Improving Bug Localisation Using Lexical Information and Call Relations

Tezcan Dilshener Michel Wermelinger Yijun Yu
Computing and Communications Department
The Open University, United Kingdom

***Abstract*—To comprehend an unfamiliar application while implementing a change request, software developers perform lexical search and navigate the application's structures as separate activities, while many methods need to be inspected to decide which path to take to locate the target. Instead of checking large number of individual methods, we address this challenge for developers by focusing on the fewer class names and class call relations. Our approach integrates lexical information retrieval with structural program dependency search to present a ranked list where the relevant classes for a change request are located in the top-*N* positions. Evaluated with six applications from five different domains, our integrated approach succeeded in finding relevant classes, with simple queries formulated from terse descriptions, on average for 72% of the change requests. Compared to a state-of-the-art tool, our tool outperformed it in 19% of the cases while performing just as well in 72% of them. These results emphasize the need for combined tool support in effectively exploring applications during software maintenance.**

Keywords—business software maintenance; lexical search; call relations; change requests; domain vocabulary; empirical study.

I. INTRODUCTION

As valuable and strategic assets to companies and playing a central role for the business, software applications require continued and substantial effort to be maintained. Often, software companies turn towards third-party developers to take over the maintenance tasks [9]. A high turnover among these developers and the impermanent nature of their employment cause the knowledge of the applications to move further away from its source. Each time a new developer is designated to perform a maintenance task, the associated high learning curve results in loss of precious time, incurring additional costs. As the documentation and other relevant project artefacts decay [11], to understand the current state of the application before implementing a change the designated developer has to go through the complex task of understanding the code.

Generally, program comprehension during software maintenance can cause additional effort for those developers who have little domain knowledge. Early attempts to aid the developers in recovering traceability links between source code and documentation utilised Information Retrieval (IR) methods like the Vector Space Model (VSM) [8]. They achieved high recall, but due to low precision they required manual effort to evaluate the results [3]. To address the shortcomings of VSM, researchers applied Latent Semantic Indexing (LSI) models that resulted in higher precision values compared to VSM [14]. Nevertheless, these probabilistic IR approaches do not consider terms that are strongly related via structural information and thus still perform poorly in some cases [18].

The current research recognised the need for combining multiple analysis approaches on top of IR to support program comprehension [15]. To determine the starting points in investigating relevant program elements (i.e. classes and methods) during maintenance work, techniques combining dynamic [16] and static [20] analysis have been exploited [18]. These combined approaches first obtain a dynamic trace of the involved classes by re-running the scenarios described in the Change Request (CR) (e.g. bug descriptions or feature requests). From there on, they attempt to retrieve other relevant elements based on the call relations by navigating the static call-graph of the application. Some of the challenges faced by these approaches are that, on one hand, the CR documents may be tersely described or may be for a new non-existing feature [17], making them unsuitable for dynamic analysis [19]. On the other hand, the complex class method call relations (i.e. nodes and edges) found in the static call-graph may cause a lot of noise in the results [24]. Therefore it is still claimed that no single IR method consistently provides a superior recovery of traceability links between program elements and CRs [25].

Our aim is to identify the opportunities of using the CR vocabulary and the application call relations to address the challenges faced by the current combined approaches as well as to reduce the program comprehension overhead. To trace a CR to the source code, we first search the source code of an application using the vocabulary extracted from the CRs and then refine the search results by utilizing the additional clues provided by the call relations. It is acknowledged that during maintenance, developers perform search tasks using lexical information as well as navigate the structural information [22], thus we also combine these information sources. Compared to existing work on traceability between code and CRs, our unique contribution is in proposing a novel scoring system based on lexical similarity and call-relations. More precisely, we attempt to answer the following research questions:

RQ1: *Does utilizing a combined approach based on lexical information and call relations improve the search performance with respect to a simple lexical string search*?

We conducted two types of searches, in both cases using the terms extracted from the CR descriptions. Since developers search for program elements, e.g. classes, using lexical match [12], the first search uses simple lexical string match by comparing the query terms with those extracted from the class.

The second search assigns a score to the classes on the result list. The score is based on where the search terms occurred in the class and based on the call relations to the neighbouring classes. We compare the effectiveness of both search techniques to show the improvements obtained by our scoring method.

RQ2: *How does the combined approach, implemented in our tool, perform compared to another state-of-the-art tool?*

We compare the search performance of our tool with an existing bug localisation tool. For this, we have first conducted traceability searches with the tool presented in [10] to replicate the results of that study. We have then conducted the same searches on the same data sets with our tool. Finally, we compared the results obtained from both tools, highlighting the benefits provided by supplementing the lexical search with call relations.

The rest of this paper is organized as follows: Section II explains the proposed approach, Section III describes the data collection procedure and analysis steps, Section IV presents the results in answering the research questions, and we discuss their implications to program comprehension in Section V. We describe related work in Section VI. Finally, Section VII concludes with additional remarks.

II. METHODOLOGY

A. Traceability through Information Retrieval

In a typical Information Retrieval (IR) approach for traceability between an application's source code and its textual documentation, like the CR documents, the current techniques first analyse the project artefacts and build an abstract high level representation of an application in a repository referred as the *corpus* where the program elements, e.g. classes, can be searched.

The existing IR techniques extract the traceability information from class names, method signatures and identifiers by parsing the source code and storing the extracted information in the corpus. During the extraction process the identifier names are transformed into individual terms according to known OOP coding styles like the camel case naming pattern where, for example, identifier *standAloneRisk* is split into *stand, alone* and *risk*. After this transformation, the resulting terms are stored in the repository and indexed with a reference to their locations in the source code files.

During a search, the underlying IR method, e.g. VSM or LSI, compares the terms entered in the query against the terms found in the indexed corpus. The matching score is calculated based on lexical similarity or probabilistic distance between the terms, depending on the IR method being used. To improve the ranking, one of the methods that existing approaches [22] utilise is to enhance the IR scoring by navigating the call-graph of the application and evaluating the relevance of the neighboring methods. Finally, the results are displayed to the developer in a list ranked by their relevance to the search.

B. Our Approach

As the current literature highlights [26], the IR approaches may consider two strongly related terms via structural information (i.e. OOP inheritance) to be irrelevant. In turn this may cause one method to be at the top of the result list and the other related one at the bottom. To cater for these structural relations, the static call-graph (CG) is utilized. However, the call-graphs usually contain many dependencies and very deep nodes/edges that make them impractical for search and navigation purposes [24].

To address these shortcomings, we propose a ranking approach that utilises a novel scoring system based on lexical similarity and application-only call-graph, where call-relations to 3rd party libraries are excluded. Our approach groups the relevant classes that are already found in a result list nearer to each other so that those relevant for the search remain within the top-*N* position of the list. This is achieved by applying a simple scoring to the found classes based on where the search terms appear, i.e. on the class name or in the class body. The score is further improved by exploiting the call relations in the application i.e. the number of calls coming in and going out of the found class.

After creating a corpus from the application's source code, including the call-relations between classes and the class terms (i.e. the terms extracted from the identifiers within a class), our approach works as described in the following steps.

(1) Search for classes using the search terms extracted from the CR's description. For each class, we iterate over the list of search terms and check if any of them match the class name or class terms.

(2) Assign a score to the class based on where the search terms occur: in the class name, in the list of class terms, or the searched term is exactly the same as the class name. The class and its score are added to a list, as the pseudo code in Fig 1 illustrates.

```
Function: relevantClassScoring()
Inputs: list of class name with terms; search terms
Output: ranked list of relevant classes
---
for each class with term list {
    score = 0.0;
    for each search term {
        if (search term == class name)
            score += 1.0000f;
        else if (search term occurs in the class name)
            score += 0.0250f;
        else if (search term exists in class term list)
            score += 0.0125f;
    }
    add class name and score to ranked result list
}
enhanceClassScoring(ranked result list);
```

Fig. 1. Procedure for class scoring by Lexical match.

(3) For each class in the ranked list, we adjust the score based on the call-relations, by navigating the call-graph to the immediate neighboring called or calling classes. Only those classes that are already in the ranked list are considered, to avoid introducing false positives in the

context of a CR, where a subset of classes are usually relevant. The score of an already ranked class is re-calculated: if many other classes call it then it is a utility function (decrease score), otherwise it has core functionality (increase score), as Fig 2 illustrates.

```
Function: enhanceClassScoring()
Inputs: list of ranked classes; call relations between classes
Output: improved list of ranked classes
---
for each ranked class r {
    if (r has a caller/callee call-relation)
        for each class c that call(c, r) or call(r, c) {
            if (c is in the ranked list)
                if (number of call-relations of c < 10)
                        // called/calling class has core functionality
                        score of c += 0.0525f;
                else
                        // called/calling class has utility function
                        score of c -= 2.5000f;
        }
}
```

Fig. 2. Procedure for improving the ranking results by call relations.

(4) Sort the list using the new score and discard the absolute score value. We will use top-10 as the cut-off point when measuring the performance of our approach, to make it comparable with existing work. The top-10 list includes only the classes whose score caused them to be placed between the 1st and 10th positions.

In cases where a lexical score cannot be assigned by *relevantClassScoring* due to the query terms not matching the terms extracted from the class name and identifiers, a probabilistic score is obtained by repeating the search with the IR model VSM. This score qualifies the class to be processed by the *enhanceClassScoring* function where the call-relations are evaluated as described in Step 3.

The rationale behind the scoring values is that class names are treated as the most important elements and are assigned the highest weight. So when a query term is identical to a class name, it is considered a 100% match and an arbitrary score of 1.0000 is assigned to it. Otherwise, the occurrence of the query term on the class name is considered to have higher importance over the terms extracted from the identifiers or method signatures. In those cases, arbitrary scores of 0.0250 and 0.0125 are assigned, respectively. Subsequently, to further optimise the grouping of relevant classes nearer to each other in the ranked result list, the call relations are evaluated. During this stage the score of core functionality classes are boosted by a factor of 0.0525 while those with utility function are reduced by a factor of 2.5000. Finally, the derived score is used for relative ranking and the absolute score value is dismissed. There is no magic in choosing the values for these factors: our algorithm was run using a small size dataset of manually selected CRs from each application, and we made adjustments to the scores to optimize the results for that training dataset.

III. Case Study Design

In order to evaluate the effectiveness of our approach, we analysed the source code of six applications addressing five independent domains: finance, integrated development environment (IDE), aspect-oriented programming, graphical user interface (GUI) and bar-code.

A. Subject applications

For the finance domain, we used two financial applications, one open source (OSS), referred to as Pillar-One, and one proprietary, due to confidentiality referred as Pillar-Two. Both applications implement the financial regulations for credit and risk management defined by the Basel-II[1] Accord [4]. Pillar-One[2] is a client/server application developed in Java and Groovy by Munich Re (a re-insurance company). The CR documents are maintained with JIRA[3], a public tracking tool. Pillar-Two is a web-based application developed using the Java programming language at our industrial partner's site and is not publicly available. It has been in production for 4 years. The maintenance and further improvements are undertaken by five developers, including in the past the first author, none of them part of the initial team. The CR documents are maintained by a proprietary tracking tool.

For the other four domains we used the same applications investigated by the authors of [10] allowing us to compare our tool with theirs: the IDE tool Eclipse[4], the aspect-oriented programming library AspectJ[5], the GUI library SWT[6] and the bar-code tool ZXing[7]. Eclipse and AspectJ are well-known large scale applications used in many empirical research studies for evaluating various IR models [30]. SWT is a subproject of Eclipse and ZXing is an Android project maintained by Google[8]. Table I shows the size of the project artefacts for all of the applications.

TABLE I. Project Artefacts Size

Application	Source files	CRs	Call relations
AspectJ	6485	286	3203
Eclipse (v3.1)	12863	3075	81177
Pillar-One (v1.6)	4355	26	7827
Pillar-Two (v7.0)	337	12	407
SWT (v3.1)	484	98	1268
ZXing	391	20	982

B. Data collection

We obtained the complete code and CR for all applications. The CRs were already implemented for the versions of the applications being analysed. In case of Pillar-Two, the affected source code files (Java classes) are already listed in the CR documents. In case of Pillar-One, the CR documents included

1. http://www.bis.org
2. http://www.pillarone.org
3. https://issuetracking.intuitive-collaboration.com
4. http://www.eclipse.org
5. http:// www.st.cs.uni-saarland.de/ibugs/
6. http://www.eclipse.org/swt/
7. http://code.google.com/p/zxing/
8. https://source.android.com/

to the source code repository commits reference numbers. By manually analysing these references in the repository, we have listed all the affected classes per CR in one document. In case of the other four applications, i.e. Eclipse, AspectJ, SWT and ZXing, we are grateful to Zhang *et al.* [10] for providing their data sets, which included for each application the source code and a document containing all CRs with affected classes.

C. Corpus preparation

The obtained project artefacts, i.e. the source code files and the CRs, were processed using our source code analysis framework tool called *ConCodeSe* (Contextual Code Search Engine). We implemented the proposed scoring and ranking approach (Section II) in ConCodeSe by extending our previous work [2]. ConCodeSe utilises state of the art data extraction, persistence and search APIs (SQL, Lucene[9], Hibernate[10], JIM [5]). Figure 3 illustrates the extraction, storage, search and analysis stages. In the top layer, the corpus creation and search services tasks are executed automatically.

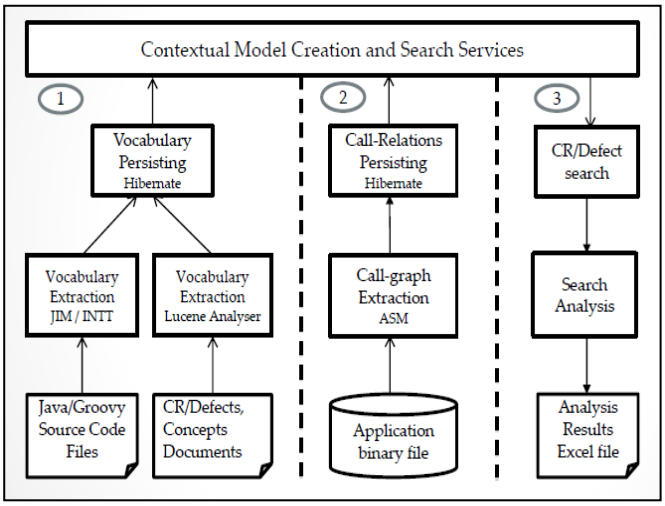

Fig. 3. ConCodeSe Data Extraction, Storage and Search.

The left hand side (1) represents the extraction and storage of terms from the source code files and from the CRs. The middle part (2) shows the extraction and storage of the call-graph information from the application binary files. Finally, in the search stage (3), the search for the classes affected by the CRs takes place. The search results are saved in a spreadsheet for additional statistical code analysis like the Spearman correlation coefficients tests [6].

1a. Source code vocabulary processing.

In the first stage, the Java and Groovy sources are parsed using the source code-mining tool JIM, which automates the extraction and analysis of identifiers from source files. It parses the source, extracts the identifiers and splits them into terms. During this step, the identifiers and metadata from the source code abstract syntax tree are extracted and added to a central store, with information about their location. Also, the tool INTT [10, 12] within JIM is used to tokenise and split the identifier names into terms. INTT uses camel case, separators and other heuristics to split ambiguous boundaries, digits and lower cases. The extracted information, i.e. the identifier names, their tokenisation and source code location, is stored in a Derby[11] relational database.

1b. Textual documentation vocabulary processing.

Also for the first stage, we developed a Java module using the Lucene framework to tokenise the text in the CR documents into terms. The module reuses Lucene's *Standard-Analyzer* class because it tokenises alphanumerics, acronyms, company names, and email addresses, etc. using a JFlex-based lexical grammar. It also includes stop-word removal. We used a publicly available stop-words list[12] to filter them out. The extracted information is stored via the Hibernate persistence API in the same Derby database.

2. Call-relations processing.

For the second stage, we developed a Java call-graph construction module using the ASM[13] tool. ASM is a simple API for decomposing, modifying, and recomposing binary Java classes. Our module reads the classes contained in the binary file of the application and builds a list of called classes (*callee*) and a list of calling classes (*caller*). Both the *caller→callee* and the *callee→caller* relations are stored in the same Derby database for use in the search stage to improve the scoring of the ranked classes in the result list. The constructed call-graph is an application-only call-graph [24] where calls to external libraries are ignored and only the call relations between the classes within the application packages are considered.

3. CRs Search.

For the third stage of the process, we developed a search module in ConCodeSe that runs SQL queries to search (1) for the occurrences of the terms in the project artefacts and (2) for all the relevant classes of a CR. The manually identified classes affected by each CR are used to compute precision and recall of the search results. *Recall* measures the completeness of the results and *precision* measures the accuracy of the results.

IV. Evaluation Results

Our approach considers the immediate neighbouring classes in the call graph and introduces a scoring technique to determine the relevance of a class to the search query terms. In [25] it is concluded that exploring 1 to 2 edges and top-5 to top-10 gives the best results. Therefore to further compare the effectiveness of our approach with the proposed method in [10], we used top-*N* ranking. In the rest of the paper, we consider a class to be located if it was ranked in the top-*N* for some *N*, and we consider a CR to be located if at least one of its affected classes was located.

***RQ1**: Does utilizing a combined approach based on lexical information and call relations improve the search performance with respect to a simple lexical string search?*

Our first aim was to check whether simple string searching with CR terms is sufficient to find the affected classes. The

9. http://lucene.apache.org/java/docs/index.html
10. http://www.hibernate.org
11. http://db.apache.org/derby
12. http://norm.al/2009/04/14/list-of-english-stop-words/
13. http://asm.ow2.org/

search resulted in very poor performance values: on average, less than 20% of an application's CRs were located, as shown in the 'lexical match' column of Table IV. Since the source code is consulted during maintenance, this lack of good agreement between the code's and the CRs' vocabulary points to potential inefficiencies during maintenance.

We analysed possible reasons for this and found that CR descriptions of the open source developer oriented applications, i.e. Eclipse and AspectJ, expose a lot of indications to the program elements i.e. class names, which aids their traceability to the source code. However, in case of the two business oriented applications, i.e. Pillar-One and Pillar-Two, the CR descriptions written by users or support desk team members are very terse (see Table II). This leads to inefficiencies when attempting to locate the relevant classes required to implement a CR. Also, in certain cases, some of the class names and the CR descriptions reflect the search terms explicitly, which helps finding the affected classes more accurately, for example in case of Pillar-Two CR #2074 a possible search term '*lambda'* occurs in the class names and identifiers as shown in Table II. However in Pillar-One CR #1619, neither the class names nor the CR reflects a possible search term '*poisson'* (see Table II).

TABLE II. SAMPLE TERM OCCURANCES IN ARTEFACTS

CR	Description	Affected Classes	Identifier
2074 (P2)	Dialog to distribute lambda factors similar to other module	HelperDistributeLambda RiskLambda ProcessorCopyLambda HelperCopyLambda	calculateLambdas, readLambda, hasLambdaDiversify lambdaFactors
1619 (P1)	unrecoverable error for error parameter poisson.	LognormalTypeIIPareto TypeIIParetoDistribution	alphaAndLambda, muAndLambda

In general, either the action-oriented nature of the CRs did not provide clues to the search terms being addressed directly or the class names of the applications did not reflect the search terms explicitly. We attempt to address these issues using the combined approach. We started our experiments with Pillar-Two because of our industrial experience with it. For example, CR #2093 has 6 affected classes and the lexical search returned 61 classes, with the 6 ranked as shown in Table III. The table also shows the improved ranking after using the call relations (Fig. 2).

TABLE III. SAMPLE RANKING WITH LEXICAL AND CALL-RELATIONS SCORING

Pillar-Two classes affected by CR # 2093	Lexical ranking	Lexical + call relations rank
QuotaShareContractStrategy.java	2	3
EventAalLimitStrategy.java	31	1
LimitStrategyType.groovy	33	2
EventLimitStrategy.java	4	4
NoneLimitStrategy.java	38	10
ILimitStrategy.java	5	5

Table IV shows the overall comparison of the lexical-only vs the combined approach for $N = 10$. The percentage of CRs located, i.e. with at least one affected class ranked in the top-10, increased considerably for every application and we can therefore answer RQ1 positively.

TABLE IV. NUMBER OF CRS FOUND BY VARIOUS SEARCH TYPES

Applications	CRs	Simple lexical match	Lexical scoring + call relations
AspectJ	286	32 (11%)	178 (62%)
Eclipse	3075	357 (12%)	1651 (54%)
Pillar-One	26	5 (19%)	15 (58%)
Pillar-Two	12	6 (50%)	11 (92%)
SWT	98	25 (26%)	82 (84%)
ZXing	20	0 (0%)	16 (80%)
Average		**19%**	**72%**

In certain cases, we observed that none of the search terms extracted from the CR description are found in the source code of a class. Consequently, in those cases the class at hand could not be used as the entry point to navigating the call-graph to re-rank its position in the result list. In such situations, searching by VSM model as described in Section 2.*B* provided the needed leverage. As illustrated in Table V, for CR # 103862 of SWT, 2 out of the 6 affected classes are ranked in the top-10.

TABLE V. SAMPLE CR CLASSES WITHOUT CALL-RELATIONS

SWT CR# 103862 Subset of classes	ConCodeSe Ranking	
	Lexical	With VSM
SWT.java	Not found	10
Composite.java	Not found	21
Display.java	Not found	9

Overall, we learned that searching for classes affected by a CR is different from searching for the classes implementing a concept because the CRs tend to have more action-oriented characteristics and their unit-of-work may cross multiple concepts. Hence a CR may document what the users experience on the User Interface, whereas, in the background other program elements may be of relevance. In other words, classes implementing different concepts may be relevant within the context of a CR. Our approach addresses these challenges by combining multiple sources of information, i.e. CR vocabulary and call relations to provide an improved search tool for the developers during maintenance tasks.

RQ2*: How does the approach, implemented in our tool, perform compared to a state-of-the-art tool?*

To compare the search performance of ConCodeSe when compared to an existing bug localisation tool, we have

performed the following tasks. First, we have conducted searches with BugLocator [10] and obtained the same results as reported on that study using their data set. Second, we have conducted the same searches using the data set of [10] with our tool. We als run both tools on our own data sets, Pillar-One and Pillar-Two. Finally, we compared the results obtained from both tools.

BugLocator implements an approach similar to ours. It is initiated with a list of closed bug reports that reference the effected files. First, it creates an indexed corpus of terms extracted from the source code files. Secondly, it searches the corpus for the relevant classes using the terms found in a CR. The result produced is a ranked list of files based on textual similarity between the queried terms from the CR and the terms extracted from the source code files. The ranking is obtained by combining two different VSM similarity calculations (i.e. how source code file terms are matched with the terms found in the bug reports). Those classes that are ranked within the top-1 or top-10 are considered effectively localised. Based on this, to compare the performance of BugLocator and our tool, we also consider the classes ranked in top-1 and top-10 as effectively localised if they match the ones listed in CR.

TABLE VI. THE LOCALISATION PERFORMANCE OF BOTH TOOLS

Applications	CRs	Top-1		Top-10	
		Bug Locator	Con CodeSe	Bug Locator	Con CodeSe
Aspectj	286	65 (23%)	67 (23%)	186 (65%)	159 (56%)
Eclipse	3075	749 (24%)	770 (25%)	1719 (56%)	1615 (53%)
Pillar-One	27	5 (19%)	7 (26%)	10 (37%)	14 (52%)
Pillar-Two	12	2 (17%)	6 (50%)	6 (50%)	10 (83%)
SWT	98	31 (32%)	54 (55%)	76 (78%)	81 (83%)
ZXing	20	8 (40%)	6 (30%)	14 (70%)	17 (85%)
Average		**26%**	**35%**	**59%**	**69%**

Table VI shows the number and percentage of CRs localised by each tool for each *N*. On average our tool provides a 10 percentual point improvement (35% over 26% and 69% over 59% respectively) over BugLocator. For example, in case of SWT ConCodeSe has put an affected class in the top position for 55% (54/98) of the CRs, whereas BugLocator only achieved it for 32% (31/98) of the CRs. The improvement in performance over BugLocator was also noticeable in the top-10 ranking category, for example in case of ZXing 85% with ConCodeSe compared to 70% with BugLocator. However, in case of Eclipse and AspectJ ConCodeSe performs only slightly better in the top-1 ranking category.

In addition, for both tools, we calculated the recall performance, i.e. the number of classes that gets into the top-1 and top-10 positions per CR. After all, putting more relevant classes in the top-*N* positions is bound to be more beneficial for the developers. Table VII shows that on average for 19% and 30% of the CRs our tool has located more relevant classes in the top-1 and top-10 positions respectively than BugLocator. Also our tool performs just as well for 72% and 54% of the CRs in top-*N* cases while performing worse in 9% and 16% respectively. For example, for AspectJ, there were 39 CRs where ConCodeSe ranked an affected class in the first position and BugLocator didn't, 23 CRs where BugLocator ranked an affected class in the first position but ConCodeSe didn't, and for the remaining CRs, either both tools ranked an affected class first or both didn't.

TABLE VII. CONCODESE RECALL PERFORMANCE

Applications	Better		Same		Worse	
	top-1	top-10	top-1	top-10	top-1	top-10
AspectJ	39 (14%)	84 (29%)	224 (78%)	161 (56%)	23 (8%)	41 (14%)
Eclipse	325 (11%)	381 (12%)	2220 (72%)	1839 (60%)	529 (17%)	854 (28%)
Pillar1	5 (19%)	10 (37%)	20 (74%)	14 (52%)	2 (7%)	3 (11%)
Pillar2	4 (33%)	8 (67%)	8 (67%)	2 (17%)	0 (0%)	2 (17%)
SWT	29 (30%)	21 (21%)	59 (60%)	60 (61%)	10 (10%)	17 (17%)
ZXing	1 (5%)	3 (15%)	16 (80%)	15 (75%)	3 (15%)	2 (10%)
Average	**19%**	**30%**	**72%**	**54%**	**9%**	**16%**

In some cases BugLocator performs better than our tool because it considers the comments of the source code. We also tried to consider the comments and repeated the same search tasks. As shown in Table VIII, on average the performance has improved from 72% to 80% for top-1 and from 54% to 70% for top-10 positions in locating just as many relevant classes as BugLocator for the CRs. Consequently, the performance advantage over BugLocator in the same top-1 and top-10 positions has dropped from 19% to 10% and from 30% to 19% respectively while the poor performance values increased from 9% to 15% and from 16% to 25% respectively.

TABLE VIII. CONCODESE RECALL PERFORMANCE WITH COMMENTS

Applications	Better		Same		Worse	
	top-1	top-10	top-1	top-10	top-1	top-10
AspectJ	24 (8%)	57 (20%)	231 (81%)	174 (64%)	31 (11%)	55 (19%)
Eclipse	171 (6%)	221 (7%)	225 (72%)	1689 (55%)	678 (22%)	1164 (38%)
Pillar1	3 (11%)	8 (30%)	20 (74%)	15 (56%)	4 (15%)	4 (15%)
Pillar2	1 (8%)	4 (33%)	9 (75%)	8 (67%)	2 (17%)	0 (0%)
SWT	21 (21%)	17 (17%)	63 (64%)	63 (64%)	14 (14%)	18 (18%)
ZXing	1 (5%)	1 (5%)	16 (80%)	14 (70%)	3 (15%)	5 (25%)
Average	**10%**	**19%**	**80%**	**70%**	**15%**	**25%**

Overall, we found that catering for the comments tends to produce noise so the recall (i.e. number of affected classes ranked in the top-*N*) deteriorates. For example, in case of the

SWT CR #84906, our approach fails to rank any classes whereas BugLocator ranks one class in the top-10. When comments are considered, ConCodeSe finds the same relevant class as BugLocator, in position 30. Since this is not a significant advantage, we decided to leave comments for future work.

TABLE IX. AGGREGATE RECALL RESULTS FOR THE SIX APPLICATIONS

Statistics	BugLocator		ConCodeSe	
	Top-1	Top-10	Top-1	Top-10
Average	0.28	0.89	0.38	1.25
Median	0	0.8	0.3	1.2
Std. Dev	0.44	0.95	0.53	1.27

Furthermore, we used the non-parametric Wilcoxon matched pairs test to statistically validate the outcome of our study since the results follow a non-standard distribution. Based on the values obtained (Z=-3.0594, W=0 and p=0.00222), we conclude that on average ConCodeSe locates significantly ($p \leq 0.05$) more relevant classes in the top-N positions than BugLocator (e.g. 1.25 vs. 0.89) as shown in Table IX.

Finally, we compared the Mean Average Precision (MAP) values of both tools. MAP provides a single-figure measure of quality across recall levels. Among evaluation measures, MAP has been shown to have especially good discrimination and stability [32]. MAP is calculated as the sum of all the average precision values for each CR divided by the number of CRs for a given application. Average precision is the precision value obtained for each relevant class listed in the top-N set. BugLocator utilises a similarity score that captures the similarity of previously reported and implemented CRs to help facilitate localisation of relevant classes for a new CR. Since our tool does not consider similarity among CRs like BugLocator does, to make a fair comparison we have taken the MAP values reported in [10] for CRs without similarity consideration.

TABLE X. MEAN AVERAGE PRECISION OF BOTH TOOLS

Applications	CRs	BugLocator MAP	ConCodeSe MAP
AspectJ	286	0.17	0.33
Eclipse	3075	0.26	0.30
Pillar-One	26	0.18	0.26
Pillar-Two	12	0.28	0.39
SWT	98	0.40	0.59
ZXing	20	0.41	0.43

As the results in Table X show, our tool achieved significant gains particularly in case of AspectJ and Pillar-One applications where the MAP values improved almost 90% from 0.17 to 0.33 and from 0.18 to 0.26 respectively. In case of Pillar-Two and SWT applications, our tool achieved 40% improvement. However, in case of Eclipse and ZXing applications the improvements in MAP values remained minimal. Given that performance on individual queries is expected to be highly variable [26], overall performance of ConCodeSe remains superior.

V. DISCUSSION

Our study set out to explore whether supplementing lexical searches with call relationship data would noticeably improve the search accuracy revealed that it does as illustrated in Table IV. Overall, as illustrated in Tables VI and VII, we observed that enhancing the lexical scoring with call-relations leads to superior search results over the probabilistic methods (i.e. VSM). The added value of our proposed approach compared to existing work [23] is that it combines simple lexical search and call-graph navigation techniques, which improves upon the weaknesses of current approaches that use probabilistic methods like VSM or LSI where two classes may be considered irrelevant although structurally related.

Our approach, implemented in our tool ConCodeSe, improves on the result presented in [10] using the same data set, indicating that utilising simple vocabulary match with call-relations aids search performance as Table VI shows. We observed that the direct nature of call-relations facilitates search to focus on a specific CR and provides entry points to the relevant classes within the context of the CR, which supports the efforts of Petrenko *et al.* [26] on using program dependencies when navigating the call graph during a search. Another reason for the success of our approach is that utilising the call-relations groups the relevant classes together within the context of the search, which adds to the findings of Scanniello *et al* [29] on clustering i.e. grouping of classes during static bug localisation search.

It has been long argued that textual information in CR documents is noisy [28]. In addition, we found that CRs reveal two characteristics: (1) *developer nature* with technical details i.e. references to code classes or (2) *descriptive nature* with business terminology i.e. use of concept terms. This also confirms the reasons why some classes are ranked in the top-10 whereas others aren't as Table VII illustrates. In other words, CR documents may be divided into two categories, those that rely on implementation vocabulary (class names) and those that use domain vocabulary; and the two vocabularies produce different results. In both cases, the CRs tend to be of an action-oriented nature, which does not provide clues to the domain vocabulary being addressed directly or the class names do not reflect domain terms explicitly [2]. Due to these characteristics, the existing concept location techniques fall short in bug localisation when a CR is not properly declaring a concept to be searched or the class names do not reflect those related concepts.

Although multiple classes may implement a concept [1], in the data sets analysed on average only 3 classes listed as being changed. This suggests that CRs are for a unit-of-work and searching for the relevant classes using terms would find all the

classes implementing a concept. Hence in the context of a CR this would lead to many false positives. Also in certain cases, we have observed that the affected classes of a CR do not have any call-relations. Consequently, in those cases navigating the call-graph does not provide any information to re-rank relevant classes at a higher position in the list. This provides further evidence that one cannot rely on the code as the single source of information.

Since CR documents may come from a group of people who are unfamiliar with the vocabulary used in the code and in the documentation, we propose that these documents contain a section for describing the relevant domain vocabulary. For example a list of domain terms implemented by an application can be semi-automatically extracted and imported into the CR management tool. Subsequently, when creating a CR, the user may choose from the list of relevant domain terms or the tool may intelligently suggest the terms for selection. Also, to deal with crosscutting concerns as illustrated by Shepherd *et al* [13], we recommend packaging the source code classes in architectural layers based on conceptual responsibility instead of technical functionality. This would first mediate improved communication with business users, since such communications take place at a conceptual level rather than technical level, and second assist in finding relevant classes by considering the architectural relations when no call-relations exist.

In software projects, a developer may get assistance from other team members or expert users when selecting the initial entry points to perform the assigned maintenance tasks. Our tool complements this by providing a contextual model i.e. corpus with set of clues to aid program comprehension during software maintenance. It allows developers to select the initial entry points based on the ranking of the relevant classes. Thus the CR vocabulary can be seen as the assistance provided by the expert team members and the class names in the search results can be seen as the initial entry points to investigate additional relevant program elements.

A. Threats to Validity

The *construct validity* addresses whether the conclusions can legitimately be made from the operationalisation of the theories. During creation of the searchable corpus by extracting the terms form source code identifiers, we relied on the JIM tool and on the ASM framework for generating the call-relations from the binary files. Although these tools were used in previous studies [10, 12], it is possible that other tools may produce different results. Also, we used arbitrary score values to assign weight to the classes in order to rank them in the result list. Although we took all precautions to avoid bias by manually training the scoring algorithm using a small dataset, we intend to use machine learning to train the algorithm in our future work. Furthermore, we intend to consider the domain concept relations and the method call-relations in addition to class call-relations.

The *internal validity* addresses the relationship between the cause and the effect of the results to verify that the observed outcomes are the natural product of the implementation. We catered for this by comparing like for like (i.e., using the same datasets and the same criteria) the search performance of our tool with an existing bug localisation tool [10]. Therefore, the improvement in results can only be due to our approach. Also, the queries in our study were taken directly from CR descriptions. It is possible that these queries may inaccurately reflect the queries developers use or that the use of different queries with vocabularies more inline with the source code would yield better results. However, using the CR descriptions as queries instead of manually formulated queries avoided the introduction of bias from the authors.

The *conclusion validity* refers to the relationship between the treatment and the outcome and if it is statistically significant. We used a non-parametric statistical test since no assumptions were made about the distribution of the results. This test is quite robust and has been extensively used in the past to conduct similar analysis [38, 41]. The results showed that the improvement in locating relevant classes for a CR in the top-*N* position by our tool over the state of the art tool is significant. Also we used a VSM model IR engine as a fall-back option for lexical search. It is conceivable that an IR engine using the LSI model may produce more or less sensitive results. We plan to experiment with LSI in future work.

The *external validity* addresses the possibility of applying the study and results to other circumstances. The characteristics of the projects (the domain, the terse CRs, the naming conventions, the kind of documentation available) are a threat to external validity. We catered for this by repeating the search experiments with different applications addressing multiple domains for comparison. We intend to repeat the experiments with other projects and artefacts in particular with more industrial applications and with those developed in object-oriented programming languages other than Java for comparison.

Finally, with regards to the complexity of our approach and the computation times, we observed that creating the corpus by processing the application's source code, CR descriptions, call relations and searching for relevant classes for all CRs takes on average 2 minutes per application on a single high end PC (i.e. 3GHz processor and 4GB RAM), except for the Eclipse project. Due to its size (Table I), creation of the corpus takes 3 hours and ranking classes for all CRs takes about 1.5 hours, which gives an average of almost 2 seconds per CR. We consider this to be acceptable since our tool is a proof of concept and in reality developers intend to search for one CR at a time. We intend to improve the run-time performance by using multi-threaded programming in our future work.

VI. RELATED WORK

Antoniol *et al.* [3] aimed to check whether source code classes could be traced back to the functional requirements. The terms from the source code were extracted by splitting the identifier names, and the terms from the documentation were extracted by normalising the text using transformation rules. They created a matrix listing the classes to be retrieved by querying the terms extracted from the text document. The method relied on probabilistic and vector space information retrieval and ranked the documents against a query. Applying precision and recall validated their results. Although the authors compare two different retrieval methods (VSM and probabilistic), they conclude that semi-automatically

recovering traceability links between code and documentation is achievable despite the fact that the developer has to analyse a number of sources during a maintenance task to get high values of recall. Our work differs in that it uses call graphs and is geared towards maintenance, because we attempt to recover traceability between CRs and source code classes.

On the use of call relations to recover traceability links, Hill *et al.* [22] and P. Shao *et al.* [21] proposed similar approaches to ours. In their approaches a query is created with search terms, then the source code is searched for the matching methods by using the probabilistic LSI method and a term/document frequency (*tf/idf*) score is obtained. Subsequently, for each method on the result list, the call-graph of the application is utilized to evaluate the relevance of the neighbouring methods. A score is assigned based on where the query terms occur in the method names, in addition to the LSI score. Finally, both scores are combined to rank the methods into the final search result list. Our work differs in that we first perform a simple lexical search for the affected classes instead of the methods. Subsequently we improve the score by evaluating the call-relations found on the static call graph based on the number of edges coming in and going out of the neighbouring classes. In [27] it is proposed that methods calling many other methods can be seen as delegating and methods called by others as performing functionality. Finally, our approach ranks the results based on the combined score to list relevant classes in the top-10 position of the result list, as the current literature concludes that top-5 to top-10 gives the best results [25].

In their study, Petrenko *et al.* [26] presented a technique called DepIR that combines Dependency Search (DepS) with Information Retrieval (IR). The technique uses an initial query to obtain a ranked list of retrieved methods. The 10 methods with the highest ranks are selected as the possible entry points to explore the shortest path on the call-graph to the method relevant for the query. The shortest path is calculated using Dijkstra's algorithm [31] and the effort needed is calculated as the number of edges (call-relations) on the shortest path plus the IR rank. The study aims to indicate a best-case scenario estimate of the effort needed to locate the relevant method for each CR. The DepIR performance is compared against pure IR and DepS (a call-graph navigation technique) by using 5 systems and 10 CR. It is claimed that on average, DepIR required a significantly smaller effort to locate the relevant methods for each CR. The study only aims to *evaluate* the theoretical effort reduction in finding the target method by combining lexical ranking with dependency navigation. Therefore it requires the target method to be known in advance, in the same way we need to know the affected classes to evaluate our search algorithm, which doesn't require prior knowledge of what classes need to be changed. Contrary to our approach, DepIR doesn't make use of the dependencies to improve the ranking, and thus to further reduce the effort in finding the right artefacts to change.

Zhou *et al.* [10] proposed a state-of-the-art traceability tool called BugLocator that automatically searches for relevant source code files based on relevant bug reports. Utilising common bug localisation processes, the approach consists of four steps: corpus creation, indexing, query construction, retrieval & ranking. The method uses a revised Vector Space Model (*rVSM*) to rank all source code files based on an initial bug report. BugLocator has been evaluated with four open source projects and the results show that on average 60% of relevant classes reported as changed are ranked in top-10 position of the result list. For the open-source case studies, BugLocator outperformed existing VSM approaches in feature location tasks. However, the need to evaluate the tool's effectiveness with industrial projects was acknowledged. Since our work also involves industrial code, we performed the same search tasks with BugLocator in place of our tool and compared the results.

VII. Concluding Remarks

We presented a novel algorithm that, given a change request (CR) and the application's code, uses a combination of lexical and structural information to suggest, in a ranked order, classes that may have to be changed to implement the CR. The approach doesn't require test harnesses or dynamic analysis, and it can handle bug reports, feature requests, and very terse CRs.

We evaluated the algorithm with a range of applications, and found that it is vastly superior to simple string search, as performed by developers using an IDE. We compared the search results to an existing approach, using the same CRs, applications and evaluation criteria, and found that overall our approach improved the ranking of the affected classes, thereby increasing the percentage of CRs for which relevant classes are retrieved and the number of relevant classes suggested among the top-1 or top-10.

We found that including source code comments for the lexical scoring may be helpful for particular CRs, but that overall it decreases the search performance.

Our study shows that it is challenging to find the classes referred by a CR: in spite of our improvements, over 40% of CRs may not be localised (Table IV). In order to help better understand and develop new search approaches, we will offer in an online companion to this paper, the datasets of our case study as a baseline for further bug localisation research.

Acknowledgments

We thank our industrial partner, a financial service provider located in southern Germany, for providing the Pillar-Two artefacts and input on information required, and to Simon Butler for his suggestions on an earlier draft of the paper. We also thank Hongyu Zhang for kindly providing the dataset used in their case study [10].

References

[1] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proc. 15th Int'l Conf. on Software Engineering*, 1993, pp. 482-498.

[2] T. Dilshener and M. Wermelinger, "Relating developers' concepts and artefact vocabulary in a financial software module," in *27th Int'l Conf. on Software Maintenance,* 2011, pp. 412-417.

[3] G. Antoniol, G. Canfora, G. Casazza, A.D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, 28:970-983, 2002.

[4] Basel II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework - Comprehensive Version, June 2006, http://www.bis.org/publ/bcbs128.htm.

[5] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, "Exploring the influence of identifier names on code quality: an empirical study," in *14th European Conf. on Software Maintenance and Reeng.*, 2010, pp. 159–168

[6] S. Boslaugh, P. Watters, "Statistics in a nutshell", O'Reilly, 2008, pp. 176-179.

[7] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *Proc. European Conf. on Object-Oriented Programming*, LNCS 6813, Springer-Verlag, 2011, pp. 130-154

[8] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing &; Management*, vol. 24, no. 5, pp. 513-523, 1988.

[9] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proc. of the Conf. on The Future of Software Engineering*, 2000, pp. 73-87.

[10] J. Zhou, H. Zhang, and D. Lo. 2012. Where Should the Bugs be Fixed? in *Proc. 34th Int'l Conf. on Software Engineering*, 2012, pp. 14-24.

[11] M. Feilkas, D. Ratiu, and E. Jurgens, "The loss of architectural knowledge during system evolution: An industrial case study," in *17th Int'l Conf. on Program Comp, 2009*, pp. 188-197.

[12] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *25th Int'l Conf. on Software Maintenance*, 2009, pp. 157 –166.

[13] D. Shepherd, L. Pollock, and T. Tourwé, "Using language clues to discover crosscutting concerns," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1-6, 2005.

[14] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. 25th Int'l Conf. on Software Engineering*, 2003, pp. 125-135.

[15] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *27th Int'l Conf. on Software Maintenance*, 2011, pp. 133-142.

[16] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49-62, 1995.

[17] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Trans. on Software Engineering*, vol. 33, no. 6, pp. 420-432, 2007.

[18] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis," *Proc.16th Int'l Conf.on Program Comprehension,* 2008, pp. 53–62.

[19] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc, "A Heuristic-Based Approach to Identify Concepts in Execution Traces," in *14th European Conf. on Software Maintenance and Reengineering*, 2010, pp. 31–40.

[20] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static techniques for concept location in object-oriented code," in *Proc. 13th Int'l Workshop on Program Comprehension*, 2005, pp. 33-42.

[21] P. Shao and R. K. Smith, "Feature location by IR modules and call graph," in *Proc. of the 47th Annual Southeast Regional Conference*, 2009, pp. 70:1–70:4.

[22] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in *Proc. 22nd Int'l Conf. on Automated Software Engineering*, 2007, pp. 14-23.

[23] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis," in *16th Int'l Conf. on Program Comprehension*, 2008, pp. 53–62.

[24] K. Ali and O. Lhoták, "Application-only call graph construction," *European Conf. on Object-Oriented Programming*, 2012, pp. 688–712.

[25] E. Hill, L. Pollock, and K. Vijay-Shanker, "Investigating how to effectively combine static concern location techniques," in *Proc. of the 3rd Int'l Workshop on Search-driven development: users, infrastructure, tools, and evaluation - SUITE '11*, 2011, pp. 37–40.

[26] M. Petrenko and V. Rajlich, "Concept location using program dependencies and information retrieval (DepIR)," *Information and Software Technology,* vol. 55, no. 4, pp. 651–659, 2013.

[27] M. Revelle, B. Dit, and D. Poshyvanyk, "Using Data Fusion and Web Mining to Support Feature Location in Software," in *18th Int'l Conf. on Program Comprehension*, 2010, pp. 14–23.

[28] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?," IEEE Transactions on Software Engineering, vol. 36, no. 5, pp. 618 –643, sept.-oct. 2010.

[29] G. Scanniello and A. Marcus, "Clustering Support for Static Concept Location in Source Code," *19th Int'l Conf. on Program Comprehension*, 2011, pp. 1–10.

[30] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proc. 8th Wworking Cconf. on Mining Software Rrepositories*, 2011, p.43-52.

[31] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.

[32] C. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval. Cambridge University Press, 2008.