TextRank Based Search Term Identification for Software Change Tasks

Mohammad Masudur Rahman Chanchal K. Roy
University of Saskatchewan, Canada
{masud.rahman, chanchal.roy}@usask.ca

***Abstract*—During maintenance, software developers deal with a number of software change requests. Each of those requests is generally written using natural language texts, and it involves one or more domain related concepts. A developer needs to map those concepts to exact source code locations within the project in order to implement the requested change. This mapping generally starts with a search within the project that requires one or more suitable search terms. Studies suggest that the developers often perform poorly in coming up with good search terms for a change task. In this paper, we propose and evaluate a novel *TextRank*-based technique that automatically identifies and suggests search terms for a software change task by analyzing its task description. Experiments with 349 change tasks from two subject systems and comparison with one of the latest and closely related state-of-the-art approaches show that our technique is highly promising in terms of *suggestion accuracy*, *mean average precision* and *recall*.**

***Index Terms*—Concept location, TextRank, Search Term, Reverse Engineering**

I. Introduction

Studies show that about 80% of the total efforts is spent in software maintenance and evolution [19]. During maintenance, software developers deal with a number of software change requests, and identifying the exact location (i.e., class, method) within a project for a given change task is a major challenge. Change requests are often made by software users, and are generally written using natural language texts. The software users might be familiar with the application domain of a software product; however, they generally lack the idea of how a particular *software feature* is implemented in the source code. Thus a *software change request* by them mainly involves one or more domain related concepts, and a developer needs to map those concepts to the source code locations within the project in order to implement the change [15, 16]. Such mapping is possibly trivial for a developer who has substantial knowledge about the target project. However, the developers involved in maintenance might be unaware of the low-level architecture of the project, and thus they often experience difficulties in identifying the source locations (i.e., classes, methods) to be changed. The mapping task generally starts with a search within the project which requires one or more suitable search terms. Previous studies [14] suggest that on average, developers with varying experience perform poorly in coming up with good search terms for a change task. For example, according to Kevic and Fritz [14], developers can come up with relevant search terms for only 12.2% of the cases. One way to help them in this regard is to automatically suggest useful and relevant search terms for the change task in the first place.

Existing studies that attempt to support developers in *feature location* tasks with search queries, adopt different lightweight heuristics [14] and query reformulation or expansion strategies [8, 12, 20], and perform different query quality analyses [9, 10, 11] and mining activities [13, 15]. However, most of these approaches expect a developer to provide the initial search query which they can improve upon, and it is often a non-trivial task for the developers as noted by other studies too [14]. Kevic and Fritz [14] propose a heuristic model for automatically identifying initial search terms for a change task where they consider different heuristics related to *frequency, location, parts of speech* and *notation* of the terms from the task description. Although the model is found promising in their preliminary evaluation, it suffers from two major limitations. First, the model is trained using a limited number of change tasks, and is not cross-validated using the change tasks from another project. Thus it is still not quite mature or reliable. Second, *tf–idf* is one of the dominating metrics in their model, and it is subject to the size of test dataset for *inverse document frequency (idf)* calculation. Thus the same model is likely to perform differently with different sizes of test dataset, and the model might require frequent re-training to keep itself useful.

In this paper, we propose a novel *TextRank*-based technique that automatically identifies and suggests search terms for software change tasks. *TextRank* is an adaptation of *PageRank algorithm* [7] for natural language texts where a text document is considered as a lexical or semantic network of words [6, 18]. *TextRank* has found its applications in different information retrieval tasks such as keyword and key phase extraction, extractive summarization, word sense disambiguation and other tasks involving *graph-based* term weighting [18]. Given the successful adoption of that algorithm in information retrieval domain, it can also be exploited for search term extraction in the context of *feature location* tasks. Actually, the algorithm is highly suited for our purpose from several perspectives. First, software change requests are generally made by the people outside the development team, and they communicate their requirements through domain level concepts and using natural language texts. A *graph-based* representation (i.e., also called *text graph*) of the task description can reveal important semantic (i.e., co-occurrence) relationships among different terms. Second, *TextRank* algorithm exploits connectivity of a term in the graph, and considers not only the terms to which the term is connected but also their weights (i.e., importance) for determining the weight of that term. The

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

SANER 2015, Montréal, Canada

algorithm continues this process recursively, and thus has a great potential to extract the most important terms from the graph which can be suggested as search terms.

Issue ID: 401358
Product: JDT, **Component:** Debug
Summary: Name *selection* for *Mac* VM *installs* needs *improvement*
Description: When you search for a JDK/JRE on *Mac*, we use information from the plist file to compute a name. This works fine most of the time, but if you happen to have more than one of the same version of VM installed they are added with the same name. To make matters a bit worse, if you edit one of the *JREs* the wizard starts out with an error complaining that the name is already in use. The attached screen shot shows the duplicated names for the Java 7 *JREs*.

Listing 1: An Example change request from `eclipse.jdt.debug`

For example, the change task in Listing 1 can be graphically represented as the *text graph* in Fig. 1 by analyzing the *co-occurrence relationships* among the words with a *window size* (i.e., number of words considered in a semantic text unit) of two. Our proposed technique analyzes the term connectivity, calculates weight (i.e., importance) for each of the terms in the graph, and then extracts the top-scored and the most suitable five terms– *Mac, selection, installs, improvement* and *JREs* as the initial search query for the task.

Experiments with 349 change tasks from two subject systems– *Apache Log4j* and *Eclipse JDT Debug* show that our approach can return relevant results (i.e., Java classes) for 49.34% of the change tasks with 57.16% *mean average precision* and 63.33% *recall*. We also compare with one of the latest and closely related state-of-the-art approaches– Kevic and Fritz [14], and report that our approach performs relatively better in terms of all performance metrics. While our preliminary results are found promising, they must be further validated with more change tasks from more subject systems. Thus the paper makes the following technical contributions:

- It demonstrates a novel use of *TextRank* algorithm in the identification and suggestion of relevant search terms for software change tasks.
- It reports a case study involving 349 software change tasks from two subject systems and comparison with one existing approach [14], and shows the effectiveness of the proposed technique.

The rest of the paper is organized as follows – Section II explains our adopted methodology and algorithms, Section III discusses the conducted experiments and findings, Section IV describes the related work, and finally Section V concludes the paper with future work.

II. Proposed Methodology

Fig. 2 shows the schematic diagram of our proposed technique for search term identification and suggestion for a change task. In this section, we discuss the different steps involved with the technique as follows:

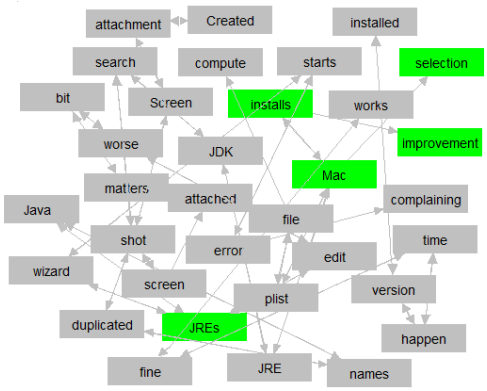

Fig. 1. Text Graph of change request in Listing 1

A. Data Collection

In order to suggest search terms for a change task, we make use of natural language description (e.g., Listing 1) for the task provided by the user. We collect actual task descriptions of 349 tasks from BugZilla official repositories. Each of those tasks is submitted using a semi-structured way, and they contain several fields such as *Issue ID* (e.g., 401358), *Product* (e.g., JDT), *Component* (e.g., Debug), *Summary* and *Description*. We use the last two fields for the analysis by our technique. While *Summary* shows the title of a requested change task, *Description* contains the user's explanation of the task in natural language texts. In order to keep the algorithm simple and lightweight, we do not consider content from the files attached to the change request. Adding more available information about the task is likely to improve the performance of our technique; however, we consider that part as a scope of future study.

B. Text Preprocessing

We analyze *Summary* and *Description* of a software change request, and perform several preprocessing before transforming the texts into a *text graph*. We consider each sentence as a *logical text unit* that contributes to the overall task description, and collect each of them from those fields. We attempt to extract such terms from a sentence that either convey useful semantics or represent the domain level concepts of the software to be changed. We thus remove stop words (i.e., frequently used non-important words) from each of those sentences, and split the dotted structured words (i.e., containing dots) into simpler words from them. A dotted word often involves multiple technical concepts, and splitting helps one to analyze each of them in isolation. For example the word– `org.eclipse.ui.part.PageBookView.createPartControl` contains a *package name* (i.e., *org.eclipse.ui.part*), a *class name* (i.e., *PageBookView*) and a *method name* (i.e., *createPartControl*), and only splitting helps to analyze those concepts effectively. It should be noted that we do not split the words based on camel-case notation given these two observations. First, change requests often contain different technical artifacts such as *library*

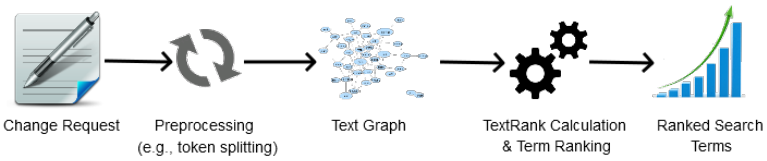

Fig. 2. Schematic diagram of the proposed technique

name, *class name* and *method name* using camel-case notations in the description, and they are of great interest to the developers implementing those changes. Splitting such words simply breaks those artifacts, and does not help in identifying suitable search terms. Second, experiments with such splitting do not result into performance improvement of our technique. We also avoid stemming on the words of the task description as it does not demonstrate any notable improvement in the performance of our technique.

C. Text Graph Development

After preprocessing, we get a list of sentences each of which contains semantically important and domain related terms, and we use them to develop a *term graph* for the task description. We represent each of those terms as a distinct node in the graph, and consider the *co-occurrence* of those terms in the sentences as an indication of semantic relationships (i.e., dependencies) among them [6, 18]. For example, if we consider the sentence– *"This works fine most of the time, but if you happen to have more than one of the same version of VM installed they are added with the same name"* from the example description (Listing 1), the preprocessed version forms an ordered list of terms– *"works fine time happen version installed"*. We note that the transformed sentence contains several phrases such as *works fine* and *version installed*, and the terms in those phrases are semantically dependent on each other for comprehensive meaning. We thus consider a *window size* of two (i.e., best performing size according to Mihalcea and Tarau [18]) as a semantic unit of words, and get the following relationships– *works*$\longleftrightarrow$*fine*, *fine*$\longleftrightarrow$*time*, *time*$\longleftrightarrow$*happen*, *happen*$\longleftrightarrow$*version*, and *version*$\longleftrightarrow$*installed*. We then encode such relationships into the connecting edges among the corresponding nodes in the graph.

D. TextRank Calculation

In order to estimate the weight (i.e., importance) of each of the terms in *text graph*, we use *TextRank* algorithm by Mihalcea and Tarau [18] which is adapted from the popular *PageRank* algorithm [7] for web link analysis. *TextRank* analyzes the connectivity details such as incoming links and outgoing links of each term in the graph recursively, and calculates its weight, $S(v_i)$, as follows:

$$S(v_i) = (1-\phi) + \phi \sum_{j \epsilon In(v_i)} \frac{S(v_j)}{|Out(v_j)|} \quad (0 \leq \phi \leq 1)$$

Here, $In(v_i)$, $Out(v_j)$, and ϕ denote the node list connected to v_i through incoming links, the node list to which v_j is connected through outgoing links, and the dumping factor respectively. In the *text graph*, each of the edges is bi-directional (i.e., terms depend on each other), and thus *in-degree* is equal to the *out-degree* for a node (i.e., term). In the context of web surfing, dumping factor, ϕ, is considered as the probability of randomly choosing a web page by the surfer, and $1-\phi$ is the probability of jumping off that page. Mihalcea and Tarau [18] use a heuristic value of 0.85 for ϕ, and we also use the same value for *TextRank* calculation. We initialize each of the terms in the graph with a default value of 0.25, and run an iterative version of the algorithm [7]. It should be noted that the initial value of a term does not affect its final score [18]. The computation iterates until the scores of the terms converge below a certain threshold or it reaches the maximum iteration limit (i.e., 100 as suggested by Blanco and Lioma [6]). As Mihalcea and Tarau [18] suggest, we use a heuristic threshold of 0.0001 for the convergence checking. *TextRank* applies the underlying mechanism of a recommendation (i.e., voting) system, where a term recommends (i.e., votes) another term if the second term complements the semantics of the first term in any way [18]. The algorithm captures recommendation for a term in terms of incoming links in the *text graph* (e.g., Fig. 1) from another terms both in local (i.e., same sentence) and global (i.e., entire document) context, and determines importance of the term. Once computation is over, each of the terms in the graph is found with a score which is considered as the weight or importance of that term in the texts.

E. Search Term Selection

Once *TextRank* is calculated, we rank the terms based on their weights (i.e., importance), and choose the search terms for a change task in a heuristic fashion. According to Kevic and Fritz [14], the terms that exist both in *Summary* and *Description* of a task are the most suitable for search terms. We adapt that idea given the fact that the overlapping terms in those fields might not be sufficient enough to form a search query. We thus first look for top-scored five terms in the *Summary* (i.e., title) of a change task. If the *Summary* is too small to provide all the terms, we collect the rest from the *Description* of the task. It should be noted that in both cases, terms are chosen based on their ranks or weights which are calculated by recursively analyzing the surrounding terms in the text graph. For instance, in the case of our example change task (Listing 1), the first four search terms– *Mac (score: 0.64), selection (score: 0.27), installs (score: 0.27)* and *improvement (score: 0.25)* come from the title. The remaining search term– *JREs (score: 1.00)*, one of the most important terms, is not contained in the title, and thus it comes from the description.

III. Experiment

In order to evaluate our proposed technique, we conduct experiments using the change tasks from two subject systems.

TABLE I
EXPERIMENTAL RESULTS

Metric	Log4j			eclipse.jdt.debug			Average
	T_3[1]	T_4[2]	T_5[3]	T_3	T_4	T_5	T_5
No. of Tasks Solved (NTS)	86(230)	98(230)	**111**(230)	48(119)	54(119)	**60**(119)	–
% of Tasks Solved (PTS)	37.39%	42.61%	**48.26**%	40.34%	45.38%	**50.42**%	49.34%
Mean Average Precision (MAP)	**66.54**%	62.82%	61.16%	**56.79**%	51.45%	53.16%	57.16%
Mean Recall (MR)	53.22%	**55.54**%	54.66%	**73.56**%	73.27%	72.00%	63.33%

[1]Results for three search terms, [2]Results for four search terms, [3]Results for five search terms

We also compare with one existing approach, and this section discusses our evaluation and validation details.

A. Dataset

In our experiments, we use 349 change tasks from two subject systems– `Log4j` by *Apache* and `eclipse.jdt.debug` by *Eclipse*. Each of those tasks were marked as *RESOLVED*, and we follow a careful approach for their selection. First we collect all the *RESOLVED* change tasks from the BugZilla repositories [1, 3], and then attempt to map them against the commit history of the corresponding projects (i.e., repositories) at GitHub [2, 4]. We analyze the commit messages of each project, and look for specific *Bug IDs* (i.e., identifiers of change tasks) in those messages. In GitHub, we note that each commit operation that solves a software bug or addresses a change request, generally mentions the corresponding *Bug ID* in the very first sentence of its commit message. We found such 230 commits in `Log4j` and 119 commits in `eclipse.jdt.debug`. We then collect the *changeset* (i.e., list of changed files) for each of those commit operations, and develop a solution set for the corresponding change tasks. Thus, for our experiments, we collect not only the actual change tasks from the reputed subject systems but also their solutions that were applied in practice by the developers. We use different utility commands such as *git, clone, rev-list* and *log* on *GitHub Bash* for collecting those information.

B. Search Engine

We use a *vector space model* based search engine– *Apache Lucene* [19] for searching the files that need to be changed for a change task. The search engine indexes the files in the corpus prior to search, and we note that the indexing by *Lucene Indexer* is not efficient. Especially the terms indexed from the source files are not meaningful and they often contain different special characters (i.e., punctuations). The indexer is basically targeted for simple text documents whereas the source files in the project contain items beyond regular texts (e.g., code segments). We thus apply limited preprocessing (i.e., stemming was avoided) on each of those source files in each project, and remove all punctuation characters from them. This transforms the source files into text like files, and the indexer becomes able to perform more effectively, especially in choosing meaningful index terms. Once a search is initiated using a query, the search engine filters irrelevant files in the corpus using a *boolean search model*, and then applies a *tf-idf* based scoring technique to return a ranked list of relevant documents. As existing studies suggest [5, 14, 17], we consider only the top ten results from the search engine for performance evaluation and validation of our technique.

C. Performance Metrics

Mean Average Precision at K (MAPK): *Precision at K* calculates *precision* at the occurrence of every relevant result in the ranked list. *Average Precision at K (APK)* averages the *precision at K* for all relevant results in the list for a search query. Thus *Mean Average Precision at K (MAPK)* is calculated from the mean of *average precision at K* for all queries in the dataset as follows:

$$APK = \frac{\sum_{k=1}^{D} P_k \times rel_k}{|S|}, \quad MAPK = \frac{\sum_{q \epsilon Q} APK(q)}{|Q|}$$

Here, rel_k denotes the relevance function of k^{th} result in the ranked list, P_k denotes the precision at k^{th} result, and D refers to number of total results. S is the solution set for a query, and Q is the set of all queries.

Mean Recall (MR): *Recall* denotes the fraction of the solution set that is retrieved for a search query. *Mean Recall* averages such measures for all queries in the dataset.

D. Experimental Results

We conduct experiments using 349 change tasks from two subject systems, and apply four different metrics–*no. of tasks solved*, *% of tasks solved*, *mean average precision* and *mean recall* for performance evaluation. We consider different sizes for the search query, and collect the performance details of our suggested queries for both subject systems which are reported in Table I. From the table, we note that our queries perform the best when five terms are used for search. For example, they return relevant results for 111 (48.26%) out of 230 change tasks from `Log4j`, and for 60 (50.42%) out of 119 change tasks from `eclipse.jdt.debug`, which is promising. Thus, on average, our queries retrieve 63.33% of the solutions from the dataset with a *mean average precision* of 57.16% for each of the subject systems. We also conduct search using six query terms; however, we note that our queries perform relatively poor in that regard for both subject systems.

We investigate whether the *preprocessing* (Section III-B) of corpus files significantly influences the performance of our queries. We re-ran the experiments on the corpus without preprocessing, and did not experience any major performance degradation. Thus, the findings actually demonstrate the robustness of our suggested search terms for a change task. We also investigate whether a list of randomly chosen five search terms from the *Summary* of change task is comparable to our suggested search terms given that our algorithm also emphasizes on *Summary* terms (Section II-E). We conducted

TABLE II
COMPARISON WITH AN EXISTING APPROACH

Technique	Metric	Log4j		eclipse.jdt.debug	
		T_3[1]	T_5[2]	T_3	T_5
Kevic and Fritz [14]	No. of Tasks Solved (NTS)	47(230)	**65**(230)	27(119)	**39**(119)
	% of Tasks Solved (PTS)	20.43%	28.26%	22.69%	32.77%
	Mean Average Precision (MAP)	54.08%	**56.90**%	50.61%	**54.92**%
	Mean Recall (MR)	**50.39**%	48.36%	66.70%	**78.53**%
Proposed	No. of Tasks Solved (NTS)	86(230)	**111**(230)	48(119)	**60**(119)
	% of Tasks Solved (PTS)	37.39%	48.26%	40.34%	50.42%
	Mean Average Precision (MAP)	**66.54**%	61.16%	**56.79**%	53.16%
	Mean Recall (MR)	53.22%	**54.66**%	**73.56**%	72.00%

[1]Results for three search terms, [2]Results for five search terms

experiments with such queries from both subject systems where we noted that the queries performed quite poorly. For example, the search engine returns relevant results for at most 54 (compared to 111) tasks from `Log4j` and 39 (compared to 60) tasks from `eclipse.jdt.debug`. Thus, the findings demonstrate that our proposed technique for search term suggestion is relatively more effective and more reliable.

E. Comparison with an Existing Approach

We compare with one of the latest and closely related state-of-the-art approaches– Kevic and Fritz [14] using our dataset (Section III-A). Kevic and Fritz [14] propose a heuristic model for search term suggestion for a change task where they consider frequency (i.e., *tf-idf*), location (i.e., *inSumAndBody, isInMiddle*) and notation (i.e., *isCamelCase*) of the terms from the task description. They suggest three search terms as a search query whereas we find our technique performing the best with five search terms in a query. We thus consider both sizes for the queries in our experiments. We implement their *relevance model* in our working environment, collect the search queries for the change tasks, and evaluate them using the same search engine (Section III-B). From Table II, we note that our queries are more effective compared to theirs for both subject systems. For example, their queries can retrieve relevant results for at most 65 (28.26%) out of 230 change tasks from `Log4j` and 39 (32.77%) out of 119 tasks from `eclipse.jdt.debug`. In terms of *precision* and *recall*, we note that our queries also perform relatively better than theirs.

IV. RELATED WORK

There exist a number of studies in the literature that attempt to support developers in *feature location* tasks with search queries. They adopt different lightweight heuristics [14] and query reformulation or expansion strategies [8, 12, 20], and perform different query quality analyses [9, 10, 11] and mining activities [13, 15]. However, most of these approaches expect a developer to provide the initial search query which they can improve upon, and their main focus is on improving a given query for a change task. On the other hand, in this study, we propose a novel technique that suggests a list of suitable terms as an initial search query for the given task. From technical perspective, we adapt an established algorithm– *TextRank* from information retrieval domain that analyzes the relative importance of the terms from the task description using a graph-based technique, and suggests the most important terms as search terms. Besides, we perform a case study using 349 change tasks that demonstrates the potential of the adapted technique.

V. CONCLUSION & FUTURE WORK

To summarize, in this paper, we propose a novel *TextRank*-based technique that automatically identifies and suggests search terms for software change tasks. Experiments with 349 change tasks from two subject systems show that our approach, on average, can return relevant results (i.e., Java classes) for 49.34% of the change tasks with 57.16% *mean average precision* and 63.33% *recall*. Comparison with one of the latest and closely related state-of-the-art approaches also shows that our approach performs comparatively better in different performance metrics. While these preliminary findings demonstrate the high potential of the proposed approach, further validation with more subject systems of diverse varieties and change tasks is warranted.

REFERENCES

[1] JDT, . URL https://bugs.eclipse.org/bugs/describecomponents.cgi?product=JDT.
[2] JDT Repository, . URL https://github.com/eclipse/eclipse.jdt.debug.
[3] . URL https://issues.apache.org/bugzilla/describecomponents.cgi?product=Log4j.
[4] Log4j Repository, . URL https://github.com/apache/log4j.
[5] 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.
[6] R. Blanco and C. Lioma. Graph-based Term Weighting for Information Retrieval. *Inf. Retr.*, 15(1):54–92, 2012.
[7] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.
[8] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.
[9] S. Haiduc. Automatically Detecting the Quality of the Query and its Implications in IR-based Concept Location. In *Proc. ASE*, pages 637–640, 2011.
[10] S. Haiduc and A. Marcus. On the effect of the query in ir-based concept location. In *Proc. ICPC*, pages 234–237, June 2011.
[11] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia. Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks. In *Proc. ICSE*, pages 1273–1276, 2012.
[12] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.
[13] M.J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In *Proc. MSR*, pages 377–386, 2013.
[14] K. Kevic and T. Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.
[15] K. Kevic and T. Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.
[16] A. Marcus and S. Haiduc. Text Retrieval Approaches for Concept Location in Source Code. In *Software Engineering*, volume 7171, pages 126–158. 2013.
[17] A. Marcus and J.I. Maletic. Recovering Documentation-to-Source-Code Traceability Links Using Latent Semantic Indexing. In *Proc. ICSE*, pages 125–135, 2003.
[18] R. Mihalcea and P. Tarau. Textrank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.
[19] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.
[20] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. ASOD*, pages 212–224, 2007.