Identifying Knowledge Divergence by Vocabulary Monitoring in Software Projects

Jan Nonnen, Paul Imhoff
University of Bonn
Computer Science III
Bonn, Germany
{nonnen, imhoff}@cs.uni-bonn.de

Abstract—During the development of a project, words used in source code add up to a big vocabulary, which may lead to a divergent word-understanding and word-knowledge between developers. Even the drop out of a single developer may lead to a big loss of knowledge about words and their meaning. By keeping track of the active developers vocabulary one is able to identify and react upon such situations, e.g., by applying pair programming to spread the knowledge around the team. In this work we propose a way to identify such situations by analysing the words contained in identifiers obtained through the commit history in a version control system. Initial empirical results are presented and analysed.

Keywords—vocabulary, active vocabulary, vocabulary evolution, software evolution, history mining, program comprehension, project health

I. INTRODUCTION

In software development every developer has individual knowledge and skills as well as a personal programming style. Writing code can also be seen as expressing concepts and ideas in a programming language. In programming languages not only the code structure can be used to express concepts, but also the identifiers carry semantics for a developer [1]. In [2] Deißenböck and Ratiu considered the knowledge in source code to be represented by the identifiers and the relations of the code structure.

Usually developers are free to use whatever names they come up with. Haiduc and Marcus [3] observed that the variety of words chosen by the developers are limited. Still they could verify that domain words were used significantly in the code of the analysed software projects. Evans [4] introduced the approach of domain-driven design with a focus on creating a shared language between domain experts and developers. Further he also emphasised to actively use this language in the source code.

In this work we want to focus on analysing the active vocabulary used in the identifiers of projects. A person’s vocabulary contains all words that they have used or have knowledge about [5], and can be subdivided in an active and passive part. The active side contains words that the person uses in speech or in written form. In contrast, the passive vocabulary contains all words that one knows or even just has seen a few times. The content of the vocabulary constantly changes. On the one side, it increases due to knowledge gaining like reading books, blogs or discussion with other people and on the other side it decreases due to forgetting.

With the help of the project history we want to retrieve the active vocabulary of each developer in the team and want to verify if a shared team language can be identified. We want to use this knowledge to provide each individual developer with words and concepts of other developers in the team that he might be unaware of. Further we want to find out whether and how the author of a particular piece of code can be identified based on the used vocabulary.

Lanza [6] defined evolution patterns based on the code structure. Our evaluations showed these patterns also in the vocabulary evolution. These patterns can then be used to identify knowledge issues due to a stagnating use of a word.

We want to apply these evolution patterns and the knowledge gained by analysing the vocabulary to identify a knowledge divergence in a project. This analysis in form of a vocabulary monitoring should visualise the deficiency and ideally provide assistance to overcome this issue.

The following section presents an overview of current related work. Afterwards, in section III the methodology of our approach is presented. In section IV we describe our devised experiment and analyse initial results. Concluding words and outlooks are shown in section V.

II. STATE OF THE ART

Vocabulary evolution: Abebe et al. [7] studied changes of the team vocabulary by analysing two software systems. Their results indicate that the code size increases faster than the size of vocabulary. Thus developers tend to use a low word variance in their vocabulary. Vocabulary stability during software evolution was investigated by Antonioli et al. [8]. They defined a stability metric as the cosine between two vectors. The vectors contained the frequencies of the words located in the entities in two compared versions.

Feature location: A feature is a set of lines of code that implements a functionality in the system. Textual feature location uses the textual information of identifiers, comments and additional documentation to locate features in the source code. Dit et al. [9] performed an extensive survey on current feature location techniques and also present current textual feature location approaches.

Topic evolution: Thomas et al. [10] presented an topic
evolution model based on differences between versions. Topics are collections of words that co-occur frequently in the project history. Another commonly used topic evolution model is the Hall model [11], which was originally invented for corpora similar to conference proceedings. In [12] Thomas et al. evaluated a topic evolution on 12 public releases of JHotdraw. Three topic evolution metrics (scattering, focus and assignment) were used to categorise change events.

Hindle et al. [13] analysed commit comments and used a time windowed topic analysis to locate trending topics in a project. Grant et al. [14] compared two techniques for analysing topic evolution. They could identify global changes in their evolution visualisation. Our approach also showed these global changes in the vocabulary.

**History mining & collaboration:** Kagdi et al. [15] created a taxonomy for software mining approaches after performing a literature survey of 80 publications of this field. Hattori and Lanza [16], [17], [18] focussed on analysing code ownership. In [16] they classified commits to development and maintenance activities based on the commit comments. Further they observed in their studies that small commits were mostly used for corrective tasks. In contrast to this, large commits where mostly related to the development of new features. The effect of forgetting has been studied since the work of Ebbinghaus [19] and was been applied to refine code ownership by Hattori et al. [18]. Their tool Syde [17] provides developers the possibility to see who changes which code locations in the project.

Code ownership was also addressed by Gîrba et al. [20]. Visualisation of the code ownership was aided with an own defined visualisation called Ownership Map. For these they observed the number of lines in a file and the last author of these.

Linstead et al. [21] mined the Eclipse 3.0 source code and identified which author was mostly contributing to which topic and which authors had mostly similar topics.

**Evolution visualisation:** A visualisation technique for project evolution in form of storylines was presented by Ogawa and Ma [22]. Lanza [6] derived different evolution patterns with the help of a visualisation in form of a matrix, named Evolution Matrix.

### III. Methodology

In this work we focus on analysing the evolution of the individual committer’s vocabulary gained from Java identifiers in the source code history of a project. Our current approach utilizes the Git source code management (SCM) system to obtain the history of a project. As we are interested in the development of the underlying vocabulary over time we do not care for branches, but simply order commits based on their timestamps. For each commit we calculate the changes based on its parent. This so called change set consists of the time and added, removed or replaced lines of code. Further we substitute replaced lines by removing the old lines and adding the new lines.

From the updated source code we eliminate all punctuation and language keywords, keeping only identifiers. Every identifier is split into constituent words using camel case rules (e.g. ‘addQueryResults’ gives add, query, results). These words form the used vocabulary of the commit. In our current approach we do not apply normalization on the words, e.g. normalizing ‘aborted’ to ‘abort’ and we do not remove non dictionary words, e.g. ‘i’. An example that motivates non-normalized use of words is given later in Section IV-B RQ4.

For each word we calculate the added and removed occurrences, thus a word that is added three times and not removed in a commit has added occurrences of 3 and 0 removed occurrences. Thus our history model \( H \) consists of a set of five tuples:

\[
H = \bigcup(c, a, w, f_a, f_d), \text{ where } c \text{ the commit time, } a \text{ the author, } w \text{ the word, } f_a \text{ the added and } f_d \text{ the deleted occurrences.}
\]

The **individual vocabulary** \( V(a) \) of committer \( a \) is defined as

\[
V(a) = \{w \mid \exists(c, a, w, f_a, f_d) \in H, f_a > 0 \lor f_d > 0\}.
\]

The **shared vocabulary** between different committer in a team can then be defined as the set of all words that the developers use together:

\[
V(a_1, \ldots, a_n) = \bigcap_{i=1}^{n} V(a_i).
\]

With this definition, the ubiquitous language from Evans [4] can be seen as the shared vocabulary of all committers in a project. The **team vocabulary** \( V \) is the vocabulary of all \( m \) committers in a project, thus

\[
V = \bigcup_{i=1}^{m} V(a_i).
\]

### IV. Experiment

In this section we want to present first insights on our research questions that we gained by an initial evaluation of two Java projects.

#### A. Research Questions

With the proposed model we want to analyse the following research questions:

- RQ1 Are parts of the team vocabulary used only by team individuals?
- RQ2 Can we find a shared vocabulary used by all team members?
- RQ3 Does the knowledge of committers in both projects differ?
- RQ4 Are parts of the team vocabulary used only temporarily?
- RQ5 Can we identify patterns in the vocabulary evolution?
B. Preliminary results

For the initial evaluation of the vocabulary evolution we analysed two projects: JGit\(^1\) and Cultivate\(^2\). JGit is an implementation of the Git SCM in Java and is hosted as an Eclipse project since 2009 and is in development since 2006. Cultivate is a static code analysis project for Eclipse that is developed at our department at the University of Bonn and it is in development since 2003. This allows us to use additional in depth development knowledge to classify vocabulary evolution patterns.

For both projects we obtained a Git repository containing the commits from October 2009 till November 2011, see Table I. During this time span JGit had 52 and Cultivate 15 different committers. Interestingly, the number of commits was higher in Cultivate (970 commits) than JGit (695). Still the size of commits in JGit was higher than Cultivate. So the commits in Cultivate were more frequent, but also smaller. Using the classification from Hattori and Lanza [16], this means that in JGit more new features were added and the development in Cultivate more focussed on bug fixing and maintenance. The size of our computed vocabulary model \(|H|\) for Cultivate was 16620 and 78292 elements for JGit, thus JGit commits in total were more than four times bigger than the Cultivate commits. The team vocabulary for JGit contained 3391 words and the vocabulary for Cultivate had 1872.

<table>
<thead>
<tr>
<th></th>
<th>JGit</th>
<th>Cultivate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Committers</td>
<td>52</td>
<td>15</td>
</tr>
<tr>
<td>Commits</td>
<td>695</td>
<td>970</td>
</tr>
<tr>
<td>Vocabulary Size</td>
<td>3391</td>
<td>1872</td>
</tr>
<tr>
<td>Evolution Model Size</td>
<td>78292</td>
<td>16620</td>
</tr>
</tbody>
</table>

Table I

Comparison of JGit and Cultivate between October 2009 till November 2011.

RQ1: Are parts of the team vocabulary used only by team individuals?

In both projects over 40% of the vocabulary was used only by a single committer (45% in JGit and. 43% in Cultivate). This individual vocabulary in both projects also contained around 30% abbreviations. Every committer had at least one word that only he used and most of the words were used more than once (3.4 average uses). This indicates that (at least in these two projects) developers can be identified by their usage of these words. Figure 1 shows how many words are used by how many committers (for the eight main contributors) in Cultivate. The size of shared vocabulary is the union of the shared vocabulary with any of one to eight committers. Therefore the shared vocabulary of one committer is the union of all words that were used only by one committer. One can see, that the vocabulary size shrinks exponentially as the number of users increases.

The ubiquitous language in this overview are the words used by all eight authors. Still these words are only a fraction of the team vocabulary used by all eight authors. Further of interest are the words that two committers jointly use. This could happen if both worked together and one adopted the use of a word from another. For example in Cultivate one developer introduced the word ‘matcher’, which was adopted later by another developer in the team.

RQ2: Can we find a shared vocabulary used by all team members?

Both projects have committers who just worked on a small piece of the project and committed only a small fraction of the code. Therefore we decided to use the actual usage count in relation to the total count to filter the committers with less than 1% committed words.

In JGit 40 out of the 52 committers were under this threshold. Nevertheless, the remaining 12 committers used 98% of the team vocabulary and only 48 words of the vocabulary were not used by them. In Cultivate only 7 out of 15 committers were under this threshold. The remaining eight committers used 99% of the vocabulary. This confirms

\(^1\)http://www.eclipse.org/jgit/

\(^2\)http://roots.iai.uni-bonn.de/cultivate
that the main committers still cover a good part of the complete team vocabulary. In the following we will focus on only these committers.

Figure 2 illustrates how the word usage correlates to the number of committers. To calculate the number of developers using a word the complete history was taken into account. One can see that the word usage in a project increases, as more developers use the word. Still, in this diagram one can identify two special regions. On the one hand, the top left region contains the words that are frequently used by a small amount of authors. On the other hand, the lower right contains words used by nearly every author, but are only rarely used in the project.

To get an idea of words that occur in these regions, we analysed Cultivate as it is developed in our department. This allows us to interpret words occurring in these regions. The top left issue A in Figure 2 contains new vocabulary words (e.g. ‘assessment’) recently introduced into the project with a new feature. As the feature was quite cohesive and the word was a central concept within this feature the vocabulary use of the words is quite high. Further as only a few developers had worked on this, the dispersion in the team had not reached a lot of team members. In the near future we expect to see a shift to the right on the X-axis for these words. This could be fostered by making the team more aware of new concepts in the project.

The bottom right issue B corresponds to a feature of the Eclipse framework that is used in the Cultivate project (‘workbench’). The workbench is mostly accessed in the project to add a selection listener to the workbench. This feature is small in size, but nevertheless a few different implementations were created during the project development and many developers worked on this feature.

In JGit a single committer used nearly 90% of the team vocabulary, the second highest committer used only 36%. In contrast to this, in Cultivate three committers have similar vocabulary percentage of over 50%. This could indicate a single point of knowledge in the JGit project. Further, as Cultivate is already in development for a few years and as it is a university research project the fluctuation of student developers in Cultivate is quite high. Still, further research is required in identifying the reasons for this disparity and whether this really represents a project evolution flaw.

In general more research is required to identify a knowledge divergence. An extension of the performed evaluation to more projects should provide a higher sample set. This can then be used to explicitly define the conditions under which a divergence occurs. Ideally one can specify how much the individual vocabulary size is allowed to differ from the mean individual vocabulary size.

**RQ4: Are parts of the team vocabulary only used temporarily?**

We considered the vocabulary model for both projects to answer RQ1. The vocabulary usage \( u(w) \) of word \( w \) is the sum gained by adding the total added frequency and subtracting the removed frequency through the history. Temporarily words in the history can be identified by having zero vocabulary usage. This means that a temporary word was added at a point in time and later as much delete occurrences were observed. JGit contained 426 (≈12%) temporary words and in Cultivate we observed 37 (1%) temporary words. This difference may be explained due to the fact that Cultivate was developed from 2003 on. Thus the vocabulary was already quite stable in 2009. Still the numbers of both projects indicate that RQ4 can be verified, thus parts of the team vocabulary are only used temporarily and are later replaced or removed.

An example of a temporary word in JGit is ‘abort’. This word is used in the early development of the project and later replaced over time by the two words ‘aborted’ and ‘aborting’. The replacement of ‘abort’ to the past and gerund form indicates a concept refinement in the project. This information would be lost by normalization of ‘aborted’ and ‘aborting’ to ‘abort’.

**RQ5: Can we identify patterns in the vocabulary evolution?**

In [6] Lanza studied project evolution with the help of software metrics, namely for each class the number of methods and number of instance variables. During our vocabulary analysis on Cultivate and JGit both showed similar patterns to the ones defined by Lanza.

A pulsar is a class that grows and shrinks repeatedly during the evolution. A word that was found to be a pulsar in Cultivate is the domain word ‘predicate’. Persistent is a class that is present in the project from the start to the end. For example, in Cultivate the word ‘id’ was identified as a persistent word. Lanza suggested to review these classes

**Figure 2.** The distribution of words according to the numbers of committers using it. A illustrates an outlier region with too high uses in comparison to authors and B illustrates too low uses in comparison to authors.

**RQ3: Does the knowledge of committers in both projects differ?**

Figure 3 visualises how much of the vocabulary each committer used and how many words she used in her commits.
for possible dead code. From the view point of vocabulary evolution, persistent words are useful, as they represent a constant knowledge and are constantly used throughout the project. One would expect core domain and technical words to be used constantly in the project. The concept behind ‘id’ is one of the core domain concepts in Cultivate. Further one can argue that normally the developers of Cultivate change often, but still the knowledge of ‘id’ is present in the team at each time. A white dwarf marks a class that loses functionality and shrinks during development. The word ‘lifecycle’ was a core domain concept in Cultivate at the beginning, but was removed slowly. Still there were some occurrences in Cultivate at the end of the evaluation, but those hint more on dead code or missing renaming. In contrast to this, there were also words introduced late in the development which were adopted faster in their usage. This is analogous to the supernova pattern found by Lanza for classes.

In both projects no word was found that mixed two patterns. Each word showed a not yet identified pattern or used exactly one pattern.

### C. Further Observations

In Cultivate it was possible to identify and distinguish between some committers by their use of words. In addition one commit was found that can be identified by the used words as a contribution of a different author than the committer. We could verify this concrete situation in the context of an observed pair programming situation. Generalising from this single successful analysis would be misleading. Still, one can see from this example that one needs to distinguish between an author and committer.

We observed that the ubiquitous language in both projects contained only technical words. In both projects there was at least one developer that had not used a single domain word. In Cultivate, for example, the words ‘smell’ and ‘metric’ are part of the domain language, but two developers didn’t use them at all. This may be related to the features and tasks on which these developers worked on.

Figure 3 (b) indicates on the first sight that the vocabulary percentage and commit size may not correlate (e.g. by considering committer 3 and 4). A correlation analysis between the vocabulary percentage and commit size in Cultivate indicate a correlation of 0.66. This is a smaller correlation than the one found in JGit (0.96). More projects need to be analysed to clearly see a trend in this correlation or provide improvement suggestions during development.

### V. Conclusion

In this work we presented the novel idea to detect knowledge divergence through analysis of the individual vocabulary usage of committers in source code. We presented a preliminary study on two projects (JGit and Cultivate) on which we evaluated our research questions. We identified a single point of knowledge in form of a disparity of vocabulary usage in the JGit project. An analysis of the vocabulary evolution resulted in a first set of evolution pattern that can be used to classify individual words and their pattern. Further we observed that nearly half of the vocabulary is used only by an individual committer, thus the shared vocabulary is limited to half of the possible vocabulary. Also the team vocabulary shared by the team members contained mostly technical words. In both projects domain terms were shared by all but one developer.

A threat to validity of the approach is that the active vocabulary does not necessarily reflect the knowledge of an author. One could have simply copied the code of another author without reviewing the details or a bigger automatic refactoring could have been performed. These are examples were the current approach would assume that the author has some knowledge about the words. Further research about the knowledge evolution of committers should provide insights in this area.

Additionally it is not necessarily needed that all committers have worked on the complete code. Normally one would assume that one can identify a feature based vocabulary. Thus an integration of feature location algorithms can improve the presented approach to a finer vocabulary analysis.
In this case one would need to identify all committers involved in a feature and which commits contributed to the feature.

In the context of an active vocabulary obtained by the commits it is unclear how far one can remember the concepts of each word. A user study should provide more insights in how far forgetting of words affects the proposed approach.

In the future we want to analyse how far program comprehension can be improved by visualising trending or stagnating words. Also showing each individual developer words that he has not used and that all others have used is future work. These visualisations could improve the overall project comprehension and provide a knowledge improvement.

Further these words can be linked with an introduction location proposed by us in [23]. Introduction locations are source code locations for a word that allows a developer to understand the meaning of the word by relying only on the source code. Information about “interesting” words for a developer in combination with an introduction location for each word is a seamless integration of both approaches.

The presented work and the gained insights showed some arguments for visualisation of the vocabulary evolution in form of a continuous health monitor for a project. It should integrate smoothly into the normal workflow of the developer, for example as an Eclipse plug-in. This visualisation should show user specific information regarding knowledge deficiencies and provide program comprehension assistance.

References


