Preprocessing Texts in Issue Tracking Systems to improve IR Techniques for Trace Creation

Mihaela Todorova Tomova, Michael Rath, Patrick Mäder
Software Engineering for Safety-Critical Systems
Technische Universität Ilmenau
{mihaela-todorova.tomova, michael.rath, patrick.maeder}@tu-ilmenau.de

Abstract

Multiple studies showed the usefulness of requirements traceability in developing software and systems. Still, a major challenge is to establish the required trace links among development artifacts. Often, information retrieval (IR) techniques combined with text similarity measures are used for this task. Applying these ideas to requirements texts found in issue tracking systems (ITS) of open source systems is difficult, because often these texts are structured and not only contain natural language. Thus, preprocessing of the textual information is required to extract the different kinds of text. In this paper, the authors study the structure of issue descriptions found in open source systems and identify several categories of text found therein, such as source code and stack traces. These text categories allow a more precise application of similarity analysis in order to create traces by comparing textual information of the same kind, i. e. source code with source code and natural language with natural language.

1 Introduction

Requirements traceability is broadly recognized as a critical element of any rigorous software development process [2]. Artifacts and the links among them build graph like structures, which can be efficiently queried to answer a variety of stakeholder questions about the system [5]. Furthermore, a high degree of traceability is beneficial in completing tasks in open-source systems [9]. However, a major challenge is to identify relevant traces among the development artifacts. The authors in [3] show, that often traces are created manually in late stages of the development process, which results in missing and incorrect links. Thus, automating this tedious manual task is highly desirable. One approach is to apply information retrieval (IR) techniques [6] and create traces based on textual similarity measures.

Issue tracking systems such as JIRA or Bugzilla are widely adopted to organize software development, especially in open-source systems. There, developers use issues to record features, improvements and defects of the systems to build. Additionally, they create traces among the issues as well as from issues to the source code [1]. However, a recent study specifically of open-source projects showed, that only up to 70% of all issues are linked to source code [7]. The textual description of the issues can be used to enrich this set of traces [4]. Nonetheless, a direct application of text similarity measures using the issue description might be limited, because often further information is stored in the text, such as source code or stack traces. In this paper, we propose a method for preprocessing issue descriptions to extract this information.

2 Analyzing Issue Descriptions

In JIRA, an issue contains a set of properties e. g. a *type*, a short *summary*, and a long *description*. The type categorizes issues e. g. into features, improvements, or bugs. Because of the amount of data, the description is used for similarity analysis by information retrieval techniques. However, the developers also use the description to store information in a structured way. Still being text, the type of information might be different from pure natural language. Figure 1 depicts this situation for improvement `PIG-1612`[1] of project Pig.

It contains two paragraphs (i. e. sections separated by a newline) of natural language followed by a code snippet written in the java programming language.

In a pre-study, we manually analyzed issue descriptions of seven open source systems contained in "The IlmSeven Dataset" [8] in order to quantify the distribution of structured information. The results are presented in Table 1. It shows, that 41% of all features and 40% of all improvements in project Groovy contain not only natural language. The statistics reveal, that structured information is quite often available, i. e. at least in every fifth studied issue, in issues descriptions. Thus, applying a preprocessing step to extract this data seems valuable.

During the issue analysis, we discovered three major types of information stored in descriptions: *text in natural language*, *java code*, and *stack traces*. Furthermore, we introduce an *unspecified* category for text, which does not fall in any of the other ones. E. g.

[1] `https://issues.apache.org/jira/browse/PIG-1612`

Pig / PIG-1612

error reporting: PigException needs to have a way to indicate that its message is appropriate for user

Details

Type:	Improvement	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	0.9.0
Component/s:	None		

Description

The error message printed to the user by pig is the message from the exception that is the 'root cause' from the chain of getCause() of exception that has been thrown. But often the 'root cause' exception does not have enough context that would make for a better error message. It should be possible for a PigException to indicate to the code that determines the error message that its getMessage() string should be used instead of that of the 'cause' exception.

The following code in LogUtils.java is used to determine the exception that is the 'root cause' -

```
public static PigException getPigException(Throwable top) {
    Throwable current = top;
    Throwable pigException = top;

    while (current != null && current.getCause() != null){
        current = current.getCause();
        if((current instanceof PigException) && (((PigException)current).getErrorCode() != 0)) {
            pigException = current;
        }
    }
    return (pigException instanceof PigException? (PigException)pigException : null);

}
```

Figure 1: Improvement `PIG-1612` for project Pig exemplifying structured information in the issue description, i. e. two paragraphs of natural language followed by a source code snippet.

Table 1: Percentage of issue descriptions containing structured information for features and improvements in seven open source projects.

	Issue Type	
Project	Feature [%]	Improvement [%]
Derby	-†	19
Drools	18	35
Groovy	**41**	**40**
Infinispan	18	16
Maven	24	24
Pig	23	23
Seam2	27	33

† Project Derby has no issues of type feature

improvement `Pig-5025`[2] contains a large stack trace and improvement `Pig-4747`[3] a Linux `rm` command, which both are categorized as unspecified. Figure 2 presents the method to extract these information categories from descriptions. Afterwards, IR could be applied purposefully by comparing information of the same nature, i. e. source code with source code and natural language with natural language.

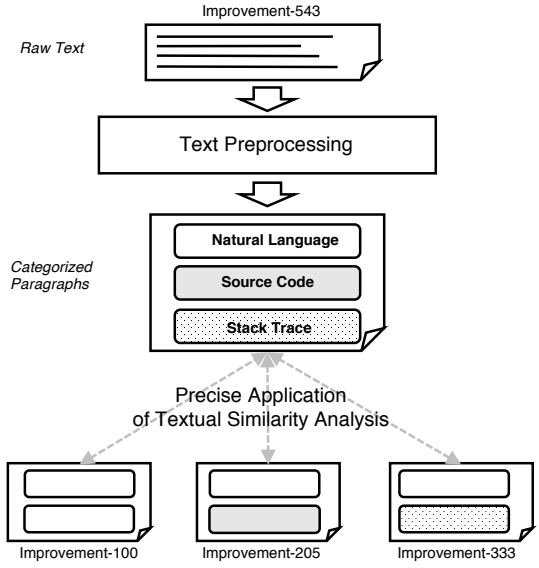

Figure 2: Categorization of paragraphs of issue description to allow precise application of textual similarity analysis in order to create traces.

3 Preprocessing Issue Descriptions

We propose the following method to process issue descriptions, shown in Figure 3. At the beginning, the issue description is divided into a list of paragraphs, which are identified by carriage returns. Afterwards, the category of each paragraph is identified individually.

In the first step, the algorithm checks, whether a paragraph is a stack trace ❶. Because of its specific format[4], this can easily be achieved by applying a regular expression. It there wasn't a match, the algorithm tries to identify the paragraph as java code ❷. Again, we use a set of regular expressions describing typical constructs such as class definitions (e. g. **public class** <ClassName> { /∗class body ∗/}), or methods (e. g. **public static int** <MethodName>

[2] `https://issues.apache.org/jira/browse/PIG-5025`
[3] `https://issues.apache.org/jira/browse/PIG-4747`
[4] `http://goo.gl/2X5yAJ`

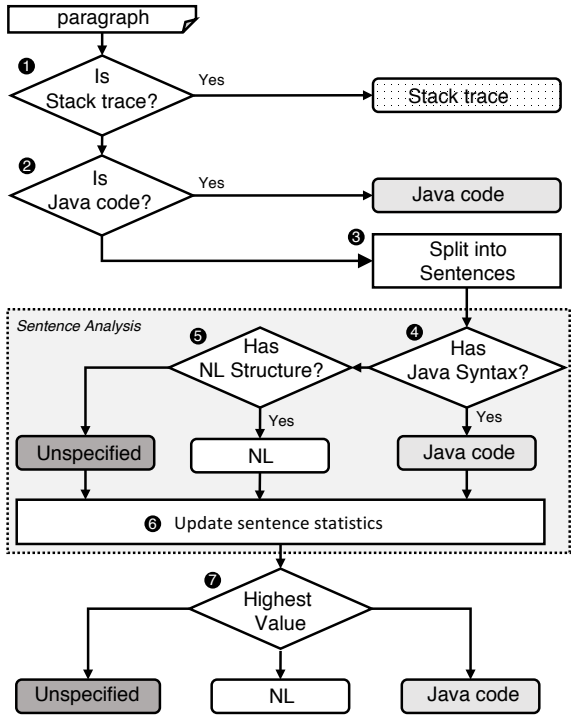

Figure 3: Proposed method to categorize paragraphs from issue descriptions in *stack trace*, *java code*, *natural language (NL)*, or *unspecified*.

(**int** var1, **int** var2) { /∗ method body ∗/}) in order to match source code. By considering that often in ITS partial code fragments are contained we want to check for both opening or closing braces or for at least the opening brace. If no java code is detected, we proceed with the next step and check for natural language. In this step we need to further examine the case in which texts may contain parts of source code and natural language. Here the proposed solution is to first split the text into multiple sentences ❸. For each sentence, we check if there is any kind of java syntax contained. Again, typical java constructs are considered, such as `camelCase` followed by equal sign and ending with semicolon, or `TitelCase` followed by variable name and also ending with semicolon, or loops constructs followed by **{}** or at least the **{**. These structures are quite uncommon in pure natural language texts, especially the usage of braces. Identifying these entities provides strong evidence against natural language and thus the sentence is marked as source code ❹. Otherwise, we search for evidence, that the sentence is written in natural language. Therefore, we inversely apply a technique widely used in text processing: stop-word removal. Stop words[5], such as *can*, *will*, *I* are common words found in natural language. Thus, we use the presence of such words in a sentence as indicator for natural language. Additionally, we search for patterns like I + <*verb*> ❺. If neither code nor natural language is detected, the sentence is marked as *unspecified*. We count the number of sentences marked as code, natural language and unspecified for each paragraph ❻. The overall category of the paragraph is derived using the highest sentence counter ❼.

[5] `http://goo.gl/Wb2jps`

4 Conclusion and Future Work

One of the most investigated means in literature to establish trace links is the application of information retrieval techniques. In this paper, we studied textual descriptions stored in issue tracking systems. Our research shows, that these descriptions are structured, whereas their content not only consists of natural language text but also of source code fragments, or stack traces. This is the case for about 25% of all features and improvements extracted from seven open source systems. We propose a method to extracts the different parts from the text. Used as a preprocessing step, this allows to purposefully apply text similarity and information retrieval techniques i. e. by comparing natural language with natural language, and source code with source code.

We plan to apply our method on multiple open source projects to study its effectiveness for trace creation using text similarity.

Acknowledgment

We are funded by the German Ministry of Education and Research (BMBF) grants: 01IS14026A, 01IS16003B, by DFG grant: MA 5030/3-1, and by the EU EFRE/Thüringer Aufbaubank (TAB) grant: 2015FE9033.

References

[1] A. Bachmann and A. Bernstein. Software process data quality and characteristics: A historical view on open and closed source projects. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 119–128, New York, NY, USA, 2009. ACM.

[2] Center of Excellence for Software Traceability (COEST). Software Traceability [Online]. `http://coest.org/bok/index.php/Main_Page`, 2015.

[3] D. Diaz, G. Bavota, A. Marcus, R. Oliveto, S. Takahashi, and A. D. Lucia. Using code ownership to improve ir-based traceability link recovery. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013*, 2013.

[4] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. On integrating orthogonal information retrieval methods to improve traceability recovery.

In *IEEE 27th International Conference on Software Maintenance, ICSM, 2011*, 2011.

[5] M. Goman, M. Rath, and P. Mäder. Lessons Learned from Analyzing Requirements Traceability using a Graph Database. *Softwaretechnik-Trends*, 37(3), 2017.

[6] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, New York, 2008.

[7] M. Rath, M. Goman, and P. Mäder. State of the Art of Traceability in Open-Source Projects. *Softwaretechnik-Trends*, 37(3), 2017.

[8] M. Rath, P. Rempel, and P. Mäder. The IlmSeven Dataset. In *Requirements Engineering Conference (RE), 2017 IEEE 25th International*. IEEE, 2017.

[9] P. Rempel and P. Mäder. Preventing defects: The impact of requirements traceability completeness on software quality. *IEEE Trans. Software Eng.*, 43(8):777–797, 2017.