On the Use of Domain Terms in Source Code

Sonia Haiduc, Andrian Marcus
Department of Computer Science
Wayne State University
Detroit, MI 48202
313 577 5408
sonja@wayne.edu, amarcus@wayne.edu

Abstract

Information about the problem domain of the software and the solution it implements is often embedded by developers in comments and identifiers. When using software developed by others or when are new to a project, programmers know little about how domain information is reflected in the source code. Programmers often learn about the domain from external sources such as books, articles, etc. Hence, it is important to use in comments and identifiers terms that are commonly known in the domain literature, as it is likely that programmers will use such terms when searching the source code.

The paper presents a case study that investigated how domain terms are used in comments and identifiers. The study focused on three research questions: (1) to what degree are domain terms found in the source code of software from a particular problem domain?; (2) which is the preponderant source of domain terms: identifiers or comments?; and (3) to what degree are domain terms shared between several systems from the same problem domain?

Within the studied software, we found that in average: 42% of the domain terms were used in the source code; 23% of the domain terms used in the source code are present in comments only, whereas only 11% in the identifiers alone, and there is a 63% agreement in the use of domain terms between any two software systems.

1. Introduction

When trying to understand the source code of a software system, developers usually start by locating familiar concepts in the source code. Keyword search is one of the most popular methods for this, but its success is strictly tied to the quality of the user queries and the words used to construct the identifiers and comments. Developers who are new to a project know little about the identifiers or comments in the source code, but it is likely that they have some knowledge about the problem domain of the software. While this knowledge can be acquired ad-hoc from other developers, often it is based on prior knowledge acquired from more traditional sources such as books, articles, etc. These sources tend to define the discourse in a domain and establish a commonly used vocabulary. For example, if a developer starts working on existing graphics software, it is likely that she already has knowledge about geometric figures, colors, textures, etc. and she will not try to get it from other developers. The problem occurs also when developers use software written by others, such as open source libraries.

It stands to reason that developers should utilize commonly used words from the domain, not only in the source code but also in requirements documents, user manuals, etc. This also helps facilitate communication between stakeholders with different background.

The problem of word choice variability in human-system communication was studied in [11]. Participants in the studies were asked to choose one or more words to describe certain concepts. The subjects were chosen with different expertise, from domain experts to students. The results of the studies showed that the level of agreement in choosing the words was very low and the probability of two people choosing the same word to describe the same concept did not surpass 0.2 on average. If developers follow the same pattern when choosing words to use in the software, then comprehension would be severely compromised. In order to assess this fact, we investigated several software libraries from the same domain (i.e., graph theory), written by different developers.

The paper presents an exploratory case study on the use of common domain terms in the source code of various software libraries from the same domain. Our aim is to answer the following research questions:

978-0-7695-3176-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPC.2008.29

1. To what degree are domain terms in a particular problem domain found in the source code of software systems in that domain?
2. Between identifiers and comments, which category is a richer source of domain terms?
3. To what degree are domain terms shared between several systems in the same problem domain? In other words, to what degree do programmers agree on choosing domain terms for naming concepts in the source code? Does this agreement follow the general pattern of lexical choice agreement between human subjects or does it have a different fingerprint?

2. Related work

In our work, we focused on two main issues: analyzing the software vocabularies in order to observe the use of domain terms and analyzing the agreement between these vocabularies in using the terms. The related work section captures these two aspects in two separate subsections. The first issue is common in program comprehension, whereas the later is specific to cognitive psychology.

2.1. The study of software vocabularies

The study of the source code vocabularies in the context of program comprehension has been the subject of a multitude of papers [1-5, 7-9, 12, 14, 16, 19, 20, 22, 23].

Researchers have long acknowledged the important role that the vocabulary of the source code, i.e., identifiers and comments, plays in understanding software systems. A particular problem, which affects program comprehension based on identifiers and comments, is the fact that often their quality is far from perfect. Researchers have studied and defined different methods to improve identifier quality. In [14] and [23] it was investigated how different naming styles (i.e., single letters, abbreviations, and full words) affect comprehension and both studies came to the conclusion that full word identifiers provide better comprehension. Caprille and Tonella [5] analyzed the function names is software systems by considering their lexical, syntactical, and semantic structure.

The comprehension value of program identifiers has been often tied to their capability of reflecting real world or domain concepts. Several papers have addressed the quality of identifiers based on relationships between the source code and the problem domain. Relevant to our work is [7], where the conciseness and consistency of identifiers is analyzed based on the relationship between concepts and their names. A formal model based on bijective mappings between concepts and names to achieve concise and consistent naming is proposed. A framework to study how the same identifiers can be trusted to represent the implementation of the same concepts was proposed in [1]. In [16] the authors proposed a method based on an improvement of the term frequency/inverted document frequency measure to mine concept keywords from identifiers in large software projects. Early work by Biggerstaff et al. [4] addressed the problem of identifying human oriented concepts in a software system and associating them with their implementation instances. The authors coined it as the concept assignment problem.

Ratiu et al. [19] studied the mappings between identifiers and real world concepts and introduced a formal framework that allows the recovery of these mappings. In [20], the authors particularly addressed the way identifiers reflect domain concepts in software libraries. They proposed a method to describe and evaluate the bias of the libraries' implementation of the real-world concepts expressed in an ontology. This work is most related to our study, although it tackles the use of domain terms in the software from a different perspective and it does not addresses the problem of lexical agreement between different software systems.

2.2. Lexical agreement

Outside the program comprehension field, but relevant to this work, researchers studied how people choose words to describe concepts. This work is relevant as we parallel our findings with the results of these earlier studies.

The study in [11] addresses the problem of word choice variability in human-system communication by performing a series of empirical studies in which the participants were asked to choose one or more words to describe certain concepts. The studies covered five different application domains, ranging from specialized domains to general knowledge. The subjects were chosen from different categories, from domain experts to students. The results showed that the level of agreement in choosing the words was very low and the probability of two people choosing the same word to describe the same concept did not surpass 0.2 on average.

The variability of word choice between people was also studied in the context of document summarization [6]. Considering a large set of source documents from newswire stories (i.e., 2,781 documents) and a set of summaries for each of these documents (i.e., 9,086 summaries in total), the authors computed the average pair-wise lexical agreement between summaries of the same document. The agreement measure was based on the normalized average number of common words between summaries and the results revealed that there was a low agreement (i.e., 24% on average) between them. Although the agreement measure used in [6] is different than the one used in [11], the results indicate a similar phenomenon that there is a rather low probability of two people to use same words when describing the same concepts or events.

An adverse problem was studied in [21], where evidence about people associating different meanings with the same word is being presented. The authors show through a series of empirical studies that even though people might agree at a rough level about the meaning of a word, they might disagree about its precise definition and in particular about the context in which the word can be used. They prove that the results do not depend on the level of expertise of the subjects or the level of specialization of the domain. The studies revealed also that along with a common domain word base, people also have an individual domain vocabulary.

The work in [15] analyzes the differences between knowing and naming objects, applied to categorization. In their empirical studies, the authors asked subjects to divide a set of objects in several categories, based on common features. Although the subjects see the similarities among the objects in much the same way, they show substantially different patterns in naming the categories.

3. Selecting and defining the domain, concepts, terms, and vocabularies

We chose to study the use of domain terms in a set of software libraries that implement algorithms and data structures common in graph theory. Our choice was motivated by the fact that the domain is well defined and understood and that there are several open source libraries easily available.

To avoid misunderstandings and support replication, we define the terminology and methodology used in this paper. By *domain concept* we refer to a concept existing in the problem domain of a software system and commonly used in the specialized literature. For example, in our study, "graph", "node", and "edge" are all domain concepts in graph theory and represent main concepts referred to in the literature. On the other hand, concepts like "pancyclic graph", "genus", "equipartite graph" are very specialized concepts in graph theory, therefore rarely used and not included in our set of domain concepts.

A domain concept is described using *domain terms.* A term is considered to be a domain term if it refers to a domain concept and it is commonly used in the domain literature. Commonly used abbreviations are also considered as domain terms. Note that there are a variety of English words describing a domain concept, which are not commonly used in that domain. For example, both words "edge" and "connection" can be used to describe the concept of an edge between two nodes, but we consider only "edge" as domain term, since "connection" is rarely used in the graph theory literature with this meaning. In other words, if there are several English synonyms for the same concept, we selected only those that are frequent in the mentioned literature. For example, we consider "node" and "vertex" as domain terms, both referring to the concept of a node in a graph, but we do not consider "point". The set of all domain terms, selected as we described, forms the *domain vocabulary.*

For our study we manually selected the most common concepts in graph theory, based on several books[1] and online sources[2]. We chose 135 domain concepts. From the same resources, for each of these concepts one or more terms and standard abbreviations that describe the concept were manually selected and included in the domain vocabulary. We also included common misspellings of these words. It is the reason why we use *term* rather than *word* as some of these terms are not English words. For example, for the concept "degree of a vertex", the terms "degree", "deg" (standard abbreviation), and "valence" were added to the domain vocabulary. The graph theory domain vocabulary constructed like this, contained initially 193 terms. Our choice of concepts and associated terms may not provide a full coverage of the domain and domain vocabulary respectively. Complete domain coverage was not the goal of our work, as we considered a set of the most common domain concepts and domain terms to be sufficient for the purposes of the case study.

The mapping between the set of domain concepts and the domain vocabulary is neither injective nor surjective. For some of the concepts there are more than one domain terms in the domain vocabulary describing it (e.g., "vertex" and "node" describe the same concept - synonyms) and some of the domain terms referred to more than one concept (e.g., "adjacent" could refer to "adjacent vertices", the "adjacency matrix", or "adjacent edges" - homonyms). We only consider single terms, as opposed to sequences of terms (i.e., n-grams) in our analysis.

---

[1] B.Bollobas, *Modern Graph Theory,* Springer, 1998;
R.Diestel. *Graph Theory,* Springer, 2005;
J.L.Gross and J.Yellen, *Graph Theory and Its Applications*, Chapman&Hall, 2006

[2] http://en.wikipedia.org/wiki/Glossary_of_graph_theory
http://en.wikipedia.org/wiki/Graph_theory
http://en.wikipedia.org/wiki/List_of_graph_theory_topics

This does not affect the results of our study, as we only count the use of the same terms in different software systems, not to which particular concept from the domain model a term refers to.

A *software vocabulary* represents the set of terms contained in all identifier and comments found in the source code. The identifiers are split according to common naming conventions. For example, "setValue", "set_value", "SETvalue", etc. are all split to "set" and "value".

The *software domain vocabulary* represents the intersection of the software vocabulary and the domain vocabulary.

A complete list with all the domain concepts and terms we chose in this work is available at:

http://www.cs.wayne.edu/~severe/icpc08data

The most laborious steps in this work are the selection of the domain concepts and their corresponding descriptive terms. These steps are also most dependent on the people involved. Ideally, one would use a well defined ontology for a domain. Unfortunately, there are currently very few domains for which both complete domain ontologies and freely available software exist. The subsequent steps are largely automated with simple text processing tools. Once better domain ontologies will exist, most of the work in such a study can be easily automated.

4. Case study

In order to answer the proposed research questions, we performed a single-case exploratory case study on six open source graph theory libraries (see Table 1), studying the use of domain terms in identifiers and comments. Graph theory was chosen to be the domain for the case study, as it is accessible and specialized enough so that the use of homonyms is not widespread. Specialization is important in the context where verifying the meaning of the domain terms in the source code of the software systems would imply a significant effort. By using a specialized domain, the probability that the domain terms have the same meaning in the software vocabularies as in the domain vocabulary increases. In consequence, a manual verification, although desirable, is less needed to ensure validity than in the case of domains with more general terms. We chose software libraries over full systems due to their focus on a addressing a particular problem domain and the absence of presentation aspects.

4.1. The objects of the case study

The data collection was opportunistic and the libraries were chosen due to their availability and access to the complete source code. The six graph theory libraries chosen include two C++ and four Java libraries.

Table 1. The size of the libraries in lines of code (LOC) including comments and number of unique terms before and after filtering and stemming

Library	LOC with comments	Unique terms	Unique terms after filtering and stemming
GOBLIN	122,139	5,201	3,077
Boost	28,451	2,933	1,635
JGraphT	25,408	2,591	1,424
GraphStream	14,033	1,972	1,059
Plexus	27,003	1,577	825
SJGraph	6,425	850	463

GOBLIN[3], one of the C++ libraries, is focused on graph optimization and network programming problems. It deals with all of the standard graph optimization problems discussed by textbooks and in courses on combinatorial optimization. The *Boost Graph Library*[4] is part of the wider set of C++ Boost libraries and provides general purpose graph classes, based on generic components.

The Java libraries include *JGraphT*[5], which is a free Java graph library that provides mathematical graph-theory objects and algorithms. JGraphT supports a rich gallery of graphs and is designed to be powerful, extensible, and easy to use. *GraphStream*[6], another Java library, manages dynamic graphs. It is composed of an object oriented API that gives an easy and quick way to construct and evolve edges and nodes in a graph. *Plexus*[7] is a Java library with specifications and implementations for generic graph data structures. *SJGraph*[8] (**S**imple **J**ava **G**raph) is as the name suggests a minimal graph library and contains a matrix- and a list- based graph implementations.

4.2. Methodology

We developed a simple parsing tool that extracts the comments and identifiers from Java and C++ code. Identifiers are split as described in Section 3. After compiling the initial vocabularies, filtering and stemming was applied to each of them.

The reason behind filtering was to eliminate terms that are irrelevant to the domain. There were two categories of words considered for elimination:

[3] http://www.math.uni-augsburg.de/~fremuth/goblin.html
[4] http://www.boost.org/libs/graph/doc/index.html
[5] http://jgrapht.sourceforge.net/
[6] http://graphstream.sourceforge.net/
[7] http://sourceforge.net/projects/plexus/
[8] https://sourceforge.net/projects/sjgraph/

common English words[9] and programming language keywords and common constructs. The elements in the first category were prepositions, conjunctions, articles, common verbs, nouns, pronouns, adverbs and adjectives (e.g., “should”, “may”, “any”, “user”, “people”, etc.). In the category of programming language keywords and constructs, both Java and C++ keywords were considered. Some examples of such terms are “class”, “private”, “if”, “while”, etc.

It is common for people to use different lexical or grammatical forms of the same word, such as using the plural form of nouns. We assume that “node” and “nodes” refer to the same domain concept so we consider them the same term. In order to identify any common root shared by two or more terms, stemming was applied to the software vocabularies. We used the freely available Porter stemmer [18] for this task. Table 1 indicates how stemming affected the size of each software vocabulary.

We applied the same process to the initial domain vocabulary. After filtering and the stemming, the number of terms in the domain vocabulary decreased to 180.

4.3. Measures

As mentioned, the goal of the case study was to answer three different, yet related, research questions. The domain and software vocabularies constructed as presented above contained the necessary data to answer these questions. In each case, we defined a set of measures that helped in quantifying the results.

In order to measure to what degree are domain terms in a particular problem domain found in the source code of software systems in that domain, we analyzed the number of terms in the software domain vocabulary for each library.

In order to assess whether the comments or the identifiers contain more unique domain terms, we rebuilt the software domain vocabularies for each library by first excluding just the comments and then excluding just the identifiers. The differences between these versions of the software domain vocabularies for each library gives us the number of unique domain terms that are used in identifiers alone and in comments alone, respectively.

The most interesting issue in the case study was to investigate the level of agreement between the software domain vocabularies (i.e., implicitly between the programmers who developed the libraries). The six libraries were developed by a number of different programmers. We take inspiration from the field of summarization and we consider each library as a (partial) summary of the graph theory domain. In [6] the authors compute the level of lexical agreement between summaries of the same source document. They found that on average there was a 24% agreement between two summaries of the same document.

We adapt here the vocabulary agreement measure defined in [6], by considering the software domain vocabularies as summaries. The intersection of each pair of software domain vocabularies is computed. Then, the percentage of domain terms shared by the libraries in a pair is calculated. Finally, the level of lexical agreement (LA) in choosing domain terms is determined by dividing the average number terms shared between two software domain vocabularies by the average number of terms in these vocabularies. The formula for computing *LA* is described below. *N* represents the number of software systems (i.e., software vocabularies) considered and V_i represents the complete software vocabulary (including both identifiers and comments) of system *i*.

$$LA = \frac{2!(n-2)!}{n!} \sum_{\substack{i,j=1 \\ i<j}}^{n} |V_i \cap V_j| \bigg/ \frac{1}{n} \sum_{i=1}^{n} |V_i|$$

Our expectation was that developers are disciplined when selecting domain terms to be included in the source code and hence, the average vocabulary agreement level between libraries should be higher than the 24% observed in summaries.

4.4. Results and interpretations

Table 2 shows the number and percentage of domain terms found in each of the six libraries. On average, 76 (42%) of the domain terms appear in the software domain vocabulary of one library and in total 139 (77%) of the domain terms were used in the six libraries together.

Table 2. The number and percentage of domain terms found in each of the six graph theory libraries

Library	Number (%) of domain terms
GOBLIN	117 (65%)
Boost	100 (56%)
JGraphT	77 (43%)
GraphStream	44 (24%)
Plexus	69 (38%)
SJGraph	44 (24%)

The number of domain terms used in the software vocabularies varies considerably from library to library. This is not a surprise as the libraries do not

[9] www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words

implement the exact same features. The sizes of the software domain vocabularies are correlated with the sizes of the software vocabularies (correlation index 0.88 – see Table 1). The only exception is GraphStream, which has less domain terms than Plexus, even though the size of its vocabulary is 22% larger than the one of Plexus.

There is also a strong correlation between the size of the software libraries when considering the number of lines of code and the number of terms in their software domain vocabularies. The correlation index in this case is 0.81. An exception is JGraphT, which even though it has less lines of code than Plexus, it has 12% more domain terms than Plexus. Also, GraphStream and SJGraph have the same number of domain terms, even though GraphStream has more than twice the number of lines of code of SJGraph.

A further interesting issue in analyzing the use of domain terms in the source code of software libraries is the comparison of the amount of domain terms included in the comments, on one hand and in the identifiers, on the other. This gives a deeper understanding of how important comments are in understanding unfamiliar software systems and about which source of information would help programmers understand more about the functionality of a software system in a domain. As mentioned above, we extracted the software domain vocabularies in two different ways and obtained two vocabularies for each library: v1 where identifiers were removed and v2 where comments were removed. The results are summarized in Table 3. Column C1 shows the average percentage of domain terms found in comments from the total number of terms in the software domain vocabularies, obtained from v1 of each library. This number accounts for terms that may also appear in the identifiers, in addition to comments. Column C2 represents the average percentage of domain terms found only in comments, which do not appear in identifiers, obtained from v1-v2 for each library. Columns I1 and I2 show the analogous measures for terms that appear in identifiers, obtained from v2 and v2-v1, respectively for each library.

The data indicates that comments are a richer source of domain terms than identifiers (note that we did not perform any test to verify the statistical significance). Except in the case of SJGraph, comments contain more domain terms than identifiers. In average, 23% of the domain terms used in the source code are present in comments only, whereas only 11% in the identifiers alone. This is a rather unexpected result as we assumed developers capture a lot of domain terms in identifiers, as this process is supported by Object Oriented analysis and design techniques. We expected that comments would rather explain design decisions and the logic behind the computational parts of the software. This finding justifies further study on the meaning of comments, which we plan to undertake.

Table 3. The percentage of domain terms in the software vocabularies in comments vs. identifiers C1 – comments; C2 – comments only; I1– identifiers I2 – identifiers only

Library	C1	C2	I1	I2
Goblin	96%	15%	85%	4%
Boost	87%	16%	84%	13%
JGraphT	99%	23%	77%	1%
GraphStream	89%	36%	64%	11%
Plexus	99%	29%	71%	1%
SJGraph	68%	16%	84%	32%

The data in Table 2 indicates that there are significant differences in the scope of the six libraries. We considered this issue when computing the lexical agreement between libraries. We consider each software domain vocabulary V_i (containing both identifiers and comments) as an unordered set of terms. We computed the Jaccard measure [13] for each pair of vocabularies (i.e., the size of their intersection over the size of their union). For each vocabulary pair (V_i, V_j), the Jaccard measure is $|V_i \cap V_j|/|V_i \cup V_j|$.

Table 4. Pairs of libraries and the Jaccard measure of their domain vocabularies

Pair no.	Pair of libraries	Jaccard
1	GraphStream & JGraphT	42%
2	GraphStream &Plexus	41%
3	GraphStream & SJGraph	35%
4	JGraphT & Plexus	57%
5	JGraphT & SJGraph	36%
6	Plexus & SJGraph	35%
7	GOBLIN & GraphStream	36%
8	GOBLIN & SJGraph	35%
9	GOBLIN & Plexus	50%
10	GOBLIN & JGraphT	55%
11	GOBLIN & Boost	68%
12	Boost & SJGraph	37%
13	Boost & Plexus	51%
14	Boost & JGraphT	55%
15	Boost & GraphStream	40%

In Table 4, the 15 possible pairs of libraries and the computed percentage of shared terms (i.e., the Jaccard measure) are represented. These measures can be used as indicators of the overlap in functionality between two libraries.

However, the Jaccard measure does not distinguish between the individual libraries in a pair, so we refined the measure to show the percentage of domain terms in each library from a pair that are also used in the other library in that pair. We consider each pair of libraries and their corresponding vocabularies as an ordered set (V_i, Vj). For each pair (V_i, V_j), we compute $|V_i \cap V_j|/|V_i|$ and $|V_i \cap V_j|/|V_j|$ (see Figure 1).

Boost and Goblin are the largest libraries, which use the most domain terms, and we can see in Figure 1 that a large percentage of the software domain vocabulary of the other libraries is in fact included in Goblin's or Boost's domain vocabulary. For example, in the pair Goblin and GraphStream (pair number 7), 34% of the terms in Goblin's domain vocabulary are used also in the domain vocabulary of GraphStream, whereas 97% of GraphStream's domain vocabulary is also used in Goblin's domain vocabulary. This suggests that most of the domain concepts implemented in GraphStream are also implemented in Goblin. Based on the same logic, we identified the domain terms common to all libraries. There are 18 domain terms used in all six graph theory libraries and all of them represent main concepts in graph theory, such as, "graph", "edge", "node", "tree", etc.

Finally, we computed the lexical agreement between the library pairs, as defined in Section 4.3. On average there is an agreement of 63% between the pairs of libraries. This result exceeded even our expectations, considering the differences between the scope of each libraries. It indicates that there is a strong probability that developers will use similar domain terms when building software that addresses problems in the same domain. The results suggest that programmers agree significantly more than people in other areas when choosing names to describe concepts in a domain.

4.4.1. Synonymy, homonymy, and domain coverage

In the domain literature we consulted (books and online resources), we found that there are more terms that can refer to the same concept. These terms are called *synonyms*. Also, there can be more concepts referred to by the same term. The terms that can refer to more than one concept are called *homonyms*. Being based on the terminology used in the domain literature, our domain vocabulary also contains terms from both categories.

With this in mind, we tried to capture the domain coverage that each of the software libraries achieves. In the cases when a domain term refers to more than one domain concept, all the domain concepts represented by that domain term were considered as being implemented in the source code. The results of the domain coverage are presented in Table 5, which describes the number of concepts represented by the software domain terms in the source code and the

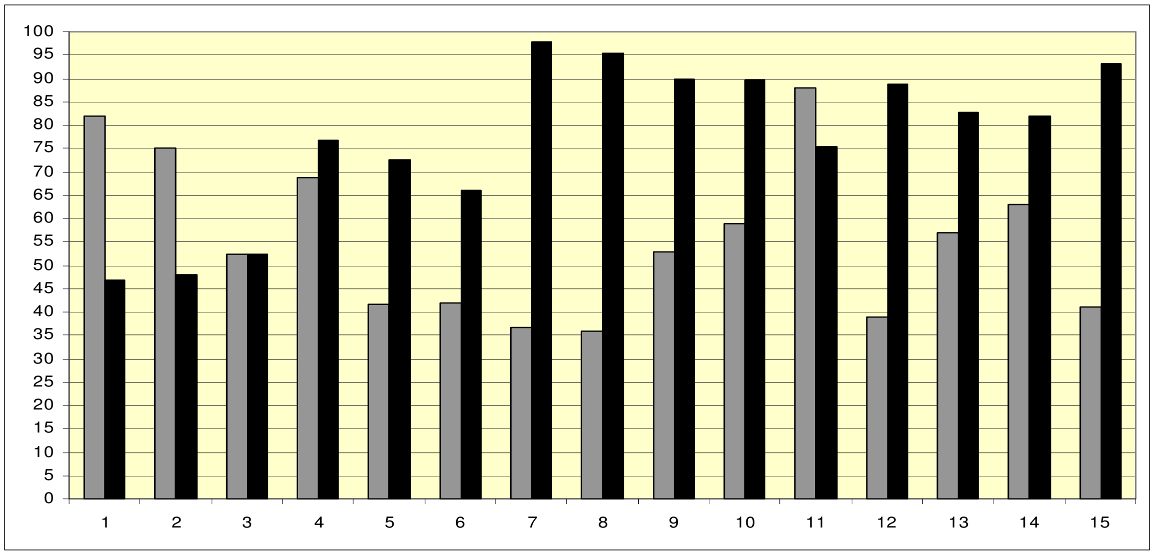

Figure 1. The percentage of shared domain terms for each of the library pairs. See Table 4 for the mapping between the library pairs and the numbers that represent them on the X-axis. The left column (grey) shows what percentage of domain terms from the first library in the pair are used in the second one. The right column (black) shows what percentage of domain terms from the second library in the pair are used in the first one.

percentage it represents from the total number of concepts in the domain.

Table 5 differs from Table 2 by the fact that Table 2 shows the number and percentage of domain terms, whereas Table 5 presents the number and percentage of domain concepts contained in the software libraries. The mapping between domain terms and concepts was done manually by the authors. We can conceive that different people may do a slightly different mapping. We do not believe that the mapping mechanism affects the results drastically.

The results reveal that Goblin and Boost implement the largest number of concepts (they are also the largest software libraries in terms of vocabulary and LOC size). Interesting is that even though Plexus and SJGraph have less unique terms than JGraphT and GraphStream do, they implement more graph theory concepts. This indicates that there is no clear correlation between the size of the vocabulary and domain coverage, as one would expect. We hypothesize that if more programmers develop a library, they would tend to use more synonyms. Verifying such a hypothesis remains as future work.

Table 5. The number and percentage of domain concepts in each of the six graph theory libraries

Library	Number (%) of domain concepts
GOBLIN	112 (83%)
Boost	95 (70%)
JGraphT	89 (66%)
GraphStream	50 (37%)
Plexus	82 (61%)
SJGraph	50 (37%)

5. Threats to validity

As with any exploratory case study, generalization of results is not straightforward [10]. The goal of exploratory case studies is to build new theories where none are provided. These studies can be holistic or embedded the latter occurring when the same case study involves more than one unit of analysis. Using embedded studies detecting "slippage" from the original research questions is easier [17]. Single case studies are usually performed on critical, extreme, unique, or representative cases. Even though single case studies are sometimes criticized due to fears about uniqueness surrounding the case and incapability of providing a generalizing conclusion, they are considered acceptable, provided they met the established objective [17, 24].

5.1. Internal validity

One threat to the internal validity of the performed case study is the fact that we did not verify the actual meaning of the domain terms in the source code of the software libraries. We assumed that the domain terms have the same meaning in the source code as attributed to them in the domain vocabulary. Even though this is very likely to be true because of the specific domain we are analyzing, only a manual verification can guarantee the validity in this case. Such a verification would be very difficult and prone to errors in the absence of the original developers of the software.

Stemming can also influence the results of our case study, as it can cause overfitting of the data by determining terms with different meanings to be mapped to the same stem. More precise stemmers may reduce this issue. We do not believe that the results would be significantly changed by using a different stemming technique.

Another aspect to be considered is the manual choice of the domain concepts and the manual selection of the domain vocabulary. In selecting the domain vocabulary we were influenced by the information sources considered (i.e., the books and articles taken into consideration) and by our understanding of the domain. It is likely that if other people would construct the domain vocabulary for graph theory, it would be slightly different than what we came up with. We refined the domain vocabulary several times before agreeing on its final form. For example, we first considered the term "length" as representing the length of a cycle or path and included it in the domain vocabulary. When we refined the domain vocabulary, we realized that this term is likely to occur in almost any software as it is hardly graph theory specific, so we eliminated it from the domain vocabulary. As mentioned before, the use of a domain ontology may alleviate this issue in future studies.

We must also keep in mind that the software domain vocabularies are not exactly summaries of the problem domain they implement because each system (as the results show) implements a somewhat different part of the domain. However, one would expect under the circumstance even less lexical agreement between two libraries. We believe that if we measure the agreement between two systems that implement the same part of the domain, the agreement would be stronger.

5.2. External validity

In order to support the external validity of the case study and assess the role of chance in the measures we computed, we also counted how many of the graph theory domain terms also occur in other systems. We selected three other libraries from three different domains (see Table 6) for this purpose. The results show that the number of graph theory terms (5% on average from the total domain terms) and concepts (6% on average from the total domain concepts) occurring in these software libraries is significantly smaller than in the graph theory libraries. Even if we attribute 5% of the results to chance, rather than programmer choice, the results convey the same message.

Table 6. The software libraries from domains other than graph theory and the number of graph theory terms and concepts they implement.

Library	**MoneyJar**	**Ftp4che**	**Jasypt**
Domain	Financial	FTP	Security
Language	Java	Java	Java
No unique filtered and stemmed terms	623	599	557
No graph theory terms	8 (4%)	11 (6%)	8 (4%)
No of graph theory concepts	9 (5%)	15 (8%)	12 (7%)

We have no indication that the results can be generalized outside graph theory or even to more complex systems than these libraries.

6. Conclusions and future work

The presented case study analyzed the use of domain terms in a set of six software libraries, with the purpose of answering a set of research questions regarding the use of domain terms in the source code. The results indicate that the domain terms are significantly and consistently used in each and across the software libraries. A set of 18 domain terms, representing main concepts in graph theory, is shared by all the graph theory libraries considered.

The case study also shows evidence that comments and identifiers both represent a significant source of domain terms, which can aid a programmer in the task of program comprehension. At the same time, when the quality and quantity of the provided comments is satisfactory, they tend to include almost all the domain terms present in a system. Some tools that support comprehension tend to ignore comments on grounds that they may not be in synch with the source code and use only identifiers. Based on our findings, we argue that comments may be more significant than identifiers when it comes to reflect domain information.

In addition, the case study revealed a strong lexical agreement between pairs of libraries with respect to the use of domain terms. This indicates that programmers tend to be more consistent with each other than other people when they describe concepts. This is an important finding as it supports the idea that developers familiar with the domain will have an easier time understanding source code in the same domain, even when written by others.

While the identifiers and comments have been studied extensively in the program comprehension community (see the related work section), the main novelty of our work is the study of lexical agreement between different software libraries with respect to the use of domain terms.

Overall, we can make a recommendation that domain information should be explicitly captured during development (maybe in the form of ontologies) and shared among developers and especially with newcomers to the project.

Our future work will mainly address the current limitations of our case study and the threats to its validity. More case studies, in a variety of domains will be performed, with the purpose of observing common, domain-independent patterns in the use of domain terms in source code. A more detailed analysis of the use of domain terms in the source code will be performed, including a mapping between the terms and their meaning in the source code. We plan to include other software artifacts in our studies such as requirement documents, user manuals, bug reports, etc.

7. Acknowledgements

This research was supported in part by a grant from the US National Science Foundation (CCF-0438970). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

8. References

[1] Anquetil, N. and Lethbridge, T., "Assessing the Relevance of Identifier Names in a Legacy Software System", in *Proceedings of the Annual IBM Centers for Advanced Studies Conference (CASCON'98)*, Toronto, Ontario, Canada, December 1998, pp. 213-222.

[2] Antoniol, G., Gueheneuc, Y.-G., Merlo, E., and Tonella, P., "Mining the Lexicon Used by Programmers during Software Evolution", in *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*, Paris, France, 2007, pp. 14-23.

[3] Baysal, O. and Malton, A. J., "Correlating Social Interactions to Release History during Software Evolution", in *Proceedings of the 4th International*

Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, 2007, pp. 7-15.

[4] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "The Concept Assignment Problem in Program Understanding", in *Proceedings of the 15th IEEE/ACM International Conference on Software Engineering (ICSE'94)* May 17-21 1994, pp. 482-498.

[5] Caprile, C. and Tonella, P., "Nomen Est Omen: Analyzing the Language of Function Identifiers", in *Proceedings of the 6th Working Conference on Reverse Engineering*, Atlanta, Georgia, USA, 6-8 October 1999, pp. 112-122.

[6] Copeck, T. and Szpakowicz, S., "Vocabulary Agreement Among Model Summaries And Source Documents", in *Proceedings of the 4th Document Understanding Conference (DUC'04)*, Boston, USA, 2004.

[7] Deissenboeck, F. and Pizka , M., "Concise and Consistent Naming", *Software Quality Journal*, 14, 3, 2006, pp. 261-282

[8] Etzkorn, L. H., Bowen, L. L., and Davis, C. G., "An Approach to Program Understanding by Natural Language Understanding", *Natural Language Engineering*, 5, 03, 1999, pp. 219-236.

[9] Etzkorn, L. H., Davis, C. G., and Bowen, L. L., "The Language of Comments in Computer Software: A Sublanguage of English", *Journal of Pragmatics*, 33, 11, 2001, pp. 1731-1756.

[10] Flyvbjerg, B., "Five Misunderstandings About Case-Study Research", *Qualitative Inquiry*, 12, 2, 2006, pp. 219-245.

[11] Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T., "The Vocabulary Problem in Human-System Communication", *Communications of the ACM*, 30, 11, 1987, pp. 964-971.

[12] Høst, E. W. and Østvold, B. M., "The Programmer’s Lexicon, Volume I: The Verbs", in *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*, Paris, France, 2007.

[13] Jaccard, P., " Étude comparative de la distribution florale dans une portion des Alpes et des Jura." *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37, 1901, pp. 547-579.

[14] Lawrie, D., Morrell, C., Feild, H., and Binkley, D., "What's in a Name? A Study of Identifiers", in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, June 14-16 2006, pp. 3-12.

[15] Malt, B. C., Sloman, S. A., Gennari, S., Shi, M., and Wang, Y., "Knowing versus Naming: Similarity and the Linguistic Categorization of Artifacts", *Journal of Memory and Language*, 40, 1999, pp. 230–262.

[16] Ohba, M. and Gondow, K., "Toward Mining "Concept Keywords" from Identifiers in Large Software Projects", in *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR'05)*, St. Louis, MO, USA, May 2005, pp. 1-5.

[17] Perry, D. E., Sim, S. E., and Easterbrook, S. M., "Case studies for software engineers", in *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, 2006, pp. 1045-1046.

[18] Porter, M., "An Algorithm for Suffix Stripping", *Program*, 14, 3, July 1980, pp. 130-137.

[19] Ratiu, D. and Deissenboeck, F., "From Reality to Programs and (Not Quite) Back Again", in *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Canada, 2007, pp. 91-102.

[20] Ratiu, D. and Jürjens, J., "The Reality of Libraries", in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007, pp. 307-318.

[21] Reiter, E. and Sripada, S., "Squibs and Discussions: Human Variation and Lexical Choice", *Computational Linguistics*, 28, 4, 2002, pp. 545 - 553.

[22] Sayyad-Shirabad, J., Lethbridge, T. C., and Lyon, S., "A Little Knowledge Can Go a Long Way Towards Program Understanding", in *Proceedings of the 5th International Workshop on Program Comprehension (WPC'97)*, 1997.

[23] Takang, A., Grubb, P., and Macredie, R., "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation", *Journal of Programming Languages*, 4, 3, 1996, pp. 143-167.

[24] Tellis, W., "Introduction to Case Study", *The Qualitative Report*, 3, 2, 1997, pp. 1-11.