Who's my Best Guy for the Job ? Automatic Extraction of Developer Expertise

Cédric Teyton
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
cteyton@labri.fr

Marc Palyart
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
mpalyart@labri.fr

Jean-Rémy Falleri
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
falleri@labri.fr

Floréal Morandat
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
fmorandat@labri.fr

Xavier Blanc
Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
xblanc@labri.fr

ABSTRACT

Context: Expert identification is becoming critical whether to ease the communication between developers in case of global software development or to better know members of large software communities. To quickly identify who is the best expert that will certainly best perform a development task, both the assignment of skills to developers and the computation of their corresponding knowledge level have to be automated. **Method:** In this paper we propose XTIC, an approach that targets this objective with the intent to be accurate and efficient. XTIC proposes a language to specify skills and levels. It then proposes an automatic process that extracts skills and levels from source code repository and a simple mechanism to ease the identification of experts. We have validated XTIC both on Open Source and industrial projects to measure its accuracy and its efficiency. The results we obtained show that its accuracy is between moderate and strong and that it scales well with medium and large size software projects.

1. INTRODUCTION

Expert identification is becoming critical whether to ease the communication between developers in case of global software development [17, 12] or to better know members of large software communities [7, 20]. As software development is more and more complex and requires many different expertises, a particular attention is currently paid to automate the identification of experts with the intent to quickly identify who is the best developer to perform a given development task.

From an abstract point of view, expert identification consists in (1) assigning skills to developers but also (2) defining their corresponding knowledge level. A skill is an abstract term that can have multiple definitions. It can be very generic such as *java*, *test* or *web design*. It can also be more specific such as library skills with for example JUnit[1] or JQuery[2]. Further, skills can even be proper to a given project or a given team of developers such as *owner of the 'core' components* for example. The levels of skills are values that must belong to an ordinal scale such as high, middle, low or a five stars level for instance. Their intent is to compare the levels of developers for a given skill in order to rank them and to identify the best expert.

Any approach that aims to automate the identification of experts has then to provide a process that automatically assigns skills to developers and compute levels. To that extent, both the semantics of skill and level have to be clearly defined. Moreover, the process has to be accurate as it must identify true experts. It should also be deterministic or at least convergent as it should always assign the same skills to the same developers with the same levels. Moreover, it has to be efficient with the main objective to identify experts as fast as possible depending on the complexity of the skill searched for and on the size of the community.

Many approaches aim to support expert identification. Some of them target specific skills such has code ownership [13] or bug fixing [1]. Other aim to provide an abstract data model that can be queried on to ease the identification of experts [17, 2]. To the best of our knowledge, none of them propose to explicitly define the semantics of skill and level, which is the main purpose of our approach.

Our objective is to help the ones that aim to identify software experts. Our purpose is to support project managers who want to assign tasks to their developers or developers who search collaborators to help them. Our approach addresses the three following questions:

- *How to specify skills and levels?*

[1] JUnit is a Java library used to develop Unit Tests in Java
[2] JQuery is a JavaScript library that eases many manipulation of the DOM tree

- *How to extract developer's skills and levels?*
- *How to classify developers to better identify experts?*

Regarding these three questions, our approach proposes a DSL (Domain Specific Language) for defining the searched skill and level. Based on this language it supports an extraction mechanism that browses software repositories (currently only source code repositories) to automatically assign skills to developers and to compute levels. It then proposes aggregation formulae to rank developers and, on top of them, a simple mechanism to ease the identification of experts.

Our approach has been completely implemented in a open source tool, named XTIC. We have validated XTIC by realizing two experiments. First, we identified experts in a sub-part of GitHub that is composed of 16 projects and 280 developers with the intent to stress the scalability of our approach. Our experiment has shown that expert identification can be realized for these projects in less than 3 hours. Second, we have used our approach in an industrial use case with the intent to validate the accuracy of our approach. It has also shown that all the expertises expressed by our industrial partners could been defined by our language. Further, the experts identified by our approach strongly correspond to experts manually identified by our industrial partner.

Our paper is organized as follows. Section 2 starts by presenting the related work. Section 3 clarifies the field of expert identification by giving some rigorous definitions. It then deeply presents our approach. Section 4 presents the two validations we have done. Finally section 5 discusses the limits our approach and suggest some improvements and the section 6 concludes.

2. RELATED WORK

Expert finding is a prolific topic in the domain of information retrieval [10, 14] and is an important subject of the Enterprise tracks of the Text REtrieval Conferences (TREC) since 2005 [4]. However these research approaches are based on generic text analyses and thus cannot fully exploit the data present in a source code repository (history, ownership).

Another example of early work is the Expertise Browser (ExB) [17] proposed by Mockus and Hersleb. They define the concept of experience atom (EA), a basic unit of experience, that can be built from each delta of each files from the VCS of the project. An EA may then be associated with one or multiple domains (functional area, technology used, purpose or type of the change). Finally to measure the level of expertise in a particular domain they count the number of EA associated with this domain. This work is close to our mainstream idea, however there is information on how the set of domains is built and how the domains are specified, and finally how the experience atoms are extracted. We argue that our DSL allows more freedom and flexibility in the specification of the targeted skills.

Specific research approaches targeting the search of experts in the case of a software development project has been accomplished. The Expertise Recommender (ER) [15] proposed by McDonald and Ackerman was one of the first. Their tool relies on two heuristics to identify experts. The first one called change history considers that the revision authors are the experts for the corresponding file. The second one called tech support uses a support database to identify users that already solved problems.

More recently, several research tools [5, 16, 21] were developed to help a developer who is looking for an expert on a file she is modifying or using. They use the touches made on the selected files to determine the list of experts.

As we have seen, some projects rely solely on the analysis of the source code [17] and others attempt to improve the recommendations by looking at other forms of data. For example Moraes et al. with the Conscius tool [18] look at mailing lists.

Codebook [2] is a tool that builds a graph by mining the source code repository, the work item database, the employee directory as well as documents. It also offers a regular expressions language that can be used to search the graph. Based on this framework they have developed several applications over the years. For example, Hoozizat [2], a web search portal for finding the people who own and are responsible for a feature, an API, a product or a service. Another example of application is CARES [12] which is a Visual Studio extension that exposes developers profiles who have contributed to a given file.

Schuler et al. have defined a technique to identify experts of Java methods by looking at the history of usage of a particular method [19]. They argue that developers that use a method should be considered with the same level of expertise than developers that change and edit the method. In their work the unit of experience considered is thus a Java method.

At that point, it is crucial to understand that the domains of expertise we target are not necessarily related to source code areas. For instance, a skill can be defined as the usage of third-party library in the software, which is often scattered in many places of the software. Also, if one want to track who adds or removes the *Deprecated* annotation on Java elements, we clearly see this is not linked with source code areas. One purpose of our approach is to allow to track either skills related to the implementation of a unit of code (ex: edition of a Java method) or usage of this unit (ex: a call to this method).

Globally our work distinguishes itself from these approaches by providing a way for specifying (with a DSL), extracting (with a tool) and measuring (according to several levels) the developers skills. To our knowledge, no existing work comes up with these three particular features. The DSL not only eases the work of writing down the skills, but is intended to offer much more possibilities when specifiying the targeted skills.

3. APPROACH

As a base hypothesis, we consider that skills can be extracted from syntactic modifications performed by developers. For instance, we consider that a developer is a Java expert if and only if she has syntactically edited Java files. The main principle of our approach is then to analyze changes made

by developers on software artifacts to extract their skills as well as their levels. It should be noted that we deliberately choose to focus on the analysis of software repositories as they contain most of the activity performed by developers. As a consequence, we support only programming skills, i.e. skills that can be extracted from artifacts that are managed within a source code repository.

As there are many definitions of skills and levels, we propose to define them using a domain specific language. This language is used to define what are the syntactical changes that a developer must perform on a software artifact to have a corresponding skill. For instance, one may consider that a developer must just modify a Java file to be a Java developer. However, another one may consider that such a definition is too permissive as Java files do not only contain Java code but also comments. She may consider that a developer must create a Java class or modify Java methods to be truly considered as a Java developer.

Once skills and levels have been rigorously defined, our approach uses them while browsing source code repositories to compute developers skills levels. A skill level is a quantitative measure that is used to compare the developers, in order to identify experts. Our extraction mechanism is incremental and has been optimized to scale well and to quickly measure developers skills even in large software projects. The comparison of skills levels are based on well known clustering algorithms.

This section presents the three main elements of our approach: the skill definition language, the skill extraction process and the expert identification process. It starts by giving definitions of the main concepts. Then it presents our language to define developers skills. Afterwards, it presents our extraction process and finally describe how we compare skills levels.

To better explain our approach, we introduce a toy example of a Java project managed by a software repository. Figure 1 presents the only Java file contained in this project. This file has been modified three times. The first version of the file ($v0$) has been created by Alice. Then the version $v1$ has been committed by Bob. Finally, the version $v2$ has been committed by Alice again. We assume that the manager of this project is interested in quantifying developers expertise with regard to the JUnit test framework. We assume that she wants to identify two kinds of JUnit expert. The first ones (Test creators) are developers that create tests and the second ones (Test Modifiers) are the developers that update existing tests.

3.1 Definitions

To ease the explanation of the approach we first lay out several definitions. Let D be the set of developers, S the set of skills and L the set of levels. The goal of any approach that aims to identify experts is to assign skills to developers with a corresponding level. In other word, such an approach computes a binding function that builds triples $(d, s, l) \in D \times S \times L$. Our approach computes such a binding function by analyzing software repositories. In particular, it analyzes all the syntactical changes that are made by developers. To that extent, we define the concept of an *atomic file change* that is the set of all the modifications performed to a file by a developer during a commit. More formally an atomic file change c is completely defined by the two versions of the file before (f_0) and after (f_1) the commit ($c = (f_0, f_1)$). The kind of the modification is either defined by the software repository or can be inferred by looking at the two files. If f_0 does not exist, that means that the file has been created. If f_1 does not exist, that means that the file has been deleted. If both files exist and have the same name, it means that the content of the file has been modified. If the names of the two files are not the same, it means that the file has been renamed or moved.

Finally, as a commit may target several files and as developers commit several times in a software project, we consider that each developer d performs a set of changes C_d. The set of changes of each developer can be computed by browsing any software repository. For instance, the repository of Figure 1 contains the two following sets of changes (C_a for Alice and C_b for Bob):

$$C_a = \{(\emptyset, \text{Foo.java}_{v0}), (\text{Foo.java}_{v1}, \text{Foo.java}_{v2})\}$$
$$C_b = \{(\text{Foo.java}_{v1}, \text{Foo.java}_{v2})\}$$

Our approach then inputs the set of changes C_d of all developers d and performs an analysis to assign skills to developers with a corresponding level. This analysis consists in matching changes ($c \in C_d$) against syntactical patterns, which have been defined using our domain specific language.

3.2 Skills and levels definitions by syntactical patterns

Rather than presenting the syntax of our domain specific language for defining skills and levels, we present in this section its main concepts. We propose to specify a skill as a set of *syntactical patterns* applying to atomic file changes. If the modifications that a developer performs during an atomic file change match at least one pattern of a skill, then the skill is assigned to the developer.

A skill definition may contain several patterns. Moreover, a developer may perform several atomic file changes. As a consequence, several matches occur for one developer and one skill during the analysis of a software repository. The number of matches corresponds to the level of the skill. A level then corresponds to the number of editing operations performed by a developer that match a skill definition.

A syntactical pattern of a skill defines a syntactical modification that can be done on the file target of an atomic file change. The modification can be done on the file (create the file, delete it or change its name) or can be done on its content (change a line or a token). We then propose that a syntactical pattern is a chain of four filters: a *path filter*, a *kind filter*, a *content filter* and a *tree modification filter*.

Both the *path filter* and the *kind filter* apply to the file target of the atomic change file. The *path filter* constraints the path of the file while the *kind filter* applies to the kind of change

```
//Version v0
class Foo {
}
```

```
//Version v1
class Foo {
 @Test
 void testFoo() {
  assertTrue(true);
 }
}
```

```
//Version v2
class Foo {
 @Test
 void testFoo() {
  assertTrue(true);
  assertFalse(false);
 }
}
```

Figure 1: A project repository containing only the `Foo.java` file, modified three time. Version v0 has been committed by Alice, then Bob has committed the version v1 and finally Alice a committed again the version v2.

that is made. The *content filter* and *tree modification filter* apply to the content of the file. The *content filter* considers that the content is a sequence of character while the *tree modification filter* considers that the content is an abstract syntax tree (AST).

These four filters are chained for the sake of efficiency. All atomic file changes that have been filtered out by one filter won't be checked by the next filters. The *path filter* applies first. It filters atomic file changes regarding the path of the file. Then the *kind filter* applies. It filters atomic file changes w.r.t. the kind of the change. Then the *content filter* applies. It filters atomic file changes w.r.t. the content of the file that is considered to be a sequence of characters. Finally, the *tree modification filter* applies.

Path filter. The purpose of the path filter is to constrain the paths of the files that must be modified by a developer to have a skill. It is a regular expression that constraints the path of the target file of an atomic file change. As an atomic file change is completely defined by two versions of a file (f_0 and f_1), which may have different path in case of renaming for instance, the path filter contains two regular expressions (one for f_0 and one for f_1). In our example, the JUnit tests are contained in Java files, the path filter is therefore the same for both the JUnit creator and the JUnit modifier, for both f_0 and f_1: `"^.*.java"`.

Kind filter. The purpose of the kind filter is to describe which are the kinds of modification that must be performed by a developer to have the skill. The kinds of modification can be: creation, modification, deletion or renaming. The kind filter is therefore defined by a mask that specifies the allowed kinds of modification. By default, only the creation and the modification of files are allowed. In our example, the JUnit creator can create, modify or rename files whereas the JUnit modifier must only modify or rename them (and not delete or create, as an updated file cannot be found in these cases). The kind filter is therefore `"added,modified,renamed"` for the JUnit creator and `"modified,renamed"` for the JUnit modifier.

Content filter. The purpose of the content filter is to describe what must be the content of the files that must be changed by a developer to have the skill. We choose to use regular expressions to specify this filter. As a regular expression is not really adapted to express absence of strings, a content filter is composed of two sets of regular expressions (the positives and negatives ones). Moreover, as the content filter may apply to f_0 and to f_1, each file may have two sets of regular expressions. In our example, for the JUnit creator and modifier, the file must contain a reference to the JUnit library, which can be specified by a regular expression

Listing 1: The annotated modification tree computed between the two ASTs corresponding to the first and second modification of Figure 1

```
<CompilationUnit>
 <TypeDeclaration>
  <SimpleName label="Foo" />
  <MethodDeclaration added="1">
   <MarkerAnnotation added="1">
    <SimpleName label="Test" added="1" />
   </MarkerAnnotation>
   <PrimitiveType label="void" added="1" />
   <SimpleName label="testFoo" added="1" />
   <Block added="1">
    ...
   </Block>
  </MethodDeclaration>
 </TypeDeclaration>
</CompilationUnit>
```

defining that the `org.junit` string must be included in the file. Such a regular expression is contained in the positive set. Further, it only constraints f_1 for the JUnit creator but constraints both f_0 and f_1 for the JUnit modifier.

Tree modification filters. The purpose of the tree modification filter is to describe what are the changes that must be performed by a developer to the AST of the file to have the skill. Before explaining the semantics of this filter, let us explain on which structure it works. We assume that the files f_0 and f_1 are in the same format, or that one of them is empty. By same format, we mean that they can be parsed into an abstract syntax tree (AST) by the same parser. We build the AST of the two files and compute an *annotated modification tree* using these two trees. The annotated modification tree is constructed by applying the longest common sub-sequence algorithm on the sequence constructed by a depth first pre-order traversal of the nodes of the two trees. It corresponds to the AST of the file after modification, where the nodes that are not contained in the longest common sub-sequence are annotated as added. Listing 1 shows the annotated modification tree corresponding to the second modification of Figure 1.

The tree modification filter is an XPath expression over the annotated modification tree. It returns a set of nodes corresponding to the location where a syntactical modification indicating the skill has been observed. As a tree modification filter may return several nodes, we increment the level of the skill for each returned node.

For instance, for the JUnit creator skill, we look for developers that have added a method definition with a `@Test` annotation. Therefore the filter is: <tree parser="java">//MethodDeclaration

[MarkerAnnotation[@added]/SimpleName[@label='Test'][@added]]</tree>. This XPath expression selects all `MethodDeclaration` nodes that contain a `@Test` annotation marked as added. For the second skill, the filter is: <tree parser="java">//MethodDeclaration[not(@added)][MarkerAnnotation[not(@added)]/SimpleName[@label='Test'][not(@added)]][..//*[@added]]"</tree>. This XPath expression selects all `MethodDeclaration` nodes that are not marked as added that contain 1) a `@Test` annotation not marked as added and 2) any node marked as added.

3.3 Skills and levels extraction

Our skills extraction process is a two steps process. The first step consists in creating the required data model by browsing a software repository. During this first step, a particular attention has to be paid on renamed and moved files as they can have a significant wrong impact on the results of our approach. A renamed/moved file in most VCS (Versioning Configuration System) is seen as a file deletion and creation, performed in the same commit. Therefore, when applying our skills extraction process, all the syntactical patterns contained in the file might be assigned to the developer that performed the renaming/moving, even if she performed no modification at all on the file content. To avoid that, we introduce a move/renaming detection algorithm that we run when extracting a set of changes from a VCS. This algorithm focuses on commits where there are both deleted D and added files A. On these commits, we compute the normalized compression distance of each element in $D \times A$. Whenever this distance is below a threshold 0.25, and whenever it involves two files that have no better distance with another file, we mark the files has being moved/renamed.

The second step aims at assigning skills to developers and to compute a level, which is a positive integer that represents how many times a syntactical pattern has been matched. This step is quite straightforward. It iterates on the atomic file changes of each developer. For each atomic file change, all the patterns that define a skill definitions are checked. As previously explained, skill definitions are composed of several syntactical patterns. These patterns are checked iteratively against changes and return each one a positive integer. These integers are finally aggregated into one integer that corresponds to the level of the skill. By default they are aggregated using a sum, but it is possible to provide a custom formula.

3.4 Expert identification

To simplify the analysis of the results of the skill extractor, we propose to apply a post-processing step. Its purpose is to map the levels of skills into a smaller ordinal scale such as `low`, `medium` and `high`. We propose to use the K-means clustering algorithm [8] to map the developers into the clusters. This algorithm creates clusters of values regarding their distance. Configured to output three clusters, it then groups lower values together, as well as higher values and medium values. Thanks to this step, one can then ask for an expert with a high level. This post-processing is optional as it depends on the number of developers being analyzed. If this number is to small, the clusters will not reflect the developer skills. In this case, the integer value is more meaningful.

4. VALIDATION

In this section we first describe the implementation of XTic. We then stress test our approach on several open-source projects, to evaluate its performance. Finally, we evaluate the accuracy of our approach in a case study conducted with AKKA, our industrial partner.

Our validation then aims to answer to the following research questions:

- Is XTic able to analyze a large-scale set of data? (RQ1)
- Are the results computed by XTic correct? (RQ2)

4.1 Implementation

XTic has been written in Java on top of the Harmony [9] framework. Harmony is an infrastructure designed to ease the development of tools that mine software repositories. It provides an abstract model that can be used to specify analysis of commits. Thanks to this model Harmony analysis such as XTic can be run on many version control systems such as Git, SVN, Mercurial, TFS or CVS. The XTic analysis also relies on the Eclipse JDT parser to perform the parsing of the Java files. The building of the clusters of developers is done by using the implementation of the K-means clustering provided by Weka [6].

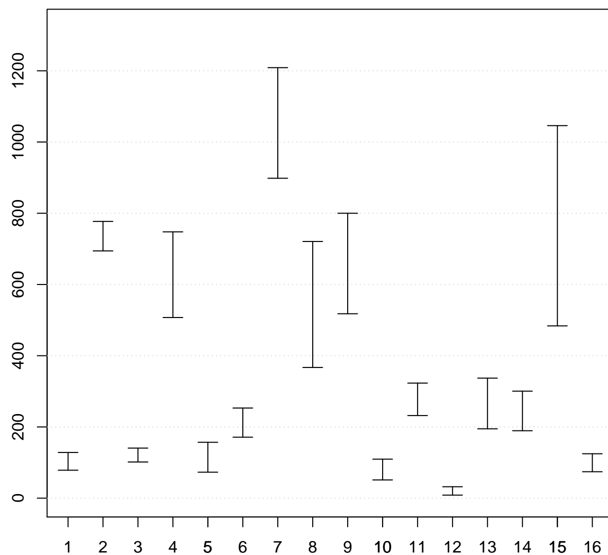
4.2 Stress test

The purpose of this stress test is to answer the first research question (RQ1): *is XTic able to analyze a large-scale set of data?* Since this analysis depends on the syntactical patterns, the size of the source code and the number of versions, it is not possible to make a single experiment to provide a definitive answer. We thus choose to apply some worst case scenarios to a corpus of 16 Java projects. All these project are hosted on GitHub, they are active and have more than one developer. They vary from 140 commits with 2097 lines of code (LoC) for `nanohttpd` to 1419 commits with 79236 Java LoC (`crunch`)[3].

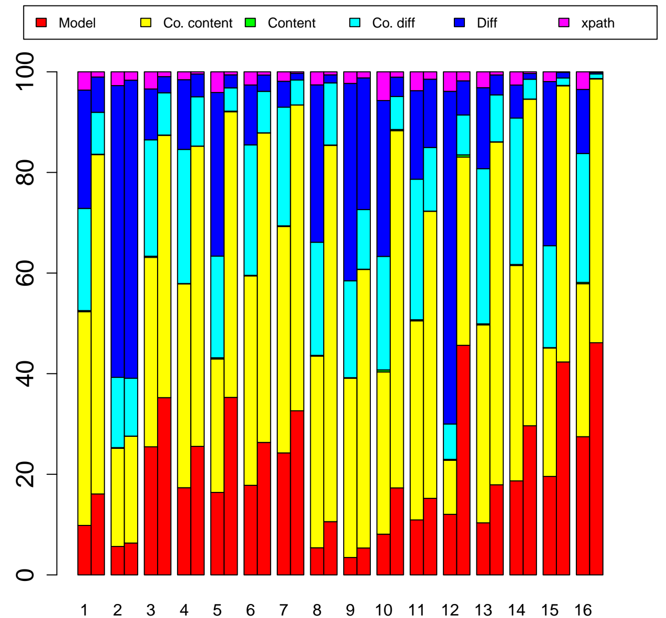
As described in Section 3, a syntactical pattern may contain filters on file names, kinds of modification (creation, edition), file contents and tree modifications. Since the two first filters are almost free to check, this experiment focuses on the last two. Searching for file content requires reading files, thus the worst case is to search for an absent string in all files. Since we are unable to figure out what is the worst case of the tree modification filter, i.e., an XPath query, we define three queries on Java files which have an heavy load:

- all nodes added: the XPath engine has to return plenty of results.
- new local variable: this requires to find specific nodes and most commits introduce variables.
- genericity usage: since genericity may be found on class declaration or instantiation this query has to perform disjunctions.

[3]More details on the corpus may be found at `http://se.labri.fr/xtic/`



(a) Time by project (high: worst case, low: JUnit).



(b) Breakdown % of running time (left: worst case, right: JUnit).

Figure 2: Time and breakdown of XTIC worst case versus JUnit test on a Java corpus.

This worst case scenario is compared with another one that only includes the JUnit skill definition. This definition considers that a JUnit tester is a developer that performs modification on Java files that contain the string `org.junit` and where a method annotated by a JUnit has been introduced or changed. It corresponds to the JUnit creator and modifier patterns described in Section 3.

Experiments are performed on a Intel Core i7 Cpu M640 at 2.80Ghz with 8Go of RAM running with Ubuntu 13.04 (kernel 3.8.0-24) and OpenJDK 1.7.0_25. All these scenarios, i.e., each part of the worst cases and JUnit, are run many times (more than 5) on different passes to avoid IO cache effects and only the worst times are reported. The Figure 2 is twofold, the left-hand side shows the whole computation time (y-axis) for each project (x-axis) as segments. The upper end of each segment represents the worst case scenario, while the lower end represents the JUnit scenario. The right-hand side of Figure 2 shows a breakdown of these running times as percentages. Each group of two bars represents a single project, where the left bar shows the worst case scenario and the right one the JUnit scenario. Breaks represents respectively (bottom up) the time spent analyzing the VCS (Model), the time of checking out the files in order to search for patterns (Co. content), the time of searching effectively these patterns (Content), the time for checking out the previous version of a file (Co. diff) in order to build ASTs and doing the tree differentiation (Diff), and finally the time consumed by the XPath queries (xpath).

Conclusion. This experiment shows mainly two things. First XTIC analysis is generally fast enough, even on large scale project, which answer our RQ1. On average, the worst case scenario (resp. JUnit scenario) is under 7 minutes (resp. 5 minutes) with a maximum of 20 minutes (resp. 15 minutes) for a project containing 1282 revision (with 135644 LoC in the last version)—it is worth noting that it is not even the biggest project neither from the number of commits nor the number of LoC.

Moreover this experiment shows some of the design choice of XTIC. When looking only the left bars, i.e., the worst case scenario, the time is mostly spent in doing tree differentiation which makes sense since all possible diffs are computed. On the more realistic example of JUnit (right bars) a simple content filter avoids most of them, thus the biggest part of time is spent doing the checkout of files. In any case the matching inside the files itself is almost free compared to other things (when carefully looking at the graphs it can be barely seen). As a first advice learned from this experiment, if possible, any skill definition should contain a content filter to avoid wasting time in tree differentiation. Adding more XPath queries is not expensive compared to the whole process as soon as a diff is already computed. Computing many skills in a single pass should be preferred in order to avoid redoing tree differentiation.

When looking at the breakdowns, the project 2 seems really singular since both bars look like the same. After manually inspecting this project, we found that 70% of the files contains test code. Again even though JUnit rules are more realistic, the whole computation time is thus subsumed by tree differentiation which explains this result.

Table 1: Number of patterns and filters in the skill definitions.

Skill	#Patterns	#Filters			
		File	Kind	Content	Tree
config	5	5	0	6	0
workflow	2	2	0	2	0
domain	2	2	0	5	0
modeling	7	7	0	0	0
graphics	3	3	0	4	0
jsf	2	2	0	1	0
plugins	4	4	0	1	0
tests	3	3	0	3	0
Total	28	28	0	22	0

4.3 Industrial case study

The purpose of this case study is to answer the second research question (RQ2): *are the results computed by* XTIC *correct?*. To that extent we perform an experiment on an industrial project where we compare the results of XTIC with developers skills estimated by a manager and a software architect.

4.3.1 Setup

We first present the software project analyzed in this case study, as well as the skills required by our industrial partner.

Project and developers. Our industrial partner AKKA Technologies[4] agreed to grant us access into a software project repository and to provide us a list of developers skills they are interested in. The project studied is a document management system which development started in 2009 and contains 3 years of development activity. A total of 13 developers are involved in this project. The software is mainly written in Java and contains 53 Java KLOC at its latest version.

Skills definitions. We asked to our industrial partner to define the set of developers skills they wanted to measure. A meeting of 2 hours with 2 project managers was necessary to come up with a set of 8 high-level skills that encompass 28 concrete skills according to our definiion. They were described in natural language first. The 2 hours include the time to introduce XTIC capabilities to the 2 project managers. Then, we wrote the skill definitions to integrate them into XTIC. We were able to express all the skills without exception and any challenging problem, and 15 minutes were needed for it. It turned out that each skill necessitate several patterns, and only file and content filters were necessary to write these patterns, as shown in Table 1. XTIC was therefore expressive enough to fit the requirements of our industrial partner. The skills definitions are not freely available for the sake of confidentiality.

Developers skills. We asked our industrial partner to deliver several skills evaluations indicating for each developer $d \in P_D$ and each skill $s \in S$ (as shown in Table 1), what is the level $l \in L = \{\text{low}, \text{medium}, \text{high}\}$ of the developer w.r.t. the skill. We end up therefore with several ternary relations that are subsets of $P_A \times S \times L$. The project manager and architect accepted to complete an independent evaluation based on their personal knowledge. We therefore have two evaluations E_{ar} and E_{ma}, which distribution in the levels are shown in Table 2. As these two persons disagree on several developers, we have also built another relation E_{ag} computed from E_{ar} and E_{ma}, where we kept only the triples (d, s, l) that were included both in E_{ar} and E_{ma}. This relation contains thus only the developers and skills on which both persons agreed. Its distribution in the levels is shown in Table 3.

[4] http://www.akka.eu/index_en.php

Table 2: Distribution of the developers in the levels for the evaluations E_{ar} and E_{ma}.

Skill	#Low		#Medium		#High	
	E_{ar}	E_{ma}	E_{ar}	E_{ma}	E_{ar}	E_{ma}
config	7	7	2	3	4	3
workflow	7	9	2	2	4	2
domain	7	7	4	0	2	6
modeling	6	11	5	0	2	2
graphics	4	5	0	7	9	1
jsf	3	3	4	2	6	8
plugins	9	10	0	0	4	3
tests	3	5	4	1	37	7
total	46	57	21	15	37	32

Table 3: Distribution of the developers in the levels for the evaluation E_{ag}.

Skill	#Low	#Medium	#High
config	4	1	2
workflow	6	1	1
domain	4	0	1
modeling	5	0	0
graphics	0	1	0
jsf	2	0	3
plugins	8	0	2
tests	2	0	5
total	31	3	14

The evaluations E_{ar} and E_{ma} contain $13 * 8 = 104$ triples. The evaluation E_{ag} contains 48 triples. It means that the two persons agreed only on about one developer out of two. Table 2 indicates that the proportion of low, medium and high developers is similar globally but can vary a lot in the rules (in graphics for instance). Low developers are the most common, followed closely by high developers. Medium developers are less frequents. Table 3 indicates that the two persons agree more frequently on the high and low developers than on medium developers. This is probably due to the fact that it is easier to know if a developer is high or low, but the border between low and medium on one hand, and medium and high on the other hand, is fuzzy and very subjective.

4.3.2 Experiment

We ran XTIC on the software project repository using the previously described skills definitions to compute an integer score for each of the 13 developers on each of the 8 skills. To convert the integer into a level, we used the K-means clustering algorithm configured to produce three clusters, as explained in Section 3. It led to an evaluation called E_{xt}. To be able to compare XTIC on the evaluation E_{ag}, that contains only the developers for which the manager and architect agreed, we computed an evaluation $E_{\text{xt}'}$ containing only the developers and skills contained in E_{ag}.

Table 4: Agreements (#A), disagreements (#D), strong disagreements (#SD) and Kappa between XTIC (E_{xt}), the manager (E_{ma}) and the architect (E_{ac}).

Couple	#A	#D	#SD	Kappa
$\{E_{\mathrm{ma}}, E_{\mathrm{ac}}\}$	48	56	21	0.26
$\{E_{\mathrm{xt}}, E_{\mathrm{ma}}\}$	67	37	6	0.59
$\{E_{\mathrm{xt}}, E_{\mathrm{ac}}\}$	59	45	14	0.42
$\{E_{\mathrm{xt'}}, E_{\mathrm{ag}}\}$	41	7	1	0.85

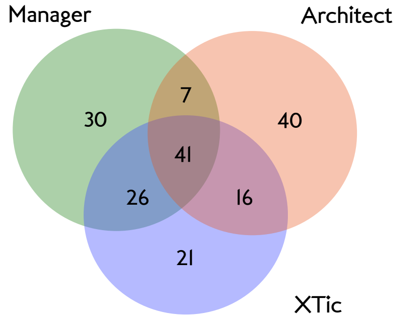


Figure 3: Agreement distribution of the results. Our approach has 21 times disagreed with both the manager and the architect.

4.3.3 Results

To measure the agreements among the three evaluations, we used Cohen's kappa coefficient [3]. This coefficient is a value between 0 and 1, with 1 meaning a perfect agreement and 0 no agreement. The comparison of XTIC evaluation (E_{xt}) with the one of the manager (E_{ma}), the one of the architect (E_{ac}), and between these two persons are exposed in Table 4. In his table, strong disagreement means that a person assigned a low level while the other assigned high level. A detailed comparison is also shown in the Venn diagram of Figure 3.

First of all, the agreement between the architect and the manager is poor (Kappa of 0.26). More over they have many strong disagreements (about 20%). This shows that evaluating developers skills is complex and prone to subjectivity. Moreover, it means that XTIC results can not conform with both the manager and the architect. The agreement between XTIC and both the manager and architect is fair (Kappa of resp. 0.59 and 0.42), which is a satisfying point. Moreover there are fewer strong disagreements for XTIC with both the manager and architect. The agreement with the manager (0.59) is good, and the number of strong disagreement is very low (only about 5%). The results therefore indicate that XTIC have a similar opinion to the manager.

Since the architect and manager significantly disagree, we produced an evaluation E_{ag} where we kept only the triples $(\mathrm{d}, \mathrm{s}, \mathrm{l})$ where the manager and architect assigned the same skill level. The developers and skills contained in this evaluation have therefore a good confidence. This evaluation contains 48 triples, which is still a fair number of developers and skills. As shown in Table 4, the results of XTIC on the developers and skills of E_{ag} contains 41 agreements and only 7 disagreements leading to a Kappa coefficient of 0.85 that indicates an almost perfect agreement.

Table 5: Precision and recall of XTIC for each level of the E_{ag} relation.

Low		Medium		High	
Prec.	**Rec.**	**Prec.**	**Rec.**	**Prec.**	**Rec.**
0.94	0.94	0.29	0.67	1.0	0.71

To have a more precise view of the results of XTIC, we computed the precision and recall of XTIC with regard to each level of skill, as shown in Table 5. We can see that the recall and precision for the low and high levels range from good to perfect. On the contrary, the precision on the medium class is poor, while the recall is just fair. This confirms that a human is more prone to assign either a low or high level, while a medium evaluation is more likely to be given by XTIC. A medium level remains fuzzy from a human point of view.

4.3.4 Discussion on discrepancies

Since a manual analysis of each case of disagreement would have taken too much time, we presented to our industrial partner only the strong disagreements of the manager, the architect and XTIC during a meeting where both the architect and the manager were present. They were asked to reach an agreement on each case. With regard to the cases of strong disagreements between the architect and the manager, it turned out that in every case, the manager were right. The reason that was because the manager is closer to the developers and therefore knows better the work they do every day and the skill they have. On the contrary, the architect assign tasks to the developers, but does not ensure that they perform it in person. Therefore they have a biased vision of the developers skills. This conclusion was very interesting for our industrial partner since knowing whom of the architects and the managers has a better vision of the developers skills was kind of an internal debate. With regard to XTIC, it remains therefore only 6 cases of strong disagreements unexplained with the manager. After having analyzed these cases, it turned out that the manager was right, but he confirmed that these particular skills were not used on the project we analyzed, the developers being assigned to other coding tasks.

5. LIMITS AND IMPROVEMENTS

The previous section shows that XTIC has a good accuracy and a good efficiency. While it offers some new supports to the field of expert identification, it still suffers from some limits. In this section we highlight these limits and explain how we can handle them.

First of all, XTIC is based on a syntactical analysis of source code. The diff technique it uses currently does not support refactoring operations such as method renames or moves. These operations are therefore considered to be *delete* and *add* operations. They have then a strong impact on the skill and level extraction as a developer that just rename or move a method will be considered to be an expert of all the skills that are related to the content of the method. To overcome this limitation, we think about using more efficient diff techniques such as Change Distilling for instance[11]. Furthermore, the parser used by XTIC currently cannot handle files that contain more than one programming language since they are

complex to parse. Such files are for instance HTML files that contain both HTML and JavaScript code. To handle them, we need to define specific parsers and diff techniques.

Secondly, XTic support expert identification by analyzing all the commits performed by developers on a software repository. Therefore it does not reflect the real knowledge level of a developer but more her working level. As a consequence, XTic will not identify as experts developers that do not make a lot of work. For instance, newcomers will always be identified as non expert even if they do have a strong knowledge. This limit comes from the base hypothesis of XTic that states that an expert is a developer who committed in her field of expertise. However, we can overcome it by performing analysis according to a time window (some weeks or some months) with the objective to normalize production of newcomers with the one of older developers.

In this paper, the DSL proposed by XTic was designed to uniformly specify a large range of skills. In practice, XTic is intended to help project managers and developers who are willing to track a defined set of skills. However, this early paper does not give any clue on how much this DSL can be appropriated by a person who does not know XTic. Indeed, in the experiments we were the persons who wrote down the skills, and we rather focused on the results that are produced. In a future work, we will have to evaluate how people appropriate our DSL so that they can use it without our intervention.

Finally, the language proposed by XTic to define skills and levels can be improved. Currently it does not allow for conjunction of skill patterns. However, we think that there exists skills that are spread over more than one file per commit. For instance, in a Java context we can define a library update skill, which consists in modifying the JUnit JAR file and in the same time editing the test files adequately. Moreover even though the expressiveness of our language remains tedious to validate, we observed that we managed to specify a various range of skills. In that direction, additional requirements and feedback from both industrial partners and open source communities would help us to improve our language, and better understand their needs. We also believe that other data sources than Version Control Systems contain skills that have an interest, for instance issue trackers systems. Our tool and the language could be extended to satisfy such new requirements.

6. CONCLUSION

The identification of expert is one of the current challenges of software engineering. In this paper we propose to address this challenge by analyzing source code changes that are managed by a software repository.

Our approach, named XTic, proposes a domain specific language to specify skills and levels. It then automatically extracts developers expertise by browsing software repositories and by matching skills and levels definitions to the work committed by the developers. Finally, it provides some simple support to rank developers and hence to ease the identification of expert.

Our approach is based on the hypothesis that a developer has to commit work in her field of expertise to be considered as an expert. Our approach is then more like *Thomas the Apostle* as it needs to observe commits and does not rely on any other claim.

We have validated our approach by stress testing its implementation to show its efficiency and by checking its accuracy. Our validation shows that it is quite efficient and that its accuracy is between moderate to strong. Without being a silver bullet, XTic provides a support for those who want to identify experts in an automated manner.

As a further work, we first think about overcoming some of the limits of XTic. In particular, we are working on improving the parsers we rely on and on adding time windows in the extracting process. We also think about integrating XTic with other approaches that are also based on syntactical analysis but that targets software artefacts other than source code, such as mails or bug reports for instance.

7. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, page 361–370, New York, NY, USA, 2006. ACM.

[2] A. Begel, K. Y. Phang, and T. Zimmermann. *Codebook: Discovering and Exploiting Relationships in Software Repositories*. 2010.

[3] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, Apr. 1960.

[4] N. Craswell, I. Soboroff, and A. P. Vries. Overview of the TREC 2005 enterprise track. 2005.

[5] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: a project memory for software development. *Software Engineering, IEEE Transactions on*, 31(6):446–465, 2005.

[6] R. De War and D. Neal. WEKA machine learning project: Cow culling. Technical report, The University of Waikato, Computer Science Department, Hamilton, New Zealand, 1994.

[7] G. Demartini. Finding experts using wikipedia. In *Proceedings of the Workshop on Finding Experts on the Web with Semantics (FEWS2007) at ISWC/ASWC2007, Busan, South Korea*, page 33–41, 2007.

[8] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[9] J.-R. Falleri, C. Teyton, M. Foucault, M. Palyart, F. Morandat, and X. Blanc. The harmony platform. Technical report, Univ. Bordeaux, LaBRI, UMR 5800, Sept. 2013.

[10] H. Fang and C. Zhai. Probabilistic models for expert finding. In *Proceedings of the 29th European conference on IR research*, ECIR'07, page 418–430, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[12] A. Guzzi and A. Begel. Facilitating communication between engineers with CARES. In *Proceedings of the*

2012 International Conference on Software Engineering, ICSE 2012, page 1367–1370, Piscataway, NJ, USA, 2012. IEEE Press.

[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*, pages 157–166, 2008.

[14] C. Macdonald, D. Hannah, and I. Ounis. High quality expertise evidence for expert search. In *Proceedings of the IR research, 30th European conference on Advances in information retrieval*, ECIR'08, page 283–295, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, page 231–240, New York, NY, USA, 2000. ACM.

[16] S. Minto and G. Murphy. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07*, pages 5–5, 2007.

[17] A. Mockus and J. D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *In proceedings of International Conference on Software Engineering (ICSE 2002)*, page 503–512, 2002.

[18] A. Moraes, E. Silva, C. da Trindade, Y. Barbosa, and S. Meira. Recommending experts using communication history. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, page 41–45, New York, NY, USA, 2010. ACM.

[19] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, page 121–124, New York, NY, USA, 2008. ACM.

[20] R. Sindhgatta. Identifying domain expertise of developers from source code. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, page 981–989, New York, NY, USA, 2008. ACM.

[21] Y. Ye, K. Nakakoji, and Y. Yamamoto. Reducing the cost of communication and coordination in distributed software development. In B. Meyer and M. Joseph, editors, *Software Engineering Approaches for Offshore and Outsourced Development*, number 4716 in Lecture Notes in Computer Science, pages 152–169. Springer Berlin Heidelberg, Jan. 2007.