Source Code Indexing for Automated Tracing

Anas Mahmoud
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
amm560@msstate.edu

Nan Niu
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
niu@cse.msstate.edu

ABSTRACT
Requirements-to-source-code traceability employs information retrieval (IR) methods to automatically link requirements to the source code that implements them. A crucial step in this process is indexing, where partial and important information from the software artifacts is converted into a representation that is compatible with the underlying IR model. Source code demands special attention in the indexing process. In this paper, we investigate source code indexing for supporting automatic traceability. We introduce a feature diagram that captures the key components and their relationships in the domain of source code indexing. We then present an experiment to examine the features of the diagram and their dependencies. Results show that utilizing comments has a significant effect on traceability link generation, and stemming is required when comments are considered.

Categories and Subject Descriptors
D.2 [**Software Engineering**]; H.3.1 [**Information Systems**]: Content Analysis and Indexing

General Terms
Experimentation

Keywords
Traceability, indexing, source code analysis, information retrieval

1. INTRODUCTION
Requirements-to-source-code traceability, automated through information retrieval (IR) methods, has been proven beneficial in several software engineering tasks such as verification and validation, software reuse, and change impact analysis [1-3]. The process consists of three steps: *indexing*, *retrieval* and *presentation*. In the indexing step, input artifacts, including source code and requirements documentation, are converted into more compact forms that are compatible with the underlying IR model. These forms are known as *profiles* [4]. In the retrieval step, IR algorithms are used to match a trace query profile with source code profiles and identify a set of candidate links in response to the query. In the presentation step, retrieved candidate links are presented to the human analyst for further validation.

Recovering traceability links between source code and textual documents via IR methods has been tackled in the literature. For example, Antoniol *et al.* [1] used tf-idf (a vector space IR model) and unigram approximation (a probabilistic IR model) to establish traceability links between object-oriented code and functional requirements, and Marcus and Maletic [2] exploited latent semantic indexing (LSI) for automatic documentation-to-source-code traceability links recovery. The research is primarily concerned with the generation of candidate traceability links with only little attention paid to indexing.

In software engineering research, source code indexing is usually employed in two main areas: code retrieval (searching) and code comprehension (understanding). The indexing process itself depends on the task at hand. For example, in code comprehension tasks, such as code summarization [5] and code clustering [6], indexing is used to convert source code into more understandable forms that help developers maintain their code [7]. For IR-related tasks, indexing is used to convert source code into more compact forms, i.e., profiles. A profile is a short-form description of an artifact, is easier to manipulate than the entire artifact, and plays the role of a surrogate at the retrieval stage [4].

In this paper, we investigate source code indexing for automated tracing. We identify the main aspects of the indexing process by reviewing the literature, and then present the results in a feature diagram [8]. The feature diagram captures the common and variable components and their dependencies in the source code indexing domain, and organizes the knowledge in a tree-like structure. We also conduct an experiment, using two different datasets: *eTour* [9] and *iTrust* [10], to examine the features of the diagram and their dependencies. We attempt to answer the question: Which aspects of the source code and which indexing practices have a significant effect on traceability?

Section 2 of this paper provides background information. Section 3 presents a feature diagram about source code indexing. Section 4 describes the experiment conducted to examine the feature diagram. Section 5 presents and discusses the results. Section 6 describes the threats to validity. And finally, Section 7 concludes the paper and suggests potential research directions.

2. BACKGROUND
Document indexing is defined as the task of assigning terms to documents for retrieval purposes [11]. The process consists of two generic steps: extracting the subject matter of a document, and expressing the subject matter in index terms to facilitate subject retrieval [12]. Search engines, which search large repositories of textual documents such as digital libraries or the Web, rely heavily on indexing to increase their retrieval efficiency and effectiveness.

Several techniques from linguistics, natural language processing, informatics and mathematics are used to index free text documents [13]. Although source code can be treated as plan text, the restricted nature of programming languages limits the ability of generalized natural language indexing techniques to perform well when applied to source code. The characteristics of source code that make its indexing a challenging task include:

- *Formality*: Source code is highly structured. Developers have to follow strict syntactic rules in order to produce a working code.
- *Naming style*: There is no guarantee that developers will use genuine words in their code, or follow a well-defined naming convention throughout the project life cycle. In most cases, developers use a combination of words and abbreviations to name their identifiers [14].
- *Reserved words*: The majority of the words in source code are programming language reserved words that have no direct relation to the problem domain.
- *Comments*: Comments have a different nature from source code and need to be processed separately [7].

In the literature, source code indexing is often described as a process that takes several steps. The input of the process is a source code document, and the output is a compact content descriptor, or a profile, which is usually represented as keywords-components matrix or a vector space model [15]. The process starts by extracting tokens from source code. Lexical analysis is then applied to extract genuine words from these tokens. Stop words are filtered out. Finally, stemming is used to remove morphological and inflexion endings [5, 16-18]. Next, we review each of these steps in more detail.

2.1 Information Extraction

Domain knowledge and code concepts are embedded in the linguistic aspects of source code including identifiers names and comments [14, 19, 20]. Code identifiers, such as names of classes, attributes, methods, and parameters, capture developers' understanding of their tasks. The underlying assumption is that developers name their identifiers in a way that is related to the functionality of source code, and not completely at random [4]. For example, an identifier named `user_id` is expected to represent a user's identification information.

The other source of knowledge in the source code is the comments. Comments serve as the internal documentation of the code. In the literature, utilization of comments in code indexing has generated some debate. The argument that supports using comments is based on the fact that programmers tend to focus on the functionality of the code with only little attention paid to its style, and so, there is no guarantee that the naming style used by the developers will be good enough to capture the domain concepts [21]. However, comments are commonly written in a language similar to that of the external documentation. Developers add comments to explain and communicate their code. Therefore, comments are expected to carry valuable information that should not go to waste [4].

Argument against using comments is also supported by several observations, such as not all source code contains comments, quality of comments and their levels of abstraction vary widely among software systems, even sometimes within the same system. Comments might be outdated or even redundant to the source code [22]. As an example, in the following line of code, comments add no value to the code concept:

```
Increment++; //incrementing by 1
```

In the literature, the benefits of utilizing comments have been stated clearly in domains like code comprehension and software reuse. For example, in their program comprehension study, Vinz and Etzkron suggest that combining comments with source code allows for much deeper understanding of source code than is possible using either code or comments/identifiers alone [7]. Also, Takang *et al.* reported that commented programs are more understandable than non-commented programs [23]. In [24], Etzkorn and Davis introduced *Patricia*, a system that uses heuristic methods to identify reusable software components through understanding comments and identifiers. Similarly, *CodeBroker* utilizes knowledge from comments and code identifiers to find software components that can be reused [25].

In the traceability literature, however, there is no consensus on whether to use the comments or not. Antoniol *et al.*, in their study of tracing object-oriented code to functional requirements, did not include comments as part of the analysis [1]. On the contrary, Marcus and Maletic utilized comments in addition to the source code to recover traceability links between documents and source code via LSI [2]. An interesting finding in their study is that with almost no comments in the source code, LSI performed at least as well as the other methods [2].

Finally, the non-linguistic aspects of source code, such as the inheritance relations, can also be utilized as potential sources of information. Utilizing the non-linguistic aspects is beyond the scope of this paper.

2.2 Lexical Analysis

Lexical analysis is used to extract meaningful words from code tokens. A "token" is defined to be any alphabetical sequence of characters separated by non-alphabetical characters or by letter capitalization [26]. It is a common practice to define identifiers by concatenating two or more words [1]. Such identifiers can be broken down into units based on commonly used coding standards, such as the location of the capital letter in the identifier name ("`firstName` → `first name`") or any other separators such as the underscore ("_") or the dash ("-") character.

Abbreviations are also commonly used by programmers to name identifiers [14]. Domain specific dictionaries or lookup tables can be used to expand abbreviations to the constituent words. For example: `hsptlRcrd` → `hsptl rcrd` → `hospital record`.

2.3 Filtering

The goal of indexing in IR is to generate a set of index terms that achieve the best performance with IR methods. Stop words are any words that are irrelevant to the code concept. Such words carry a very low information value and can affect the retrieval process negatively [15]. We identify four types of stop words that are usually filtered out of the code profiles.

- *Generic stop words*: stop words that are used in natural language, such as (*and, but, the*). A list of the most common stop words in English can be found in [27].
- *Programming language specific stop words*: the set of keywords reserved by the programming language, such as (*`integer, string, class, static`*). These words have no direct relation to software features.

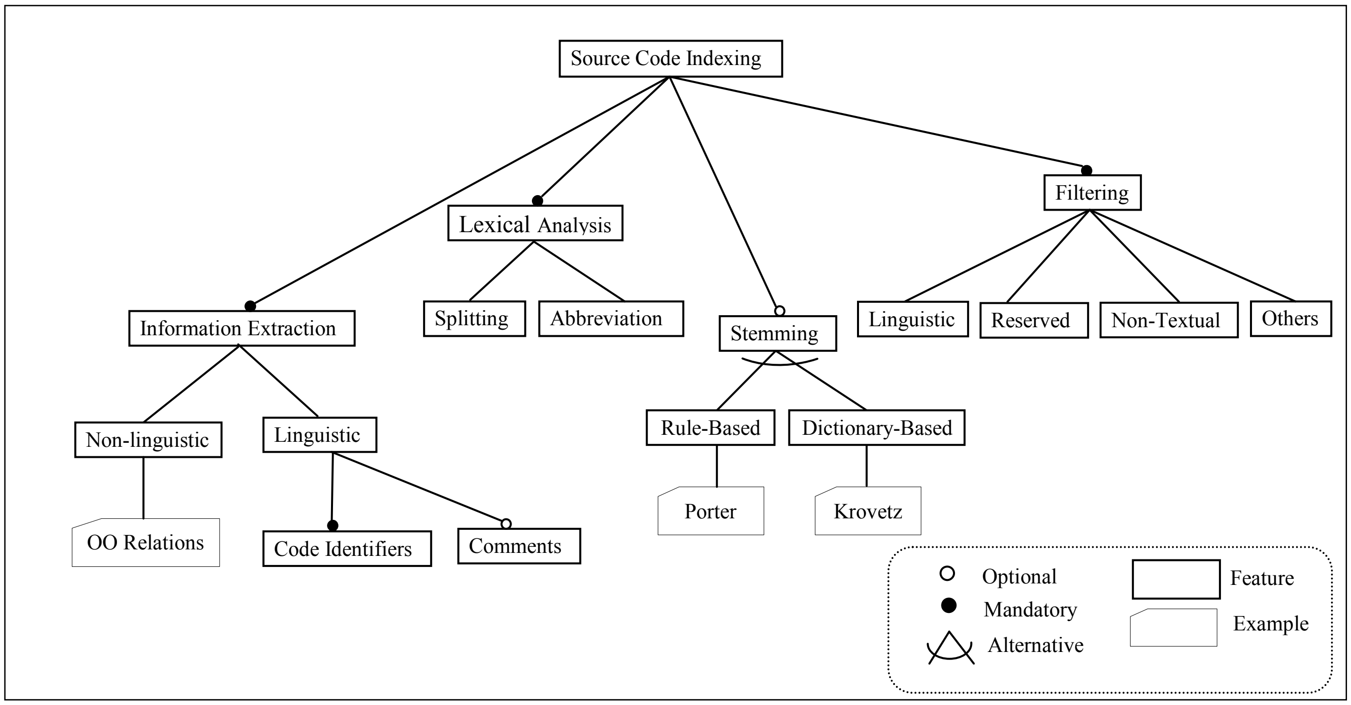


Figure 1. A feature diagram for source code indexing

- *Non-textual tokens*: set of language operators and special characters which are used to perform certain functions, such as (+,−, %, @).
- *Other stop words*: Sometimes developers include some tokens in their comments that are used throughout the project as references, such as (*Author, Param, Date, Copyright notice or license terms*). These tokens have no information value and can be removed.

2.4 Stemming

Stemming is the process of reducing a word to its root. It is a commonly used IR technique to reduce required resources by only keeping one representation for each word. Stemming enhances the matching rate by reducing terms with the same meaning into a single term, therefore, improving the effectiveness and the efficiency of the retrieval system [28].

Several stemming approaches have been applied in IR research. Among these, rule-based stemming is one of the most popular. Rule-based stemming uses a large number of language-specific rules to reduce words to their canonical morphological representations. Porter algorithm [29] is one of the most employed rule-based stemmers in IR research.

Rule-based stemming is simple to implement and maintain, and has a modest computational cost. However, its quality depends highly on the set of rules applied. Also, its performance may downgrade when dealing with irregular cases such as “eat” and “ate”. To overcome this problem, a dictionary-based approach is sometimes used. This approach mainly involves maintaining known morphological word roots that exist as real words in a lookup table. Krovetz’s stemmer [30] is an example of a dictionary-based English stemmer where potential root forms are contained in the dictionary.

The use of stemming in IR-related tasks does not come without risk. It has been observed that as words get stemmed they lose an important part of their meaning. This leads to information loss that can have a negative effect on retrieval precision [28]. This risk becomes more obvious in free text retrieval, where free vocabularies and grammars are used to form sentences and give them their logical meanings. But, is this the case in source code? It has actually been observed that code identifiers and comments are usually expressed in a simplified form of the natural language, with a smaller vocabulary set and simplified grammar [31]. Etzkorn and Davis [24] studied several software packages and found that around 83% of the comments written in sentence form are in the present tense, and around 78% of identifiers names are noun-related. These observations raise questions concerning the effect of using stemming when indexing source code for traceability recovery.

3. FEATURE DIAGRAM

Figure 1 shows a feature diagram that is a result of our analysis of the source code indexing domain. Our domain analysis is concerned with identifying the variabilities and commonalities of approaches in the code indexing domain, thereby developing and organizing an information infrastructure to support knowledge reuse. A feature diagram is a hierarchy of common and variable features characterizing the set of instances within a domain. It helps in determining the scope of the domain and provides an external view that the stakeholders can understand and communicate easily [8]. In our case, the feature diagram provides a taxonomy and representation of design choices for approaches dealing with code indexing for traceability.

Figure 1 depicts a feature diagram we use as a basis for our discussion. It is important to note that we do not aim for this domain characterization to be immune from change. In fact, we expect this knowledge representation to evolve as our understanding of code indexing matures. The experiment we describe in the next section is an attempt to further our understanding empirically. Our main goal here is to show the vast range of available choices as represented by the current code indexing approaches from a reuse perspective.

Figure 1 follows the notations defined in [8]. The features (denoted by the boxes) of the concept *source code indexing* are described, which is located at the top of the feature diagram. The boxes directly connected to *source code indexing* are the direct sub-features or sub-steps. The little circles at the edges connecting the features define the semantics of the edge. A filled circle means mandatory. Thus, every code indexing shall perform *information extraction*, *lexical analysis*, and *filtering*. Because whether to include *comments* and to perform *stemming* have generated some debate in the literature, they are identified as optional features currently (denoted by the outlined circle at the edge). Alternative features means an exclusive-or choice, so when stemming is performed in practice, a rule-based or dictionary-based stemmer is applied.

4. EXPERIMENTAL DESIGN

Experimentally validating all the features in a feature diagram and identifying all their possible dependencies can be tedious [32]. In our experiment, we chose comments and stemming as our independent variables as they were marked as optional features in Figure 1. We attempt to answer the questions: Should comments be considered when tracing source code? Is stemming required?

4.1 Datasets and Variables

We used two datasets in our experiment: *eTour* and *iTrust*. *eTour* is an electronic touristic guide developed by final year students at the University of Salerno (Italy) [9]. *iTrust* is a medical application, developed by software engineering students at North Carolina State University (USA), which provides patients with a means to keep up with their medical history and records and to communicate with their doctors [10]. Table 1 shows the characteristic of each dataset. The table shows the size of the system in terms of lines of source code (LOC), lines of comments (COM), the number of use cases (UCs), the number of source code classes (CCs) and number of correct links between use cases and code classes. Traceability links were provided in both datasets' documentation [9, 10].

Table 1. Experiment datasets

	LOC	COM	UCs	CCs	Links
eTour	17.5K	7.5K	58	116	394
iTrust	18.3K	6.3K	38	226	314

As mentioned earlier, our independent variables are comments and stemming. We used Porter stemmer [29] for its computational efficiency. Our dependent variable focuses on the quality of the automatically generated candidate requirements-to-source-code traceability links. For that, we used well-known IR metrics: *precision* and *recall* [15]. Recall (R) is a *coverage* measure and refers to the proportion of relevant links that are retrieved. Precision (P) is an *accuracy* measure and refers to the proportion of retrieved links that are relevant. In the traceability literature, the automatic trace generation approaches have emphasized recall over precision, i.e., it is essential to automatically retrieve close to 100% of the related artifacts. The assumption is that it would be easier for human analysts to discard the incorrect traceability links than to discover the missing links. To emphasize the recall, we use the F_2 measure, which weights recall twice as much as precision. The general F_β is described as:

$$F_\beta = \frac{(1 + \beta^2) \cdot (P \cdot R)}{(\beta^2 \cdot P + R)}$$

The controlled variables in our experiment include the Porter stemmer, a list of stop words [27] and reserved programming language keywords, and the vector space IR model *tf-idf*. Formally, if Q and W are two artifacts' profiles in the vector space, then their similarity is measured as:

$$S = \frac{\sum_{i=1}^{n} w_i \cdot q_i}{\sqrt{\sum_{i=i}^{n} w_i^2 \cdot \sum_{i=i}^{n} q_i^2}}$$

where $w_i = tf_i(f) \cdot idf_i$, $qi = tf_i(q) \cdot idf_i$. $tf_i(f)$ and $tf_i(q)$ are term frequency of term$_i$ in Q and W respectively. idf_i is the inverse document frequency, and is computed as $idf_i = log_2(t/df_i)$, where t is the total number of profiles in the corpus and df_i is the number of profiles in which term$_i$ occurs. *tf-idf* is one of the most commonly used IR methods in the automated tracing literature, e.g., [1, 3]. Its performance is comparable to other models, such as latent semantic analysis [2] and probabilistic networks [33].

4.2 Experimental Settings

To answer our research questions, we identify four experimental settings with all the possible "comments" and "stemming" combinations. We control the rest of the features shown in Figure 1 for their effect. These settings are summarized in Table 2: a tick (✓) indicates the feature is selected; a cross (×) indicates otherwise.

Table 2. Experiment settings

Case	Information Extraction		Stemming	Lexical Analysis	Filtering
	Code	Comments			
C	✓	×	×	✓	✓
CS	✓	×	✓	✓	✓
CC	✓	✓	×	✓	✓
CCS	✓	✓	✓	✓	✓

Base case analysis (C)

The base case in our experiment includes indexing source code only. Code identifiers are extracted and lexically processed, stop words are filtered out, no comments are considered and no stemming is performed. This case represents a reference point for comparing other case's performance.

Stemming the source code (CS)

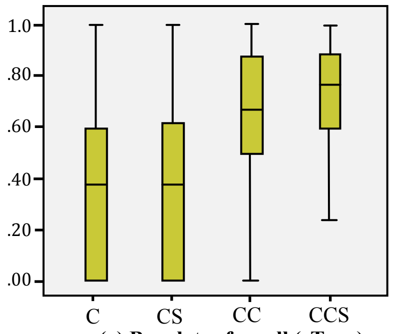
To investigate the effect of stemming source code, all source code profiles generated in the base case are stemmed using Porter's algorithm.

Considering comments (CC)

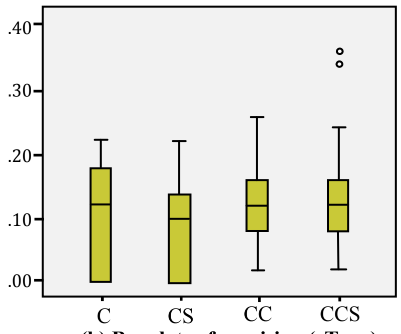
In this case, source code in our datasets is indexed with comments. The comments, in addition to the code identifiers, are extracted and lexically processed. All irrelevant stop words are removed and no stemming is performed.

Stemming comments (CCS)

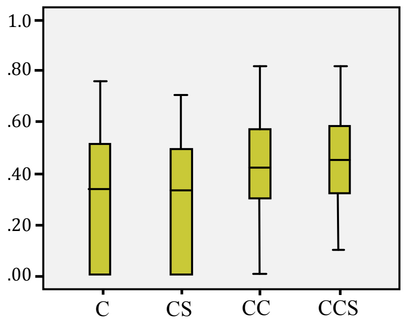
In the last case, all CC profiles from the previous case are stemmed using Porter's algorithm.



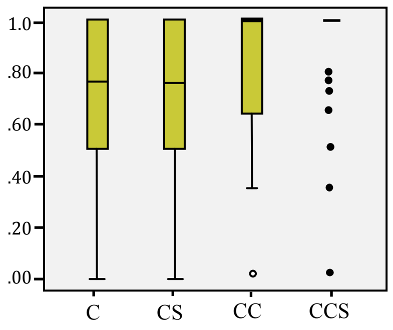
(a) Boxplots of recall (eTour)



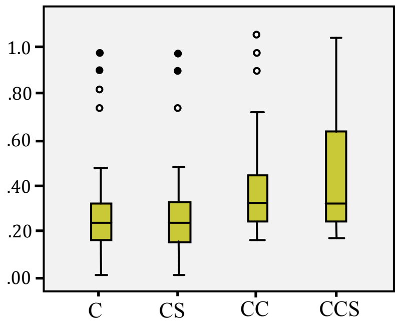
(b) Boxplots of precision (eTour)



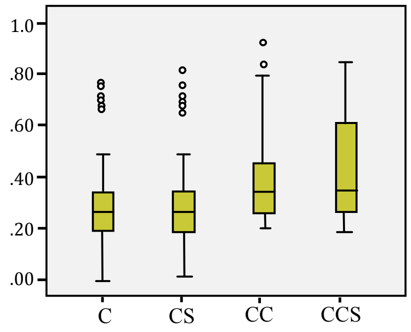
(c) Boxplots of F_2 measure (eTour)



(d) Boxplots of recall (iTrust)

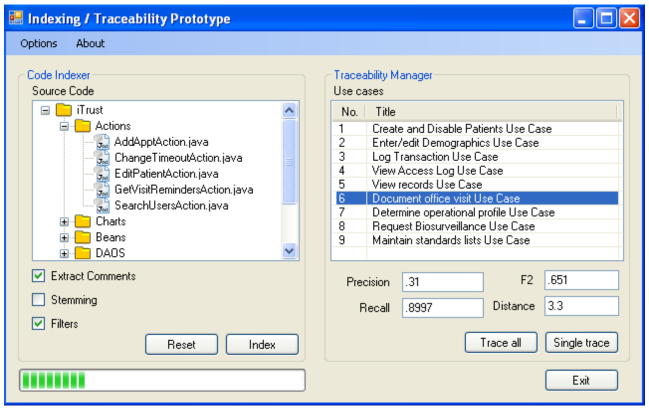


(e) Boxplots of precision (iTrust)



(f) Boxplots of F_2 measure (iTrust)

Figure 3. Descriptive statistics for the quality of automatically generated traceability links



[✓] Extract Comments
[ ] Stemming
[✓] Filters

No.	Title
1	Create and Disable Patients Use Case
2	Enter/edit Demographics Use Case
3	Log Transaction Use Case
4	View Access Log Use Case
5	View records Use Case
6	Document office visit Use Case
7	Determine operational profile Use Case
8	Request Biosurveillance Use Case
9	Maintain standards lists Use Case

Precision .31 F2 .651
Recall .8997 Distance 3.3

Figure 2. The experiment prototype

4.3 Tool Support

A prototype was implemented to carry out the experimental analysis in this paper. Figure 2 shows a screenshot of the tool. The prototype has two main functions: a code indexer and a requirements-to-source-code tracer. The code indexer uses regular expressions to match and capture identifiers and comments in a source code file. Porter algorithm is used to perform stemming, and generic and programming language specific stop words lists are provided to filter out irrelevant terms. The prototype also has a control panel to allow the user to control the settings of the indexing process, such as whether to include comments or to do stemming. After performing indexing, all the generated profiles are stored in the artifacts database to be used later in the tracing process. For each dataset, we traced all the requirements (use cases) to code classes. Tracing is performed by *tf-idf*. The gold standards from the project repositories [9, 10] are used to evaluate the quality of the automatically generated traceability links.

Table 3. ANOVA results for traceability link quality

(a) eTour

Experiment Settings	Recall		Precision		F_2	
	F	***Sig***	***F***	***Sig***	***F***	***Sig***
C x CS	56.267	.057	.663	.428	.004	.949
C x CC	80.83	.000	5.71	.019	21.06	.000
CC x CCS	19.267	.000	3.371	.151	3.17	.083

(b) iTrust

Experiment Settings	Recall		Precision		F_2	
	F	***Sig***	***F***	***Sig***	***F***	***Sig***
C x CS	1.00	.325	2.076	.161	1.181	.301
C x CC	9.212	.005	11.260	.002	22.828	.000
CC x CCS	6.82	.014	3.36	.060	7.083	.047

5. RESULTS AND DISCUSSION

This section presents the data collected during the experiment and our quantitative data analysis. We reported descriptive statistics in boxplot and inferential statistics via ANOVA (analysis of variance) [34]. A boxplot reveals much about the data: its dispersion, its center, and how skewed the data is. Side-by-side boxplots quickly illustrate the relationships of these characteristics for multiple data distributions. Figure 3 shows the boxplots that describe the quality of automatically generated traceability links under our experimental settings. ANOVA is a collection of statistical models and procedures in which the observed variance is partitioned into components due to different explanatory variables. We used the 0.05 alpha level (α=0.05) to test the significance of difference among our experimental settings. All results were observed at a 70% threshold, a widely used heuristic that achieves an acceptable compromise between precision and recall [35]. Under this threshold, only the top 70% of the ranked candidate links were considered in the analysis; the remaining 30% were ignored. Table 3 shows the ANOVA results for traceability link quality.

5.1 Should Comments Be Indexed?

To assess the effect of indexing comments on the generated links quality, all use cases in both datasets were traced to the CC profiles. The performance, in terms of recall, precision, and F_2, was compared to the base case performance (C), where no comments were considered.

Analysis shows that considering comments in the indexing process has a significant effect on the retrieval effectiveness. In both datasets, the recall, precision, and the F_2 measure were improved significantly. Take recall as an example: in Figure 3a, there is a significant shift of the median between the C and CC boxes.

5.2 Should Stemming Be Performed?

Two cases were considered to test the effect of stemming on traceability link quality. In the first case, all use cases in both systems were traced to the stemmed source code profiles (CS). The goal was to test the effect of stemming source code identifiers on the results. The performance was compared to the base case (C) performance. Using analysis of variance, no significant difference was detected in terms of recall, precision, and F_2 (CS = C). This leads to the conclusion that if only source code is considered in the indexing process, then no stemming is required.

In the second case, comments profiles were stemmed (CCS). The results were compared to the CC (unstemmed comments) case. The analysis of variance shows that stemming comments has improved the recall significantly in both datasets ($CCS_{Recall} > CC_{Recall}$). However, the average precision was affected negatively for the *eTour* dataset, and showed no significant improvement for the *iTrust* dataset. The negative effect on the precision could be that stemming causes loss of information [28], which results in retrieving more irrelevant links. The effect on F_2 was statistically significant in *iTrust* only.

The analysis shows that, for both datasets, when stemming is applied to comments, it improves the recall significantly. However, if only the code is to be used (for example in cases where the code is not commented), then stemming is unnecessary. As mentioned earlier, developers do not use fancy language in naming their identifiers; they usually stick to the base form of the word, which limits the effect of stemming when dealing with source code identifiers. However, comments are usually written with more freedom and in complete sentences. Even though simplified grammars are usually used, stemming comments is still able to improve the recall. To further confirm these findings, we observed the percentage of terms affected by stemming in both datasets. We found that the percentage of comments terms affected by stemming was 15% and 23% in *eTour* and *iTrust* respectively. However, the percentage of code terms affected by stemming was only 4.2% and 4.7% in *eTour* and *iTrust* respectively.

5.3 Summary

To summarize our findings, we refer to the feature diagram presented in Figure 1. We chose to examine the optional features “comments” and “stemming” since their application in source code indexing has generated some debate. In the traceability literature in particular, it is not uncommon to exclude comments when source code is indexed, e.g., [1]. Even if comments are included, its effect seems only marginal [2]. The results of our experiment show a strong tendency of including comments in the code indexing process.

As for stemming, we would still keep it an optional feature. However, we would add a “requires” dependency link from “comments” to “stemming” in the feature diagram. This indicates that if comments are considered, then stemming is required. In fact, whether to perform stemming or not is a tradeoff between recall and precision. In cases where recall is favored over precision, the recommendation is to use stemming when the decline in precision is not statistically significant [28].

6. THREATS TO VALIDITY

Several factors can affect the validity of our study. As for construct validity [34], we feel that the independent variables (“comments” and “stemming”) and dependent variables (precision, recall, and F_2) accurately measure the concepts they purport to measure: variabilities in the code indexing process for the independent variables, and quality of automatically generated traceability links for the dependent variables. To address internal validity [34], we use a 2x2 factorial design (cf. Table 2) to consider all the combinations of the two independent variables while keeping the configuration of the remaining factors unchanged across the settings.

The subject systems and the controlled variables in our experiment can pose threats to external validity [34]. In particular, the results of this study might not generalize beyond the object-oriented software systems, the requirements-to-source-code traceability, the *tf-idf* retrieval method, the Porter stemmer and the lists of stop words employed, etc. Several strategies are used in our experiment to help mitigate these threats. First, we choose a representative stemmer and stop word lists to experiment, so that the results are informative about the experiences of the typical situation. Second, extensive empirical studies have shown that the quality of automatically generated traceability links is almost equivalent, independent of the underlying IR method [36]. Therefore we expect that our results will hold even though different IR models are employed. Third, we experiment two midsize datasets, *eTour* and *iTrust*, from two different domains, and find converging results. This helps generalize our findings to other application domains. However, both *eTour* and *iTrust* were developed by university students and may not be representative of a program written by industrial professionals. It is therefore unknown if the results will generalize to other software systems, other application domains, or larger systems.

7. CONCLUSION

In this paper, we have tackled the problem of indexing source code for supporting requirements-to-source-code traceability link generation. We introduced a feature diagram to describe the indexing process, and conducted an experiment using two datasets, *eTour* and *iTrust*, to examine some of the diagram features and their dependencies. The results showed that considering comments in the indexing process helps improve the traceability link quality significantly. Stemming was also found useful when comments were considered. However, if comments were ignored then the overhead of stemming is unnecessary.

The findings of our experiment emphasize the importance of adopting a good naming convention while writing code. Meaningless names or abbreviations cause low similarity between requirements and source code. The results also emphasize the importance of considering comments. Commented code is not only more understandable, but also easier to be traced. Future work for this study will focus on validating other aspects and levels of the feature diagram using large-scale industrial datasets.

8. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation", *TSE*, 28(10): 970-983, 2002.

[2] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing", *ICSE*, pp. 125-135, 2003.

[3] J. H. Hayes, A. Dekhtyar, and S. K.Sundaram, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods", *TSE*, 32(1): 4-19, 2006.

[4] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries", *TSE*, 17(8): 800-813, 1991.

[5] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "Supporting Program Comprehension with Source Code Summarization", *ICSE - NIER Track*, pp. 223-226, 2010.

[6] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: identifying topics in source code", *IST*, 49(3): 230-243, 2007.

[7] B. L. Vinz and L. H. Etzkorn, "Improving program comprehension by combining code understanding with comment understanding", *KBS*, 21(8): 813-825, 2008.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study", Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.

[9] eTour, http://www.cs.wm.edu/semeru/tefse2011/.

[10] iTrust, http://agile.csc.ncsu.edu/iTrust/wiki/doku.php.

[11] N. Fuhr and C. Buckley, "A probabilistic learning approach for document indexing", *TOIS*, 9(3): 223-248, 1991.

[12] J. E. Mai, "Analysis in indexing: document and domain centered approaches", *IJPM*, 41(3): 599-611, 2005.

[13] G. Salton and M. E. Lesk, "Computer evaluation of indexing and text processing", *Journal of the ACM,* 15(1): 8-36*,* 1968.

[14] D. Lawrie, H. Feild, and D. Binkley, "Extracting Meaning from Abbreviated Identifiers", *SCAM*, pp. 213-222, 2007.

[15] C. D. Manning, P. Raghavan, and H. Schutze, *An Introduction to Information Retrieval*, Cambridge University Press, Cambridge, England, 2008.

[16] A. Michail and D. Notkin, "Assessing software libraries by browsing similar classes, functions and relationships", *ICSE*, pp. 463-472, 1999.

[17] L. Cerulo, "On the Use of Process Trails to Understand Software Development", Ph.D. Thesis, Università Degli Studi Del Sannio, Italy, 2006.

[18] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code", *WCRE*, pp. 214-223, 2004.

[19] N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system", *CASCON*, pp. 4-14, 1998.

[20] A. Marcus and D. Poshyvanyk, "The Conceptual Cohesion of Classes", *ICSM*, pp.133-142, 2005.

[21] H. M. Sneed, "Object-Oriented COBOL Re-cycling", *WCRE*, pp. 169-178, 1996.

[22] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories", *JSS*, 79(7): 939-953, 2006.

[23] A. Takang, P. Grubb, R. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation", *JPL,* 4(3): 143–167, 1996.

[24] L. H. Etzkorn and C. G. Davis, "Automatically Identifying Reusable OO Legacy Code", *Computer*, 30(10): 66-71, 1997.

[25] Y. Ye and G. Fischer, "Reuse-Conducive Development Environments", *ASE*, 12(2): 199-235, 2005.

[26] S. Ugurel, R. Krovetz, and C. L. Giles, "What's the code?: automatic classification of source code archives", *SIGKDD,* pp. 632-638, 2002

[27] C. Fox, "A stop list for general text", *ACM SIGIR Forum*, 24(1-2):19-21, 1990.

[28] G. Kowalski, *Information Retrieval Architecture and Algorithms*, Springer, 2011.

[29] M. F. Porter, "An algorithm for suffix stripping", In *Readings in Information Retrieval*, Morgan Kaufmann, 1997.

[30] R. Krovetz, "Viewing morphology as an inference process", *SIGIR*, pp. 191-202, 1993.

[31] L. H. Etzkorn and C. G. Davis, "A documentation-related approach to object oriented program understanding", *IWPC*, pp. 39-45, 2002.

[32] J. Guo and Y. Wang, "Towards Consistent Evolution of Feature Models", *SPLC*, pp. 451-455, 2010.

[33] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best Practices of Automated Traceability", *IEEE Computer*, 40(6): 27-35, 2007.

[34] C. Wohlin, P. Runeson, M. *Höst, M. C.* Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

[35] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Enhancing an artefact management system with traceability recovery features", *ICSM*, pp. 306 – 315, 2004.

[36] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery", *ICPC*, pp. 68-71, 2010.