A Dictionary to Translate Change Tasks to Source Code

Katja Kevic, Thomas Fritz
Department of Informatics
University of Zurich, Switzerland
{kevic, fritz}@ifi.uzh.ch

ABSTRACT

At the beginning of a change task, software developers spend a substantial amount of their time searching and navigating to locate relevant parts in the source code. Current approaches to support developers in this initial code search predominantly use information retrieval techniques that leverage the similarity between task descriptions and the identifiers of code elements to recommend relevant elements. However, the vocabulary or language used in source code often differs from the one used for describing change tasks, especially since the people developing the code are not the same as the ones reporting bugs or defining new features to be implemented. In our work, we investigate the creation of a dictionary that maps the different vocabularies using information from change sets and interaction histories stored with previously completed tasks. In an empirical analysis on four open source projects, our approach substantially improved upon the results of traditional information retrieval techniques for recommending relevant code elements.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]

General Terms

Experimentation

Keywords

Dictionary, location, change task, interaction history

1. INTRODUCTION

A lot of approaches to support and speed up developers in locating relevant places in the source code for change tasks are based on information retrieval (IR) techniques (e.g., [15, 18]). All of these approaches require a similarity in the terms used in change tasks and in the source code of a project [7]. While studies have shown that there is some textual overlap between the vocabulary used for change tasks and the one used for changed classes [16], these vocabularies can differ significantly and hamper the effectiveness of these approaches [11, 19]. This variability can stem from the difference in the people developing code and the ones creating change tasks, the often large number and fluctuation of people that work on a project, as well as the variability in terms people use to describe the same concept.

In our work, we consider the vocabulary used for the change tasks and the one used for the source code of a project as two different languages. Similar to Hindle et al. [13], we assume that source code is a *natural* language written by real people. A concept can then be expressed in the source code language (denoted as SCL) as well as in the natural language used in change tasks (denoted as NL), and the problem of locating relevant code for a change task can be seen as a translation problem of a concept from NL to SCL.

In this work, we investigate whether we can automatically create an effective dictionary from NL to SCL and use it to translate a change task into relevant source code elements. We analyze which mapping between NL and SCL is best to create the dictionary, either using change sets associated with change tasks or interaction histories captured as task contexts by Mylyn [4]. While change sets contain the actual code changes for a task, a task context captures a developer's interaction, such as selections and edits, with code elements in the IDE for a task. Finally, we investigate whether our dictionary approach is more effective in locating relevant code elements than a well-established IR technique.

In an empirical investigation on four open source projects, we found that a dictionary created from a combination of change set and task context information performs the best over all projects and outperforms the IR approach by 73% on class level and by 450% on method level. The results also indicate that a dictionary approach works better when there are more change tasks to create the dictionary from and that for projects with few change tasks, combining the dictionary with IR techniques would be beneficial.

This paper makes the following research contributions:

- A novel approach for concept location by translating change tasks to source code elements via a dictionary.
- An investigation of different NL-SCL mappings based on change sets and task context.
- An empirical analysis on 4 OS projects showing its effectiveness and improvement over traditional approaches.

This work represents an initial step towards an automatic translation of change tasks to source code with the potential to enhance existing concept location techniques.

2. RELATED WORK

Research related to concept location can be categorized into dynamic, static and hybrid approaches. An overview can be found in [7]. Dynamic approaches (e.g., [9, 21]) use

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597095

execution traces of a program to locate relevant places in the code, while static approaches leverage static information, such as change tasks or source code. Hybrid approaches (e.g., [5, 8]) combine dynamic and static approaches.

Static approaches that mine source code and change task repositories as well as interaction history are most closely related to our research. Many static approaches use information retrieval (IR) techniques (e.g., [18, 22, 15]) and differ mainly in their granularity of indexing units, preprocessing activities and similarity measurement definitions. Query reformulation approaches also use IR techniques and static information to help map change tasks in NL to SCL (e.g. [12]). Different to our dictionary approach, all of these IR based approaches require a similarity between the vocabularies used for change tasks and source code.

Other static approaches apply mining techniques to identify code elements that are relevant to the code a developer is working on, varying in the mined repositories, such as source code (e.g. [23]) or interaction histories (e.g. [6]). Our approach differs in trying to recommend relevant places in the code based on a change task written in natural language.

3. DICTIONARY APPROACH

In our approach, we consider the vocabulary used for describing change tasks and the one used for identifiers in the source code as two separate languages. Similar to a dictionary for two languages, such as English and German, our approach creates an approximate mapping between the terms of both languages, the natural language used for change tasks (NL) and the source code language (SCL). To look up relevant code elements for a change task, the dictionary can then be used to translate the terms in the change task to source code elements.

3.1 Automatic Creation of NL-SCL Dictionary

The dictionary to map NL to SCL is built by mining previously resolved change tasks from task repositories and the source code associated to these change tasks. We assume that the change tasks written in NL approximately map to the relevant code elements in SCL. In particular, we look at two sources of information for the mapping, change sets that are stored in a version control system and referenced by the change task, and task contexts gathered by Mylyn [4], if existent, and attached to the change task. While a change set only contains the actual changes for the change task, a task context also contains information on the elements that a developer navigated and selected for resolving the task and the interest shown by the developer for these elements.

To create the dictionary, our approach, first, goes through each change task and preprocesses the summary and description, by removing all punctuation and stop words as well as stemming the words, resulting in a list of distinct terms. For each term of a change task, it then creates or updates the mapping in the dictionary from the term to all code elements (unprocessed class and method identifiers in SCL) in the change set or task context of a change task. Each mapping between a term in NL and the terms in SCL has a weight that is one at first. Every new encounter and addition of the same mapping to the dictionary will then increment the weight by one to capture the relevance of the mapping.

3.2 Translation

To map a change task to relevant code elements, the task's summary and description is preprocessed by removing punctuation and stop words and applying stemming. For each term, not written in camelCase notation, the dictionary is used to retrieve the three terms in SCL for which the mapping has the highest weight, if existent. To identify the best translation of a change task into SCL, a weight is calculated for each NL term in the change task based on tf/idf [1] [17] and used as a multiplier to update the weights of the mapping. The retrieved SCL terms are then sorted based on the weight of the mapping and for duplicate SCL terms only the one with the highest weight is kept. For terms in camelCase notation, we assume that they are already in SCL and do not require a mapping [2]. Therefore, we only check if they exist in the SCL dictionary and then put them into the sorted list with a maximum weight. The result is a weighted list of source code identifiers that represents our top translation results of the change task.

4. EMPIRICAL ANALYSIS

We are interested in the following research questions:

RQ1 Can we create a dictionary that effectively maps natural language terms to source code elements on class and method level?

RQ2 Which source of information for the NL-SCL mapping is most valuable: change sets and/or task context?

RQ3 Does the dictionary improve upon well-established information retrieval approaches that assume a certain language overlap or similarity?

To answer these questions, we gathered information from four open source projects: Mylyn.Tasks, Mylyn.Context, Remus and AspectJ (see Table 1). We chose projects with respect to their availability of associated change sets and task contexts, their use in previous empirical analysis of similar approaches, their size and their problem domain. While Mylyn.Tasks (135 kLOC) and Mylyn.Context (37 kLOC) are projects in the domain of task management, AspectJ (553 kLOC) is a language extension to Java and Remus (123 kLOC) supports the management of different kinds of information. Both Mylyn.Tasks and Mylyn.Context are subprojects of the Mylyn open source project [4] and have many change tasks as well as associated change sets and task contexts available. We intentionally included two subprojects of Mylyn in our empirical analysis, to investigate whether the substantial differences in available change tasks and task contexts matter. AspectJ is a project that has been used in several previous analysis of source code recommendation approaches (e.g. [22]) and contains a lot of change tasks. Out of all change tasks available for AspectJ, only two have a task context associated and we therefore excluded AspectJ from the evaluation of our dictionary approach with task context. Finally, Remus is a project that is a lot smaller in its number of available change tasks, but also has task context available for several of them.

4.1 Data Collection

For each project we retrieved all available change tasks from a project's task repository (all projects use Bugzilla [2]). We mined commit comments in a project's change history

[1] idf is thereby calculated based on all past change tasks.

[2] We did not consider under_score notation since it was mainly used to reference different project versions rather than source code elements.

Table 1: Projects with number of all change tasks, of change tasks with change sets (CHS), and change tasks with CHS and task contexts (TC).

Project	# tasks	with CHS (dataset 1)	with TC (dataset 2)
Mylyn.Tasks	1029	396	280
Mylyn.Context	220	74	38
Remus	61	28	20
AspectJ	2737	951	-

to link change tasks by ID to change sets. If available, we also retrieved the task context captured using Mylyn [4, 14].

For our empirical analysis we created two datasets: dataset 1 includes all change tasks that have at least one change set associated with them; dataset 2 is a subset of dataset 1 and includes all change tasks that have at least one change set and one task context associated. For the AspectJ project we only created dataset 1, since only two tasks had a task context associated.

For each project we split each dataset into a training set to create the dictionary and a test set. For reasonable training and test sets and to be able to compare them across projects, we chose the test set size to be ten. Also, to be able to compare the results of the dictionaries using different information sources (change sets and/or task contexts) as well as preserving the chronological order, we chose the last ten change tasks in dataset 2 as the test set for Mylyn.Tasks, Mylyn.Context and Remus and the last ten change tasks in dataset 1 for AspectJ. All change tasks that were resolved before the ones in the test set were used as the training set. All training and test sets are available at [1].

To evaluate the different sources of information, we created three kinds of dictionaries: one with change sets using the training set of dataset 1; one with task contexts using the training set of dataset 2; and one with change sets and task contexts, where available, using the training set of dataset 1. To evaluate our approach on different granularity levels, we created dictionaries for the class and method level, using ChangeDistiller [10] for the change sets.

4.2 Data Analysis

For each change task in a test set, we used the corresponding dictionary to translate the change task into source code elements (either classes or methods). Since developers rarely scan through a very long list of recommendations [20], we used the top ten translation results to assess the relevance of the translation. We consider a recommended source code element as relevant if it is included in a change set associated with the change task, a technique commonly used in source code recommendation [7].

To compare our dictionary approach with established IR approaches, we used FLAT^3 [18], an Eclipse plug-in, that provides a suite of feature location techniques. Besides dynamic techniques, FLAT^3 offers a well-established technique using IR, employing Lucene [3], a state of the art text search engine. For each change task in a test set, we composed a query from the summary and description of the change task and ran the query in FLAT^3 to retrieve source code recommendations. FLAT^3 also removes stop words and stems terms in the query. Again, we solely considered the top ten recommendations and assessed their relevancy based on their existence in the associated change set.

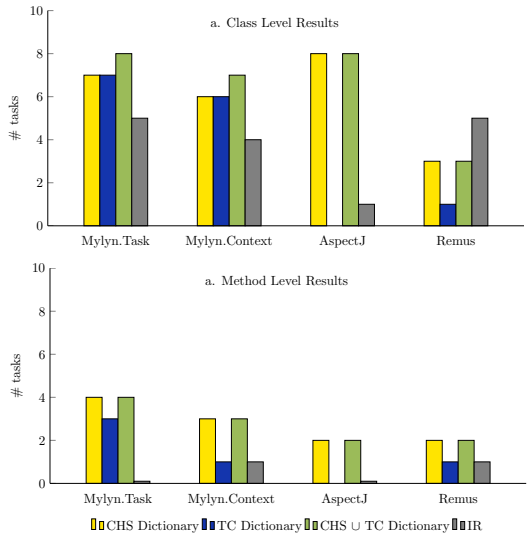

Figure 1: Results of Empirical Analysis.

4.3 Results and Discussion

The results of our analysis are presented in Figure 1. A dictionary created from change set information is able to map a change task to a relevant source code element on class level on average in 60% of the cases for all four projects. The results also indicate that the more data there is to train the dictionary at class level, the better the result of the translation (Pearson's correlation coefficient $r = .80$). For example, while AspectJ has 941 change tasks to train the dictionary and recommends relevant elements in 80% of the cases, Remus with only 18 change tasks to train has relevant recommendations in only 30% of the cases. On method level, the dictionary is only able to recommend relevant elements in 28% of the cases and the size of the training set has no influence (Pearson's correlation coefficient $r = -.18$). We hypothesize that one of the reasons for the low number might be that change tasks often require the creation of new methods, in which case the class recommendation might be correct, but the method to create could not have been in the dictionary yet. Overall for *RQ*1, our results show that if there is enough training data, the dictionary is able to effectively map change tasks to relevant source code elements at class level, but not necessarily at method level.

For *RQ*2 the results show that a dictionary trained with change set information is as good or better than a dictionary trained with task context. A dictionary trained on both, change sets as well as task contexts (CHS ∪ TC) outperforms the other two approaches and achieves an overall 65% of relevant class recommendations, and even 77% if excluding the Remus project that has less than 20 change tasks to train on. This indicates that while the change task to task context mapping does not contain more valuable information for creating a dictionary, it contains different and complementary information so that the combination provides the best dictionary for translation.

Comparing our best dictionary approach (CHS ∪ TC) with information retrieval (IR) in FLAT^3 shows that our approach improves upon IR by 73% over all four projects at class level (450% at method level). Only for the Remus project, the IR approach retrieved more relevant results than the dictionary at class level (5 versus 3 cases). We hypothesize that this again stems from the small number of training

data for creating the dictionary, since there were only 18 change tasks and their associated change sets and task contexts. For the AspectJ project, the IR approach performed particularly poor, also in comparison to ours (1 versus 8). We hypothesize that the project's age and the number of people that contributed and wrote change tasks for the AspectJ project in combination with the source code size lead to a higher disparity between the NL and SCL vocabulary, thus decreasing the effectiveness of the IR approach.

Overall, the results show that a dictionary created from change set and task context information is most effective in recommending relevant source code elements and can substantially outperform a well-established IR approach. Future studies are needed to confirm these preliminary results. The results also indicate that these approaches could be used complementary, favoring IR techniques while there is only little training data for a dictionary and favoring a dictionary once there is a reasonable amount. Since not all projects contain task context information or might not have enough change set information initially, this combination would be particularly beneficial. In future work, we plan on determining a good threshold for a reasonable training set.

Threats. A threat to external validity is that our preliminary evaluation only looks at ten change tasks in four projects. To mitigate the risk, we chose these projects to cover a variety of problem domains, sizes and availability of change task, change set and task context information. Furthermore, we queried only the latest revision of a project's source code with FLAT^3, which already included the changes of the change tasks to test. This may favor the results of the IR approach, as the specific vocabulary of added code pieces was already existent.

Since task context information is not always perfect due to developers starting or stopping to record their interaction too early or late for a task, dictionaries created from this information might be polluted with irrelevant code elements. We believe that with sufficient training data, these effects will be negligible and only hamper the presented results.

In our comparison, we used FLAT^3 as an IR approach. While BugLocator [22] is a more recent and, according to their results, a better approach using a revised Vector Space Model, we were not able to completely replicate their evaluation. Instead of the 286 change tasks mentioned, we could only map 150 change tasks to change sets, since we were not able to recover which revision of AspectJ was used in their evaluation. The dictionary created from these 150 mappings identified relevant classes in 69% of the cases, while Zhou et al. report relevant classes in 59.44% of 286 change tasks. However, since we were not able to completely reproduce their study, these results can only be used as an indication for a comparison.

5. CONCLUSION

In this paper we presented an approach that creates a dictionary out of task, source code and task context history to translate change tasks into relevant source code elements. In an empirical analysis of four open source projects, we found that our dictionary can effectively translate change tasks to relevant code elements at class level, and that the bigger the training set the better the results. These results indicate the potential in treating source code as a different language than the one used in change tasks. In future work we plan to extend our study and investigate the training set size needed for a "good" enough translation as well as looking into other information sources for creating the dictionary, such as comments in source code, which are mostly written in natural language.

6. REFERENCES

[1] `bitbucket.org/kkevic/datasets-nl-src-dictionary/downloads`.

[2] `bugs.eclipse.org/bugs/`.

[3] `lucene.apache.org/core/index.html`.

[4] `www.eclipse.org/mylyn/`.

[5] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *ICSM*, pages 357–366, 2005.

[6] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *IEEE Symp. Visual Languages and Human-Centric Computing*, pages 241–248, 2005.

[7] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Jour. of Softw.: Evolution and Process*, 25(1):53–95, 2013.

[8] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[9] W. Eric Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *Jour. of Sys. and Softw.*, 54(2):87–98, 2000.

[10] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[11] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Comm. ACM*, 30(11):964–971, Nov. 1987.

[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 *ICSE*, pages 842–851, 2013.

[13] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE*, pages 837–847, 2012.

[14] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *FSE*, pages 1–11, 2006.

[15] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223, 2004.

[16] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus. On the relationship between the vocabulary of bug reports and source code. In *ICMS*, pages 452–455, 2013.

[17] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[18] T. Savage, M. Revelle, and D. Poshyvanyk. Flat3: Feature location and textual tracing tool. In *ICSE*, pages 255–258, 2010.

[19] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *ICPC*, pages 123–132, 2008.

[20] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *ICSM*, pages 157–166, 2009.

[21] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proc. Conf. Softw. Maintenance*, pages 200–205, 1992.

[22] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.

[23] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.