Find your Library Experts

Cédric Teyton, Jean-Rémy Falleri, Floréal Morandat and Xavier Blanc
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
E-Mail: {cteyton,falleri,fmoranda,xblanc}@labri.fr

***Abstract*—Heavy usage of third-party libraries is almost mandatory in modern software systems. The knowledge of these libraries is generally scattered across the development team. When a development or a maintenance task involving specific libraries arises, finding the relevant experts would simplify its completion. However there is no automatic approach to identify these experts. In this article we propose LIBTIC, a search engine of library experts automatically populated by mining software repositories. We show that LIBTIC finds relevant experts of common Java libraries among the GitHub developers. We also illustrate its usefulness through a case study on the Apache HBase project where several maintenance and development use-cases are carried out.**

I. INTRODUCTION

Open Source Software communities become larger and turn to be real social networks. One famous example is GitHub that not only provides a dedicated development environment for each of its projects but also offers global facilities to all of its members allowing them to communicate together and to exchange about their software practices [1].

In such software social networks any member has a role that depends on the contributions she makes. Ye and Kishida have defined the roles (project leaders, core developers, active developers, bug fixers, etc.), which have a major impact on the structure of the community and then on software quality [2]. In particular, Robles et al. have observed that it is recommended for an OSS project to keep its core developers during its lifetime [3].

Even if these roles are primary to measure how a software project is organized, there are other roles, such as technological guru or expert, that worth to be defined for measuring the technological strengths or weaknesses of a community or a project and for optimizing the technical communications that occur between its members. Further, by knowing the experts of a given technology, any developer can think about being connected to them to follow their advice or to directly contact them to solve an issue.

In this paper we propose to automate the identification of experts within OSS communities. As there exist several domains of expertise, we choose to focus on third party library experts. Our motivation is driven by the fact that software nowadays depends on many third party libraries [4]. For instance, Apache Struts depends on 35 libraries and Hibernate on 42. As a consequence, software developers must master the libraries used by the project they contribute to, otherwise they will not be able to ensure their maintenance.

Our contribution is twofold. First we propose an abstract model for library expert identification. This model is based on a syntactical analysis of the commits performed by the developers to measure their library usages. The main idea is that a developer is an expert of a library if she introduces source code that uses this library. Our second contribution is a query language that can be used to identify library experts within large communities.

Our contributions come with a tool, named LIBTIC, that is intended to be used by project leaders in at least the three following situations. First LIBTIC can be used to evaluate the library knowledge among the developers of a given project. Second it can be used to identify developers of a given project that are experts of a given library with the objective to ask them to perform tasks that require such an expertise. Third it can be used on a whole community to identify if there are experts that can be contacted to answer some very specific questions about libraries.

The remainder of this paper is structured as follows. Section II presents the abstract model of our approach. Section III describes our implementation to extract and query developer expertise. Section IV presents how we used our implementation to validate our approach. Related work is presented in Section V. Finally perspectives and conclusion seal the paper.

Our tool and data are freely available at:

http://www.labri.fr/~cteyton/libtic/

II. LIBRARY USAGE

This section starts by providing definitions for library and developer usage. Based on these definitions it then presents how expertise is measured and how library experts can be identified.

A. Library and developer usage

A software library is a software component that provides facilities designed to be invoked through well defined interfaces. For the sake of simplicity, we consider that each facility provided by a library (such as a function, a type or a class) is identified by a syntactic symbol.

Definition 1 (Library): Let L be the set of libraries. A library $l \in L$ has an associated set of exported symbols S_l that identify all facilities provided by the library. The notion of exported symbol depends on the considered language. For instance in Java, the exported symbols are the public types, methods and attributes.

Software developers contribute to the source code by frequently committing their changes. We consider that a developer uses a library if one of her commits introduces

at least a symbol of the library. Moreover, we also track the version of the library that is being used by the developer. For the sake of simplicity, we consider that versions of libraries are totally ordered.

Definition 2 (Developer usage): Let D be the set of developers. A developer $d \in D$ uses a symbol $s \in S_l$ of a library $l \in L$, if one of the commits she performed introduces s in the source code. The version of library used at the time of the commit is noted v. We note (d, l, s, v) the tuple that defines the usage of a library by a developer. We note U the union of all the usages of libraries performed by all the developers that contribute to a set of software projects.

As an example, we consider that L contains one library h2 that proposes the symbols $load$, $start$ and $stop$. *Alice*, a software developer, performs a commit introducing $load$ from the version 1 of h2 and another commit that introduces $start$ from the version 2 of $h2$. The developer usage U is then noted $U = \{(alice, h2, load, 1), (alice, h2, start, 2)\}$.

B. Library expertise

We propose to identify library experts according to library dimensions. A library dimension defines some requirements on an expertise. For our concern, a library dimension must target one library and can optionally define some filters that limit the set of symbols and/or versions of the library.

The identification of experts can be done against one or several library dimensions and will return experts that have used the symbols defined by the dimensions. With our example, we can define a library dimension that targets the h2 library and that considers all of its symbols and all of its versions. Changes performed by *Alice* will then be considered when identifying experts of h2. We can define another dimension that targets the h2 library but that considers only the version 2. For such a dimension, only the second usage of *Alice* will be considered.

Definition 3 (Library dimension): An expertise dimension (l, dim_s, dim_v) targets one library $l \in L$ and optionally defines a filter function for symbols $dim_s : S_l \rightarrow \mathbb{B}$ and a filter function for versions $dim_v : V_l \rightarrow \mathbb{B}$ of the library. We note Dim the set of dimensions.

To measure the expertise that a developer d has regarding a given library dimension dim, we compute the ratio between the number of symbols he uses ($u(d, dim)$) with the total number of symbols provided by the dimension ($p(dim)$).

Definition 4 (Symbols usage): Given a library dimension $dim = (l, dim_s, dim_v) \in Dim$, the set of symbols of l used by a developer $d \in D$ is:

$$u(d, dim) = \{s \in S_l | \exists v (d, l, s, v) \in U \wedge dim_s(s) \wedge dim_v(v)\}$$

Definition 5 (Provided symbols): Given a library dimension $dim = (l, dim_s, dim_v) \in Dim$, the set of provided symbols is:

$$p(dim) = \{s \in S_l | dim_s(s)\}$$

The expertise a developer has regarding a given library dimension is a real value between 0 and 1 that expresses the percentage of used symbols among the provided ones.

Definition 6 (Library expertise): The expertise that a developer $d \in D$ has for a given library dimension $dim = (l, dim_s, dim_v) \in Dim$ is:

$$e(d, dim) = \frac{|u(d, dim)|}{|p(dim)|}$$

Our approach takes into account the number of symbols used by each developer and not the number of times they are used. We argue that as soon as a developer has introduced a symbol in the code, he can be considered to know the symbol. As a consequence, we argue that the number of known symbols is a better indicator that the number of times they are used to identify experts. For instance a developer may use several times the same pattern while alternative, i.e. other symbol, where available. Moreover taking into account duplicates would not lead to single number between 0 and 1 thus would be harder to interpret. Further our approach considers that all symbols of a library have the same importance. This obviously does not correspond to the reality as all libraries have important symbols as well as minor ones. However our approach aims to be automated and independent of any library, and therefore cannot assign weight to library symbols.

With our example, $e(alice, h2) = 2/3$ as *Alice* uses two of the three symbols provided by h2. If now we consider that *Bob* has introduced $stop$ from the version 1 of h2 in a commit and $stop$ from the version 2 of $h2$ in another commit, then $e(bob, h2) = 1/3$ as *Bob* uses only one symbol of h2.

C. Expertise distance

To compare the difference of expertise among developers, we use the classical Euclidean distance. To that extent, we define the reference point $\top$ that corresponds to a complete expertise for every library dimension ($\forall dim \in Dim,\ e(\top, dim) = 1$). For each developer considered by the comparison, we then compute a vector of expertise. With our example, we now consider a second library slf4j where $e(alice, slf4j) = 1/2$ and $e(bob, slf4j) = 1$. If we consider the dimensions that target h2 and slf4j without any filters, we then have the following vector $v_\top = (1, 1)$, $v_{alice} = (2/3, 1/2)$ and $v_{bob} = (1/3, 1)$. Using the Euclidean distance, we obtain that *Alice* is again a better expert as her distance with $\top$ is 0.60 whereas it is 0.66 for *Bob*.

III. LIBTIC PROTOTYPE

This section presents LIBTIC that implements our approach. The first component of LIBTIC aims at building the developer usage database (DUD) that contains all usages of libraries made by developers (all $u \in U$ as defined in Section II). The second component of LIBTIC executes the identification of library experts. This component is a kind of query engine, where queries specify the library dimensions of the identification. Figure 1 presents the user interface of this component where queries can be defined and where the list of corresponding experts are returned.

While our approach is not restricted to any particular language, LIBTIC is only a proof of concept and hence has some restrictions. Currently it only supports projects in the Java language. Projects also need to be managed through Apache

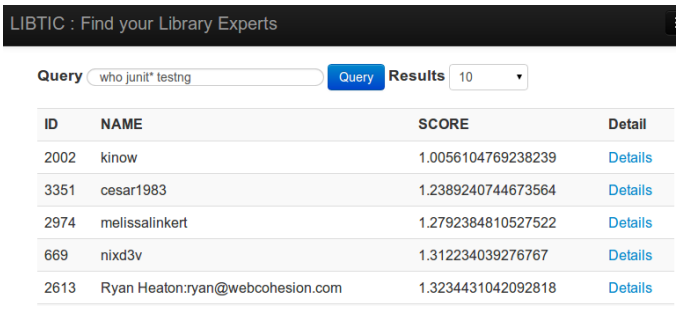

Fig. 1: Screenshot of LIBTIC.

Maven. Finally projects should be stored on a version control system such as Git or SVN.

A. Building the developer usage database

Building the developer usage database (DUD) requires two components: a *symbol index* which maps symbols of a project to a library and a version, and a *usage extractor* which collects the developer usage of a project.

1) Symbol index: The symbol index aims at identifying which library defines a given symbol and at which version. It is a function which maps symbols either to both a library and a version or to $\emptyset$ if it cannot identify the library or the version. To build this index, we have performed a two step process. The first step consists in analyzing all libraries in all their versions to build the complete set of symbols they provide. This step ends with a huge set of tuples (s, l, v) indicating that the symbol s is provided by the library l at version v. The second step aims at transforming this set to an index where symbols are the keys and where (l, v) are the values.

As many libraries provide similar symbols at different versions and as it may appear that different libraries provide same symbols, it is difficult to assign each symbol to only one library at one version. To overcome this issue, we decided to use dependency information from the project where the symbol is being found. This dependency information ($deps_p$) is a set that contains all the couples (l, v) on which the project depends. As a consequence, the keys of the symbol index are now couples $(s, deps_p)$. The index then only retains tuples (s, l, v) where $(l, v) \in deps_p$. Finally if s is unique among the remaining tuples, its corresponding tuple is returned. Otherwise, if s does not exist in the index or if it is contained in several tuples, the index returns $\emptyset$.

For our Java prototype we define symbols as fully Java qualified names of methods including the types of all arguments. This allows to deal trivially with method overloading. Even though this is not perfect, fully qualified names of methods avoid most of names conflict of symbols coming from different libraries. On the other hand method overriding is ignore since this task fall to the library designer.

The first step of our process has been populated by extracting symbols of libraries managed by the Maven repository, which stores a huge number of libraries at many versions. The jar files of all the libraries have been downloaded and symbols have been automatically extracted by using the *Javassist* bytecode analyzer [5]. This process is straightforward since the bytecode contains already fully qualified names. In practice only public methods were extracted. For the second step of our process, the index has been computed thanks to information provided by Maven configuration files that defines the library dependencies of each project.

2) Usage extractor: The usage extractor is the keystone of the DUD extraction process. It assumes that for any project version, dependencies are known, i.e. library names and versions[1], and that the symbol index is built accordingly.

For any given project it analyzes each version recorded in the VCS in any order. Files reported as added or modified by the VCS are further analyzed. Deleted files are discarded as they cannot be a location of a symbol introduction. In particular, we consider that a symbol is used only when it is first introduced in a particular location of a source code file. Each added or modified file is then parsed, resulting in an abstract syntax tree (AST) that contains nodes having a type, a value (text), and a position in the source file. This step works for any language where a parser exists. Then a text *diff* is computed between each analyzed file with all of its parent versions.

As a final step the usage extractor walks through AST nodes that refer to a symbol. Obviously this knowledge is language dependent and a *node filter* has to be provided for each programming language. Whenever such a node is found and if its position is in the scope of added lines (+) in all the diffs, the value of the node is searched in the symbol index.

Finally, each symbol s that maps to a library l at the version v is added in the DUD as a usage $(committer, l, s, v)$, while each symbol mapping to $\emptyset$ is discarded.

This whole process can be applied incrementally, processing new projects or revisions as they appear to keep the DUD up-to-date.

In our prototype, we rely on the Eclipse Java Development Tools (JDT) for the Java parser and the node filter.[2] JDT is capable of extracting an AST from a Java source code. Unfortunately symbols in the Java source code are usually referred using a short name while fully qualified names are required for the index. Even if sometimes qualified names of symbols can be deduced from the import statements, it is not possible when two or more imports use wildcards. Moreover, JDT allows to resolve names as long as the Java classpath is configured correctly, which can be easily done thanks to Maven. After source code parsing, the Java node filters accept only the following nodes from the AST: class definitions, field definitions, method definitions, variable definitions and method calls.

3) Limitations: While being mostly language agnostic, this approach has a few limitations. Some are due to the simplicity of the prototype, others are more open research questions.

a) Libraries and versions: As previously explained we assume that we know for any project revision the exact libraries and versions they use. While this is fairly easy to retrieve when a dependency manager is used, it can be very difficult for a project that does not use any automated dependency management. To have a symbol index that works without any

[1] If the version is unavailable, a wildcard is used.
[2] http://www.eclipse.org/jdt/

information on the project dependencies, we could investigate techniques to detect automatically the provenance of an entity, such as [6].

b) False symbol introduction: We obtain symbol introductions by applying a text differencing algorithm (described in [7]) between two successive versions of a source code file. However this algorithm is not robust to several situations. The symbols contained in the following elements are seen as introduced: a renamed source file, a copy-pasted code snippet, a code snippet moved across files. In these cases, a false symbol usage is added in the DUD. However the number of these false positives could be reduced by using origin analysis techniques (such as [8]) or more powerful differencing algorithms. While the technique is proposed with text differencing, it is not restricted to it, any other technique which is able to provide added or removed symbol should work, e.g. structural differencing [**?**].

c) Multiple identities: When committing a revision to a VCS, users can usually choose freely their names. It is therefore not uncommon that a same developer has several distinct names on the repositories he commits on. This phenomenon can decrease the quality of the results obtained by our tool. Therefore we highly recommend to apply an identity merging algorithm on the names (such as the ones explained in [9]) gathered from the VCS.

B. Querying the model

Collecting data is only a first step, analyzing it efficiently and easily represents another challenge. The DUD is stored in a relational database, however we remarked that using SQL to query it leads to cumbersome, hard-to-write and error prone queries. Since being able to quickly find the library experts is the most important feature of our tool, LIBTIC introduces a simple domain specific language (DSL) over SQL to query the database. Even if this DSL is only composed of two main operators, it is powerful enough in a lot of use cases (Section IV). Additionally a few more constructs are added to our DSL to ease data manipulation. Without being exhaustive this section contains a quick overview of the LIBTIC DSL that should be enough to reproduce the results provided in Section IV of this article.

Who, who, how and how*:* Quickly finding library experts enforcing a set of constraints is the main purpose of LIBTIC. Therefore writing such a query should be as short as possible. This is the goal of the `who` operator, which is the main operator of our DSL. This operator takes as a mandatory argument a set of *library constraints*, each representing a library dimension. It returns all the developers having a distance strictly lesser than 1 on every dimension, ranked by their distance to the set of library dimensions. A library constraint is expressed as a library name, an optional version filter and an optional symbol filter. A symbol filter is a set of simplified regular expressions. A symbol matching any of the simplified regular expressions is retained by a filter. For instance the query `who guava {*.html.* *.io.* } guava{*.math.*} guava >4` returns all developers knowing at least the version 4 of guava and one or more symbols in the io or html packages and one or more symbols from the math package. The only special characters of our simplified regular expressions are the classical `*` and `?` shell wildcards, with their usual semantics. The `who` operator also accepts a simplified regular expression as a library name. In this case, all libraries matching this pattern are aggregated as if they were the same library. Additionally, an optional set of simple regular expressions can be given as a first argument to the `who` operator. It will restrict the results of the query to developers having a name matching at least one pattern. For instance the query `who {alice bob*} guava{*.math.*}` returns the developers that know at least one symbol of the math package of the guava library and that have *alice* as name or any name starting by *bob*.

Since `who` returns developers having a distance lesser than 1 for all dimensions, it might be too restrictive in some situations. The `who*` can be used to relax constraints by returning developers having a distance lesser than 1 in at least one dimension of the query. It can be noticed that this starred version returns a superset of the classical `who`, and when a developer is returned by both query the distance remains the same.

Since in several cases it is interesting to know the distance of the developers for each library instead of the distance to the query, we have also defined the `how` operator and a starred version. These operators consider each library as independent for the computation of the distance. Finally both `who` and `how` have the same syntax.

Describe and find: Another common operation performed in LIBTIC is searching for details of libraries, symbols or developers. This is why we introduced the `desc` and `find` operators. `desc` is used to consult the details of a particular user. Basically it returns the symbols he used together with their version and libraries. Results may be restricted to some libraries by appending library constraints (with the same syntax than `who`). `find` is used to look for the exact name of libraries, symbols or developers. All these operations are done with `find user|lib|sym` followed by a name which may contain wildcards. Additionally symbols lookup may also be restricted to a set of libraries, e.g., `find sym *.io.* guava*`.

Variables: Finally, a simple mechanism of variables is provided by this language to deal with complex or repetitive constraints. Variables can represent constraints (`@var`), or set of users or symbols (`$var`). Both are stored in different namespaces and may contain wildcards which will be expanded in their respective context, e.g., the snippet `set $o {*o*} ; set @g guava ; who $o @g $o` returns only developers having a "o" in their names and knowing a symbol of guava which also contains a "o".

IV. EVALUATION

This section presents the two experiments we made with LIBTIC. The first one aims to demonstrate that LIBTIC does identify library experts. The second one aims to highlight how LIBTIC can be used by project leaders to manage their software projects. These two experiments have been performed on a corpus of software projects that has been extracted from GitHub. To respect the privacy of the GitHub developers, the results of all our experiments contain fake developers names.

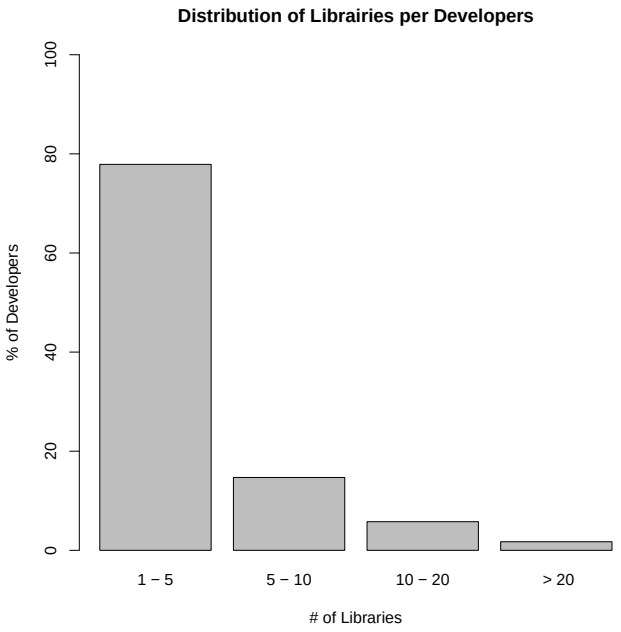

Fig. 2: Distribution of the number of known libraries per developer

A. Extracting the corpus from GitHub

Our two experiments have been done on a large developer user database (DUD). We choose to extract this DUD from the GitHub software project hosting platform because GitHub contains a huge amount of software projects and developers and because it defines an easy-to-use REST API[3].

To that extent, we queried the REST API of GitHub to get a list of Java projects using Maven and managed to collect 6330 projects. As explained in Section III, for each project we downloaded all of its library dependencies using Maven to populate the set of libraries and symbols. Then, we browse the source code of each project, revision by revision, to populate the set of developers and the DUD.

Since GitHub developers are free to give any name when they commit some source code, they can appear with several names. To reduce the fragmentation of the developer names explained in Section III, we applied the following simple identity merging technique, relying on the GitHub REST API. We search for the committer name in GitHub and if GitHub returns a user we take its login, if not we take the committer name.

Collecting libraries and symbols of our corpus has taken about 50 hours. Populating the developers and symbol usage has taken about 150 hours. The DUD contains 3705 authors, 1026 libraries, 51585 symbols and 161917 symbols usage. Figure 2 shows the distribution of the number of libraries known per developers. According to this chart, a vast majority of programmers (77%) only use very few libraries (between 1 and 5 libraries), and less than 2% use more than 20 libraries.

[3]http://developer.github.com/v3/

B. Looking for library experts

In this experiment, we want to show that LIBTIC is able to find true library experts among a large number of developers. We are interested in two situations: finding experts of one library and finding experts of several libraries at the same time. To find experts of one library, we randomly selected 3 libraries (guava, servlet-api and jackson-core-asl) that are used in at least 20 projects from our corpus. For each such library, we executed a query to find the related experts. To ensure that the highest ranked developers returned by LIBTIC are true experts of the library, we first manually investigated the two highest ranked developers. Second, we sent a mail to these developers where they are presented a list of 4 random libraries from our corpus and the library for which they have been identified as experts. In the mail, they are asked to rate from 0 (don't know the library) to 5 (expert of the library) their expertise in the libraries from the list. To find experts of several libraries, we apply the same process, but we arbitrarily chose the junit and testng libraries as target libraries since we have shown that they are often replaced in software projects [10]. Globally, we had a 50% response rate, with a mean response time of 2 days.

To find experts of guava, we use the following query: `who guava`. The highest ranked developer is *guava-expert-1* with 55 used symbols, and the second is *guava-expert-2*, with 41 used symbols. We analyzed the profile page of *guava-expert-1* on GitHub. Among his public repositories, there are four Java projects. We looked at the source code of these projects, and three are using the guava library. Moreover, this user is the only committer on these repositories. We can therefore assume that he has a good knowledge of guava and that our prototype was right. We also looked at the *guava-expert-2* profile, and he is also using extensively guava in his personal projects. Only *guava-expert-1* answered to our mail. He attributed himself a score of 4/5, the highest one from the list we furnished.

For servlet-api, we use the following query: `who servlet-api`. The highest ranked developers returned are *servlet-expert-1* and *servlet-expert-2*, who used respectively 106 and 72 symbols of the library. By looking at the GitHub page of *servlet-expert-1* we see that he is the creator of Jenkins, a continuous integration server programmed in Java. Since Jenkins defines many servlets, our prototype seems once again to be right. Regarding the user *servlet-expert-2*, his web page states that he is a "*Web Architecture Consultant specialized in open source frameworks*". His GitHub profile page shows that he is very active in a lot of Java projects. It is therefore very likely that he is an expert of Java servlets which are a classical web component in Java applications. Only *servlet-expert-2* answered to our mail. He attributed himself a score of 5/5, the highest one from the list we furnished.

For jackson-core-asl, we use the following query: `who jackson-core-asl`. We identified *jackson-expert-1* as the highest ranked expert with 68 used symbols. In second position, *jackson-expert-2* used 31 symbols, but contrary to the aforementioned developer, he uses more recent versions of the library, ranging from 1.8.2 to 1.9 instead of 1.1.1 and 1.5.5 for *jackson-expert-1*. As indicated by his web page, *jackson-expert-1* is a very active developer of Jersey, a Java platform to build REST services. Since JSON is the main output format of REST services, it is very likely that *jackson-expert-1* is an expert of this library. We looked the page of *jackson-expert-2*,

and we saw that this developer maintains an online tool to validate JSON documents. This tool uses the Jackson library as backend. Once again, our prototype seems to have found two relevant experts for this library. *jackson-expert-1* did not answered to our mail, but *jackson-expert-2* attributed himself a score of 4/5 , the highest score in the list we furnished.

For junit and testng, we use the following query: `who junit* testng`. This query returns 25 developers. The highest ranked developer is *test-expert-1*. By using the query `desc test-expert-1 junit* testng`, we can see that he knows 26 symbols of junit and 85 symbols of testng. The second developer is *test-expert-2* and knows respectively 14 and 31 symbols of junit and testng. The resume of *test-expert-1*, available on his web page, states that he has worked as a "*software quality engineer (writing tests, executing, creating automated tests, preparing CI environments, analyzing source code and writing tools to assist developers and testers)*". In the answer to our mail, he attributed himself 3/5 and 3/5 for these libraries, the highest scores in the list we furnished. We can therefore conclude that he seems a good candidate to explain the differences between junit and testng. *test-expert-2* did not answer to our mail, and no additional results by manual searching could be found to confirm his skills.

C. Using LIBTIC in a software project

In this experiment we conduct a two case studies on a software project to show the usefulness of LIBTIC. We chose the Apache HBase project to perform this case study because it was one of the most active projects in our corpus. There are 33 developers on its source code repository, and it uses 29 libraries. In the first case study, we use LIBTIC to assess the library knowledge of the HBase developers. In the second case study, we use LIBTIC as a task recommendation system for the issues involving the guava library in HBase issue tracker. The two queries shown in Listing 1 define two variables containing respectively the HBase developers and the libraries used by HBase. This information is used in both case studies.

1) Assessing library knowledge: From a management point of view, we argue that having a global view of the library expertise of all developers is very important. For instance, developers that have a good expertise on many of the libraries of a project are very important to the project. These developers should be kept in the development team. Also, it is dangerous to have libraries for which there is only one expert in the team. In this case training a few more developers on these libraries would be recommended.

By using LIBTIC we can easily compute what we call a library expertise matrix that indicates which developers are expert of which libraries. Such a matrix for the HBase project is easily built using the following query: `how* $hbase_devs @hbase_libs`.

The library expertise matrix of HBase, shown in Table I, reveals that there are only 21 developers out of 33 that are expert of at least one library. It is clear from this matrix that the different libraries have different number of experts. For instance 8 libraries out of 29 (in bold) have only one expert developer. Developers should be trained on these libraries, and their usage in the project should be reconsidered. Moreover, we see that 3 developers out of 33 (in bold) are expert of more than a half of the libraries of the project. They are therefore very important to the project.

2) Recommending tasks: We now investigate if LIBTIC can be used to identify candidate members to realize software maintenance or development tasks that require an expertise on some libraries. To that extent, we sought for tasks involving libraries in the issue tracker of HBase[4]. We focused on tasks related to guava library as an example, and went through each issue report containing the keyword guava either in the summary, description or comment field. We looked into the issue description and comments to ensure that the issues were actually in relation with the guava library. Using this process, we gathered a total of 11 issues.

Then, we used LIBTIC to retrieve the HBase developers that are experts of guava using the following query: `who $hbase_devs guava`. To ensure that developers would have been recommended only using information from LIBTIC at the time of the issue creation, we run this query on a DUD extracted from the HBase repository until the creation date of the issue. Therefore, the number of guava experts increase with the date of the issue creation, from 1 to 10 experts for the last issue. Then, we manually found the logins on the issue tracker corresponding to these developers.

For each issue involving guava, we manually collected the list of the HBase developers that were involved in the resolution process of the issue. We discarded all the developers that appeared in the issue but whose only role is to report the bug or change the issue status. We also discarded users that did not correspond to a HBase developer (such as *HBase-QA* or *Hudson*). Finally we partitioned the set of the retained developers in two: experts (for developers knowing guava), and non-experts (for developers with no expertise of guava). The results obtained using this process are shown in Table II. Additionally, we show in this table the precision and recall of a recommendation system that would have recommended the issues to all the experts of guava at the time of the issue creation.

In Table II we can see that the recall of such a recommandation system increases with time. For the first issues, the precision is good, but the recall is low. For the last issues, the recall is very good and the precision lower, which is interesting for such a recommandation system. Moreover, even if the precision seems low, the other experts might have participated to the issue if they had been notified of its existence, improving the precision. In summary, LIBTIC seems to be, after a training time, a recommendation system with a high recall value for issues involving libraries.

V. RELATED WORK

Repository mining has already been used to exhibit profiles in the usage of source code [11], [12], [13]. All these approaches however focus on elements (such as a file, class or method) contained in the projects, but not on the external libraries. It should be noted that Ma et al. already shown that usage and implementation expertise have a similar accuracy to find developers [11]. They argue that developers who use a method at least know what the method does, without any idea

[4]https://issues.apache.org/jira/issues/?jql=project%20%3D%20HBASE

Listing 1: Variables used by queries of Section IV-C

```
set $hbase_devs {hbase-dev-1 ... hbase-dev-21} ; # HBase developers

set @hbase_libs jaxb-api hadoop-core high-scale-lib jersey-core mockito-all libthrift
    jersey-server jackson-mapper-asl commons-lang stax-api avro protobuf-java jersey-json jetty
    commons-cli junit jsr311-api slf4j-api metrics-core servlet-api-2.5 jsp-2.1 commons-logging
    guava htrace thrift hadoop-test zookeeper log4j json ; # HBase libraries
```

about the implementation. Our approach is based on the same assumption that considers that a developer knows a library symbol as soon as he uses it.

Developer profile has been studied in bug triaging. Nguyen et Al. propose profiles of developers based on bug reports [14]. DREX recommends developers for bug resolution based on previous expertise on similar bugs [15]. Servant et al. propose a tool to assign to developers the fixing of test case failures based on fault localization and history of the related source code [16]. Our approach is related to these triaging approaches as it proposes a model for library usage. Our case study shows that it could be used to improve the identification of the developer that would best fix library-related bugs.

Surian et al. propose a recommendation system of collaborators in Sourceforge [17]. Developer profiles are yielded based on projects properties and keywords, which are too coarse-grained to target specific libraries. Similarly, CARES is a Microsoft project standing as a Visual Studio extension that exposes developers profiles who have contributed to a given file [18]. These two approaches cannot be used to capture the library usage of developers.

Ye et al. propose a tool that helps developers to learn about usage of Java APIs [19]. The tool inputs developers code and records which Java API are used by it and how. Further, one can ask the tool to return all developers that have used the API in a specific way to show examples. This approach focuses on library API learning. On the contrary, our approach on retrieving skilled developers on libraries. Also, their tool does not work with version control systems, since when developers submit their binary Java program, they have knowledge of any method called in the program. Therefore, this system lacks accuracy and is unsuited for collaborative development.

Understand API usage has been investigated in past years. Ekoko et al. explain how developers face unfamiliar APIs [20]. Zhong et al. propose MAPO, an approach that queries source code repository to extract API usage patterns as sequences of operations [21]. Zhang et al. propose an approach to recommend API parameters [22]. Our approach may be used to improve the understanding of API usage. However, our main goal is to provide relevant developers based on a static analysis of API usage.

Lämmel et al. propose a large-scale study on API usage over a large set of open source projects [23]. Their approach is different from ours as it is manual and based on the import headers of source code file. Using a set of 77 libraries, they evaluate how much of their symbols are used. Moreover, this approach does not consider the temporal aspect but just a snapshot of each project (the last one), not its history, and not developers contributions. Bauer et al. assess library usage of Java projects thanks to a static analysis that is based on JAR files located in the projects code base [24]. They attach a visualization to understand and evaluate where each library is used in the project, but they do not propose developer profiles.

Okur et al. performed a large scale experiment to study how programmers use parallel libraries from Microsoft [25]. They observed that most often the libraries are misused or not exploited as efficient as they could be. They also noticed that usage patterns follow a Pareto rule meaning that few library elements are used most of the time. This kind of study targets mainly library designers. Our approach is useful in this context to identify experts having a wide coverage of API usage for a given library. They become candidate of interest to communicate with library designers.

The API evolution problem has been widely studied in various manners to update client code when an API evolves. Dig et al. have studied manually the changes through several versions of four libraries and showed that 80% of the changes are refactoring [26]. A technique to address this problem is to mine source code that already performed an API update, as Schafer et al. suggested [27]. Similarly, Dagenais et al proposed SemDiff [28], a client-server connected to a framework source code repository that mines its modifications to generate recommendations to clients. Another technique consists in analyzing the API internal structural changes to determine origin of new elements. Some promising results have been achieved in this area [29], [30]. It should be noted that a recent study from Cossette performs a retroactive study on several library updates performed manually [31]. They listed the different changes and adaptations they had to make. They argue that existing automatic approach such as the ones exposed above are not enough satisfying, since the problem of API evolution is too complex and requires inevitably human intervention. In our context, our approach can identify developers that have experimented several versions of an API and thus can advice for on evolution task.

VI. Conclusion and future work

In this paper, we propose to automate the identification of third party library experts. Our motivation is driven by the fact that software nowadays depends on third party libraries. Our contribution is twofold. First, we propose a theoretical definition for a library expert. This definition is based on a syntactical analysis of the commits performed by the developers. The main idea is that a developer is an expert of a library if he has committed changes to the source code that uses the library. Second, we provide a query language that can be used to identify library experts within large communities.

We demonstrated the feasibility of our approach by collecting developers expertise on a large set of Java projects

TABLE I: Library expertise of the HBase developers. Libraries are the rows and developers the columns. Libraries in bold are libraries for which there is only one expert. Developers in bold are the developers that have expertise on more than a half of the libraries.

	hbase-dev-1	hbase-dev-2	hbase-dev-3	hbase-dev-4	hbase-dev-5	hbase-dev-6	hbase-dev-7	hbase-dev-8	hbase-dev-9	hbase-dev-10	hbase-dev-11	hbase-dev-12	hbase-dev-13	hbase-dev-14	hbase-dev-15	hbase-dev-16	hbase-dev-17	hbase-dev-18	**hbase-dev-19**	hbase-dev-20	**hbase-dev-21**
avro																			●		
commons-cli	●																		●	●	●
commons-lang														●							
commons-logging	●	●		●	●	●	●		●	●	●		●	●	●		●		●	●	●
guava	●				●			●					●	●			●	●	●	●	●
hadoop-core	●		●	●	●	●		●			●	●	●	●					●	●	●
hadoop-test	●																				
high-scale-lib											●										
htrace																					●
jackson-mapper-asl																			●		●
jaxb-api	●																		●		●
jersey-core																			●		●
jersey-json	●																		●		●
jersey-server	●																		●		●
jetty	●																				
json	●																				
jsp-2.1																					●
jsr311-api																			●		●
junit	●				●			●		●	●	●	●	●		●	●	●	●		●
libthrift																			●		●
log4j						●													●		
metrics-core																			●		●
mockito-all							●												●		●
protobuf-java	●										●								●		●
servlet-api-2.5	●																		●		●
slf4j-api																			●		●
stax-api	●																		●		●
thrift																			●		●
zookeeper	●																				

TABLE II: Classification of the developers involved in the resolution of guava related issues, and precision/recall of a recommendation system that would have recommended the issues to all the guava experts.

Issue number	Date	#Experts	#Experts in issue	#Non-experts in issue	Precision	Recall
2714	11/06/10	1	1	2	1	0.33
2724	14/07/10	1	1	2	1	0.33
3264	23/11/10	4	2	2	0.5	0.5
3609	07/03/11	6	1	1	0.17	0.5
3952	04/06/11	6	2	1	0.33	0.67
4012	21/06/11	6	2	2	0.33	0.5
4385	13/09/11	7	1	1	0.14	0.5
4569	10/10/11	7	2	0	0.29	1
5739	06/04/12	9	3	0	0.33	1
5955	08/05/12	9	4	0	0.44	1
6368	10/07/12	10	5	0	0.5	1

hosted on GitHub. These projects were managed by Apache Maven. To that extent, we designed a prototype, named LIBTIC, that extracts developers library usages and supports our query language. We however claim that our approach is language independent as it is only based on static source code analysis.

LIBTIC is intended to be used by project leaders. We evaluated its effectiveness in three situations related to software maintenance. First, it can be used to evaluate the library knowledge among the developers of a given project. Second, it can be used to identify developers of a given project that are experts of a given library with the objective to ask them to perform tasks that require such an expertise. Third, it can be used on a whole community to identify if there are experts that can be contacted to answer some very specific questions about libraries. In all these scenarios, LIBTIC was able to return relevant developers profiles. Main conclusion of our study is that library expertise is a promising concept to integrate in any software environment.

While very promising, this approach is still underused. We plan to extend it to provide benefits for library providers. Since our approach helps to best understand how developers use libraries, such knowledge can be used by library providers to identify missing services or to measure the impact of a library update on projects that depend on it. Moreover mining results reported by the tool with machine learning techniques may also provide interesting pattern. For instance if some library expert is likely to require expertise on another one.

Validation of this tool is also a long and hard task. We plan to set up a controlled experiment to validate the tool in real conditions to avoid subjectivity in the evaluation process. When applied to other field, e.g. bug triaging, this approach should have more validation and comparison with already existing dedicated tools.

Finally we think about improving our approach to make it supporting any kind of technological expertise. We do think that not only library expertise can be measured by looking at the symbol usage made by developers. For instance, we think about measuring the expertise in a programming language by looking at the use of several symbols such as annotations or generics for Java or C#.

REFERENCES

[1] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, ser. CSCW '12. New York, NY, USA: ACM, 2012, p. 1277–1286. Available: http://doi.acm.org/10.1145/2145204.2145396

[2] Y. Ye and K. Kishida, "Toward an understanding of the motivation of open source software developers," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 419–429.

[3] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution of the core team of developers in libre software projects," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, p. 167–170. Available: http://dx.doi.org/10.1109/MSR.2009.5069497

[4] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *Proceedings of the 12th international conference on Top productivity through software reuse*, ser. ICSR'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 207–222. Available: http://dl.acm.org/citation.cfm?id=2022115.2022138

[5] S. Chiba, "Javassist – a reflection-based programming wizard for java," in *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct. 1998.

[6] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle, "Software bertillonage: finding the provenance of an entity," in *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, pp. 183–192.

[7] E. W. Myers, "An o(ND) difference algorithm and its variations." in *Algorithmica*, 1986, pp. 251–266.

[8] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, p. 166–181, Feb. 2005. Available: http://dx.doi.org/10.1109/TSE.2005.28

[9] M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," *Science of Computer Programming*, no. 0, pp. –, 2011. Available: http://www.sciencedirect.com/science/article/pii/S0167642311002048

[10] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering 2012, 15th-18th October 2012, Kingston, Ontario, Canada*, IEEE, Ed., Kingston, Ontario, Canada, Oct. 2012, pp. 289–298. Available: http://hal.archives-ouvertes.fr/hal-00761204

[11] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, "Expert recommendation with usage expertise," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 535–538.

[12] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, p. 121–124. Available: http://doi.acm.org/10.1145/1370750.1370779

[13] H. Kagdi, M. Hammad, and J. Maletic, "Who can help me with this source code change?" in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 2008, pp. 157–166.

[14] T. T. Nguyen, T. Nguyen, E. Duesterwald, T. Klinger, and P. Santhanam, "Inferring developer expertise through defect analysis," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 1297–1300.

[15] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "DREX: developer recommendation with k-nearest-neighbor search and expertise ranking," in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, 2011, pp. 389–396.

[16] F. Servant and J. A. Jones, "WhoseFault: automatic developer-to-fault assignment through fault localization," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, p. 36–46. Available: http://dl.acm.org/citation.cfm?id=2337223.2337228

[17] D. Surian, N. Liu, D. Lo, H. Tong, E.-P. Lim, and C. Faloutsos, "Recommending people in developers' collaboration network," in *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, ser. WCRE '11. Washington, DC, USA: IEEE Computer Society, 2011, p. 379–388. Available: http://dx.doi.org/10.1109/WCRE.2011.53

[18] A. Guzzi and A. Begel, "Facilitating communication between engineers with CARES," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, p. 1367–1370. Available: http://dl.acm.org/citation.cfm?id=2337223.2337420

[19] Y. Ye, Y. Yamamoto, K. Nakakoji, Y. Nishinaka, and M. Asada, "Searching the library and asking the peers: learning to use java APIs on demand," in *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, ser. PPPJ '07. New York, NY, USA: ACM, 2007, p. 41–50. Available: http://doi.acm.org/10.1145/1294325.1294332

[20] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: an exploratory study," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, p. 266–276. Available: http://dl.acm.org/citation.cfm?id=2337223.2337255

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented*

Programming, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, p. 318–343. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_15

[22] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, p. 826–836. Available: http://dl.acm.org/citation.cfm?id=2337223.2337321

[23] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, p. 1317–1324. Available: http://doi.acm.org/10.1145/1982185.1982471

[24] V. Bauer and L. Heinemann, "Understanding API usage to support informed decision making in software maintenance," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, p. 435–440. Available: http://dx.doi.org/10.1109/CSMR.2012.55

[25] S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, p. 54:1–54:11. Available: http://doi.acm.org/10.1145/2393596.2393660

[26] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006. Available: http://dx.doi.org/10.1002/smr.328

[27] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*, 2008, p. 471. Available: http://portal.acm.org/citation.cfm?doid=1368088.1368153

[28] B. Dagenais and M. Robillard, "SemDiff: analysis and recommendation support for API evolution," in *IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009*, 2009, pp. 599–602.

[29] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, p. 325–334. Available: http://doi.acm.org/10.1145/1806799.1806848

[30] H. A. Nguyen, T. T. Nguyen, G. Wilson,Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, p. 302–321. Available: http://doi.acm.org/10.1145/1869459.1869486

[31] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, p. 55:1–55:11. Available: http://doi.acm.org/10.1145/2393596.2393661