. To our knowledge, only Thung *et al.* addressed this problem and built a recommendation system using data mining over a large set of existing Java projects managed under Maven [17]. It inputs a set of libraries and outputs the ones that are the most commonly used along them from the observation of the initial corpus of projects. Lämmel *et al.* propose a large-scale study of AST-based API-usage over a large set of open-source projects [18]. Their work provides an insight on how a specific API is globally used by client projects. In particular, they categorize whether a client calls the API (*library-like usage*) or extends it (*framework-like usage*). It may be interesting to integrate such information in our library migration graphs as some libraries may be more appropriate than others, depending on client usage requirements. Robillard *et al.* investigate the obstacles met by developers when learning an API [19]. Their study points out the lack of documentation or learning resources, which in our opinion can intervene in a migration context.

9.4. Software categorization

An orthogonal research area to our problem is the software categorization problem that aims to identify similar software. This problem is usually solved by computing similarity score based on specific attributes, such as keyword identifiers as MudaBlue [20] does, or API calls [21, 22]. Those techniques require either the source code or the binaries versions of a set of libraries to compute similarity scores among them. Thung *et al.* suggest to use collaborative tagging on platforms such as SourceForge in order to improve precision of existing approaches [23]. More recently, Wang *et al.* proposed an approach to assign tags to software projects using mining of existing projects tags and descriptions [24]. These approaches can be used in our context to create groups of equivalent libraries but without any guarantee on the fact that a library of a group can be replaced by any other equivalent library of the group. We therefore choose to use migration graphs to create categories of similar libraries.

10. CONCLUSION

As software systems intensively depend on external libraries, software developers must think about migrating libraries when they are not updated, or when competing ones appear with more features or better performance, for instance. In this paper, our purpose is to help software developers who are thinking about replacing their dependent libraries and who do not know which are the best libraries to migrate to.

We then provide recommendations to help developers to choose among a set of candidate libraries to migrate to. To that extent, we first present a generic approach to identify library migration trends from a corpus of software projects. Second, we have made a complete analysis of the library dependency and migration phenomena. Third, based on this complete analysis, we provided recommendations to highlight best challenger libraries to migrate to as well as worst candidates.

To realize our analysis and to provide our recommendations, we have designed an approach that automatically identifies library dependencies and semi-automatically library migrations. Our approach relies on a static analysis of the source code to identify library dependencies. To extract library migrations, it mines VCSs to observe migrations and requires a manual review to clean wrong observations. Our approach has been completely prototyped and currently only supports Java software projects. We applied our approach over 20 000 Java projects from the GitHub platform and have identified 329 migration rules and classified the 164 involved libraries within 32 distinct categories.

By analyzing this large set of projects, our intent was to better understand what are the software projects that do depend on libraries and what are the ones that do perform migrations. Our analysis has shown that Biggest projects depend on more libraries than Smallest projects. It also appears that several libraries are only used by Biggest projects. Further, only a small set of libraries are very famous and widely used by

DOI: 10.1002/smr

most of the projects (whatever their size). Regarding migrations, our analysis shows that projects do migrate but it is occasional. Further, Biggest projects perform more migrations than Smallest projects. Finally, our analysis has shown that libraries can be grouped into categories of similar libraries in order to exhibit trends of migration.

The recommendations we propose are based on categories of libraries and focus on two main factors: the technical domain covered by the libraries of a category and the date of the migrations. These two factors are used to identify challenger libraries to migrate to as well as libraries that should be avoided. The results of our study show that our approach is exploitable for software developers who are looking for library recommendations.

Currently, the major limitation of our approach is its lack of precision. The main purpose of our work on detecting library migrations should be replicable to compute periodically the migration trends, that is why we plan to design an automatic verification of the correctness of a migration rule, using for instance a classifier. The main purpose is to remove the current burden of the necessary manual analysis required to clean the rules. Another limitation of our approach is the fact that it does not support versions of library. As a consequence, an update of a library is not considered to be a migration in this study. Supporting versions is highly complex as software projects almost never define which versions of the libraries they depend on. To obtain such an information, an analysis of the runtime dependencies must be performed, which is still an open issue.

As a future work, we would be interested in performing a controlled experiment with developers that want to perform a migration to check whether their decision is influenced when possessing migration trends data and the charts we introduced. For instance, we would want to check that our library migration graphs can be used to identify libraries to migrate to. Also, we would be interested to identify what time window of migration trends developers are more interested in.

We also plan to extend our approach to assist developers while they migrate their code to become compliant with a new library. As our approach identifies software projects that already performed migrations, we plan to analyze the source code of these projects before and after the migration in order to detect migration patterns. Such patterns abstract refactoring actions that must be performed to be compliant with the new library. The goal is then to automatically apply them in software projects that want to perform the same migration.

REFERENCES

1. Baldassarre MT, Bianchi A, Caivano D, Visaggio G. The question we are interested to answer here is. An industrial case study on reuse oriented development. Proceedings of the 21st IEEE International Conference on Software Maintenance. ICSM '05, IEEE Computer Society: Washington, DC, USA, 2005; 283–292, doi:10.1109/ICSM.2005.20.
2. Tonelli Bartolomei T, Czarnecki K, Lämmel R, Storm Tvd. Study of an API migration for two XML APIs. *2nd International Conference on Software Language Engineering* (*SLE*), vol. 5969/2010, Denver, USA, 2009; 42–61, doi:10.1007/978-3-642-12107-45.
3. Tonelli Bartolomei T, Czarnecki K, Lämmel R. Swing to SWT and back: Patterns for API migration by wrapping. *26th IEEE International Conference on Software Maintenance* (*ICSM*), Timisoara, Romania, 2010.
4. Teyton C, Falleri JR, Blanc X. Automatic discovery of function mappings between similar libraries. *20th Working Conference on Reverse Engineering 2013*, *14th-17th October 2013*, *Koblenz*, *Germany*, IEEE, 2013; 192–201.
5. Teyton C, Falleri JR, Blanc X. Mining library migration graphs. *19th Working Conference on Reverse Engineering 2012*, *15th-18th October 2012*, *Kingston*, *Ontario*, *Canada*, IEEE (ed.), Kingston, Ontario, Canada, 2012; 289–298, doi:10.1109/WCRE.2012.38.
6. Falleri JR, Teyton C, Foucault M, Palyart M, Morandat F, Blanc X. The harmony platform. *Technical Report*, Univ. Bordeaux, LaBRI, UMR 5800 Sep 2013.
7. Davies J, Germán DM, Godfrey MW, Hindle A. Software bertillonage: finding the provenance of an entity. *Proceedings of the 8th International Working Conference on Mining Software Repositories*, *MSR 2011* (*Co-located with ICSE*), *Waikiki*, *Honolulu*, *HI*, *USA*, *May 21–28*, *2011*, *Proceedings*, IEEE, 2011; 183–192.
8. Zhong H, Thummalapenta S, Xie T, Zhang L, Wang Q. Mining API mapping for language migration. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10* 2010; **1**:195, doi:10.1145/1806799.1806831.
9. Zheng W, Zhang Q, Lyu M. Cross-library API recommendation using web search engines. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ESEC/FSE '11, ACM: New York, NY, USA, 2011; 480–483, doi:10.1145/2025113.2025197.
10. Winter VL, Mametjanov A. Generative programming techniques for java library migration. *Proceedings of the 6th international conference on Generative programming and component engineering*. GPCE '07, ACM: New York, NY, USA, 2007; 185–196, doi:10.1145/1289971.1290001.

11. Schäfer T, Jonas J, Mezini M. Mining framework usage changes from instantiation code. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008; 471, doi:10.1145/1368088.1368153.
12. Dagenais B, Robillard M. SemDiff: analysis and recommendation support for API evolution. *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*, 2009; 599–602, doi:10.1109/ICSE.2009.5070565.
13. Wu W, Guéhéneuc YG, Antoniol G, Kim M. AURA: a hybrid approach to identify framework evolution. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10, ACM: New York, NY, USA, 2010; 325–334, doi:http://doi.acm.org/10.1145/1806799.1806848.
14. Nguyen HA, Nguyen TT, Wilson Jr G, Nguyen AT, Kim M, Nguyen TN. A graph-based approach to API usage adaptation. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '10, ACM: New York, NY, USA, 2010; 302–321, doi:10.1145/1869459.1869486.
15. Cossette BE, Walker RJ. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12, ACM: New York, NY, USA, 2012; 55:1–55:11, doi:10.1145/2393596.2393661.
16. Mileva YM, Dallmeier V, Burger M, Zeller A. Mining trends of library usage. *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. IWPSE-Evol '09, ACM: New York, NY, USA, 2009; 57–62.
17. Thung F, Lo D, Lawall J. Automated library recommendation. *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013; 182–191, doi:10.1109/WCRE.2013.6671293.
18. Lämmel R, Pek E, Starek J. Large-scale, AST-based API-usage analysis of open-source java projects. *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, 2011; 1317, doi:10.1145/1982185.1982471.
19. Robillard MP, Deline R. A field study of API learning obstacles. *Empirical Softw. Engg* 2011; **16**(6):703–732, doi:10.1007/s10664-010-9150-8.
20. Kawaguchi S, Garg P, Matsushita M, Inoue K. MUDABlue: an automatic categorization system for open source repositories. *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004; 184–193, doi:10.1109/APSEC.2004.69.
21. McMillan C, Linares-Vasquez M, Poshyvanyk D, Grechanik M. Categorizing software applications for maintenance. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011; 343–352, doi:10.1109/ICSM.2011.6080801.
22. McMillan C, Grechanik M, Poshyvanyk D. Detecting similar software applications. *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12, IEEE Press: Piscataway, NJ, USA, 2012; 364–374.
23. Thung F, Lo D, Jiang L. Detecting similar applications with collaborative tagging. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012; 600–603, doi:10.1109/ICSM.2012.6405331.
24. Wang T, Yin G, Li X, Wang H. Labeled topic detection of open source software from mining mass textual project profiles. *Proceedings of the First International Workshop on Software Mining*. SoftwareMining '12, ACM: New York, NY, USA, 2012; 17–24, doi:10.1145/2384416.2384419.

AUTHORS' BIOGRAPHIES

Cedric Teyton is currently a Phd student in the LaBRI Software Engineering research group and supervised by Xavier Blanc. He received his master's degree from the University of Bordeaux and worked in his master thesis on the language VPraxis designed to query a software history. His research interests include software maintenance, API evolution and software repositories mining.

Jean-Rémy Falleri is currently an associate professor at the Bordeaux Institute of Technology and a member of the LaBRI Software Engineering research group. He received his PhD degree in 2009 from the University of Montpellier 2. He has also worked in the RMoD research group of Inria Lille led by Stéphane Ducasse. His research interests include software engineering, software maintenance and model driven engineering.

DOI: 10.1002/smr

Marc Palyart is a research assistant in the LaBRI Software Engineering research group. He holds a BSc (Hons) from the Dundalk Institute of Technology and an MSc and PhD from the University of Toulouse. His research interests include software engineering, model-driven engineering, high-performance computing and smart environment.

Xavier Blanc is currently full professor at the Bordeaux 1 University. His current research is about software evolution and repository mining. Since 2011, he is deputy director of the computer science laboratory (LaBRI) of the Bordeaux 1 University.

DOI: 10.1002/smr