Natural Language Processing based Detection of Duplicate Defect Patterns

Qian Wu, Qianxiang Wang

School of Electronics Engineering and Computer Science, Peking University

Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education

Beijing, China, 100871

{ wuqian08, wqx}@sei.pku.edu.cn

ABSTRACT

A Defect pattern repository collects different kinds of defect patterns, which are general descriptions of the characteristics of commonly occurring software code defects. Defect patterns can be widely used by programmers, static defect analysis tools, and even runtime verification. Following the idea of web 2.0, defect pattern repositories allow these users to submit defect patterns they found. However, submission of duplicate patterns would lead to a redundancy in the repository. This paper introduces an approach to suggest potential duplicates based on natural language processing. Our approach first computes field similarities based on Vector Space Model, and then employs Information Entropy to determine the field importance, and next combines the field similarities to form the final defect pattern similarity. Two strategies are introduced to make our approach adaptive to special situations. Finally, groups of duplicates are obtained by adopting Hierarchical Clustering. Evaluation indicates that our approach could detect most of the actual duplicates (72% in our experiment) in the repository.

Keywords

Duplicate, Defect Pattern, Natural Language Processing, Information Retrieval

1. Introduction

Static defect analysis tools, such as FindBugs [4], PMD [5], and Jlint [1], are widely used to detect and report software defects by analyzing source code or bytecode against pre-defined defect patterns. One piece of the most essential work of developing such tools is the collection of defect patterns, which generally describe the common characteristics of certain types of software code defects. However, knowing what patterns to check poses a major challenge for the development of such tools, because these patterns are often undocumented or specified in an ad hoc manner. [2]

To solve this problem, various approaches have been proposed to infer these patterns by mining software repositories [8], program traces [9, 10] or API documentations [11]. While these approaches endeavor to automate defect pattern collection, they suffer from high false positive rates and tremendous human efforts are indispensible to examine these patterns generated automatically.

Meanwhile, some open defect pattern repositories such as CWE [13] and CDP [14] are built, to collect different kinds of defect patterns, and to support the development of static defect analysis tools. In these repositories, authorized users are allowed to submit defect patterns, and then each pattern is assigned to a developer of static analysis tools who will work to make the tool capable of detecting the corresponding type of defects. This process is demonstrated in Figure 1. By learning from emerging defects, programmers could extract defect patterns, which will support the enhancement of static defect analysis tools. These enhanced tools will in turn further improve the software quality.

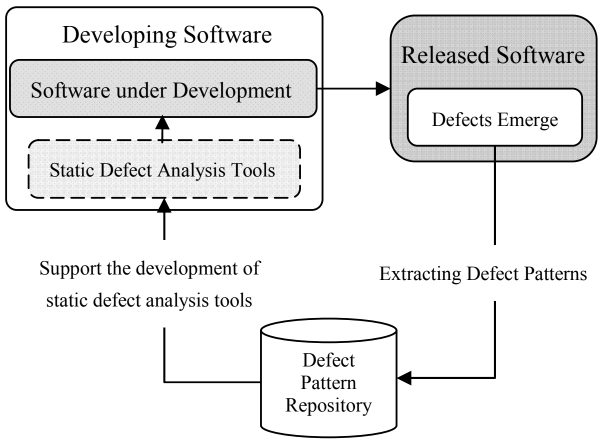
Developing Software
Software under Development
Static Defect Analysis Tools
Released Software
Defects Emerge
Support the development of static defect analysis tools
Extracting Defect Patterns
Defect Pattern Repository

Fig 1. The role of defect patterns in Software Life Cycle

However, when a user enters a defect pattern, without the exact knowledge of what already exists in the repository, she/he may submit duplicates. In addition to the unnecessary work load caused by duplicates, when several developers of a static defect analysis tool work on the same patterns, it may lead to inconsistencies, and also the tool performance is ruined because the tool is actually checking the program against the same pattern more than once.

To reduce the duplicate defect patterns, users are often prompted to make a series of searches before submission to ensure no similar ones exist and meanwhile triagers are employed, who examine newly added entries and eliminate incorrect and duplicate ones. To indentify duplicates, triagers may either rely on their knowledge of what exists in the repository or they must perform a series of searches to find potential duplicates. Both approaches involve a cost of time and energy, and further, the manual nature of these techniques also tends to result in missed identifications of possible duplicates due to faulty memory or inadequate searching. [15]

In this paper, we proposed an effective approach to detect potential duplicate defect patterns using natural language processing, based on our previous work [3]. In our approach, defect patterns are first represented by Vector Space Model, and then these patterns are clustered automatically based on their textual information, finally, clusters of similar patterns are suggested to either the users who is currently entering a pattern or the triagers.

The most related existing research lies in the field of duplicate defect detection [15, 18, 22], which aims for finding duplicate defect reports in defect repositories, such as Bugzilla [23], JIRA [24]. Defect pattern repositories are different from defect repositories, mainly in that, defect reports in the latter are specific to a certain project, thus the execution traces or the defect occurring contexts are available and can be exploited by the duplicate detection; however, because defect patterns are independent of any specific project, the duplicate detection becomes even more difficult with the relatively less and imprecise information.

This paper is outlined as follows. Section 2 demonstrates our research on how a defect pattern should be composed; Section 3 describes our approach in detail and Section 4 presents the experiment and the experience of applying the approach and finally concludes.

2. Defect Pattern Composition

This section discusses: in what way can a defect pattern be expressed both clearly with enough information, and what's more important, making it easy to detect duplicates.

Defect patterns generally describe the common characteristics of certain types of software coding defects, indicating the recurring correlations between signaled errors and underlying bugs in a program. [7] Each defect pattern may correspond to several defect instances in actual projects, and all of the instances embody the characteristics as described in the pattern. According to whether the instances of one pattern is incurred when using a library, defect patterns are divided into two categories: *"Library Independent"* and *"Library Specific"* [16]; the former type of defect patterns are incurred independently of the use of software libraries, whereas patterns of the latter type could only be incurred when applying a library, typically pertaining to a class (or method/field) in the library. E.g., a defect pattern of *"Library Specific"* may describe "The return value of String.replace() is ignored, thinking it will update the object on which the method is invoked.", which is related to "java.lang.String".

In order to fulfill the preceding goals, there should be several considerations:

- **The more defect patterns involved in comparison, the harder to indentify duplicates.** One way to lower the number of patterns involved in comparison is by classification, and there are two candidate dimensions: by "platform" and by "content type". For example, the platform of the defect pattern "Nesting transactions in Hibernate would result in transaction exceptions" is "Hibernate"; and the content type of "Comparison of String parameter using = = or! =" could be "String".
- **Distinguish between semantically different parts of a defect pattern.** NLP (Natural Language Processing) based on vector space model mainly exploits the appearance of textual information without understanding the semantic meaning. Thus, when mixing the symptoms of the defect with its consequences, NLP could not extract and differentiate them reasonably, resulting in confusion and information lost. In addition, random arrangement of contents would interfere with duplicate detection. E.g. suppose two patterns have identical textual descriptions, but one of them includes example code in the description while the other doesn't, as a result, the presence of code will compromise their similarity.
- **Highlight the key attributes of defect patterns.** As in Topic Detection, the extraction of "Date" and "Location" is helpful of new event identification [15]; obtaining the subject the defect pattern is about can assist in distinguishing different patterns apart. For *"Library Specific"* patterns, the subjects are the correlating classes, methods or fields. For *"Library Independent"* patterns, we use "keywords" instead to represent the subject. E.g., the keyword of defect pattern "cleaning up in catch block" is "catch", and "misuse of ++" has the keyword "++".
- **To simplify and standardize user input.** We divide "severity" into several levels from which user could simply choose one when entering a defect pattern. The levels are: "Error", "Fragile", "Security vulnerability", "Suspicious", "Performance degradation", "Dead Code" and "Bad Style".

From the above considerations, one defect pattern should comprise the following fields: "summary", "description", "platform", "content type", "correlating classes, methods, fields", "keywords", "severity", "Bad Example", "Correct Example". A typical defect pattern is presented below.

Example 1

Summary: Ignorance of the return value of String.replace().

Description: The return value of String.replace() is ignored, thinking it will update the object on which the method is invoked.

Platform: J2SE

Content type: Method Misuse

Correlating: java.lang.String.replace()

Keywords: return value, String, replace

Severity: Error

Bad Example:

```
String aString = "test";
aString .replace('s', 'x');
if("text".equals(aString)){…}
```

Correct Example:

```
String aString = "test";
aString = aString .replace('s', 'x');
if("text".equals(aString)){…}
```

A Typical Defect Pattern of *"Library Specific"*

3. Approach Details

Given one defect pattern, the main idea to detecting its similar ones is to do clustering first, and then all patterns within the same cluster of the given one are considered as potential duplicates.

To cluster defect patterns, our approach processes defect patterns in the repository according to these steps:

1. For each pattern, extract the natural language fields and represent each field as a document vector. (Section 3.1)
2. Compute similarities for two patterns.(Section 3.2)
3. Perform Data Clustering based on the pre-computed similarities. (Section 3.3)

3.1 Defect Pattern Representation

Among the many fields of a defect pattern, all are expressed in natural language except the example codes, "platform", and "content type". Our approach first extracts all the natural language fields and then each field goes through the three processes of tokenization, stemming, and stop words removal to arrive at a bag of words which could be used later. [18] After tokenization, sentences and paragraphs are turned into streams of words. Stemming aims at identifying the ground form of each word. Then by removing stop words, words that do not carry any specific information and hence are not likely to be of any help in the similarity analysis are eliminated. Despite the common stop word list [19], all words from the defect pattern templates are also excluded. To enhance the performance of NLP, a thesaurus is also applied, so that synonyms are considered identical in our approach.

After the preceding three steps, each field of a defect pattern is turned to a bunch of words; next our approach represents each field using Vector Space Model (VSM) [17], which is a widely adopted technique in NLP. In VSM, each document is converted to an *n*-dimensional vector $<w_1,w_2,…,w_n>$, where *n* is the number of unique index terms appearing in all the documents and w_i ($1 \le i \le n$) is the weight of the *i*-th index term and defined by Formula (1).

$$w_i = tf_i \times idf_i \qquad (1)$$

In Formula (1), tf_i refers to the term frequency, which counts the number of occurrences of the *i*-th index term in the document, and idf_i refers to the Inverse Document Frequency [22] defined by Formula (2).

$$idf_i = log\frac{Dsum}{Dw_i} \qquad (2)$$

In Formula (2), *Dsum* is the total number of documents, and Dw_i is the number of documents that contains the *i*-th index term. After converting documents to vectors, the similarity of two documents can be calculated using the cosine similarity [17] of the two vectors as defined below.

$$cos < D_1, D_2 >= \frac{\sum_{i=1}^{n} w_{1i} \times w_{2i}}{\sqrt{\sum_{i=1}^{n} w_{1i}^2 \times \sum_{i=1}^{n} w_{2i}^2}} \qquad (3)$$

So far, by regarding each field as a document when adopting VSM, each defect pattern is now transformed into several vectors; and by calculating the cosine similarities we obtain the similarities between each two corresponding fields of each pair of defect patterns.

3.2 Field Importance Calculation

To obtain the similarity between two defect patterns, we need to combine the similarities of each two corresponding fields. The most straightforward way is to sum the similarities of each field weighted by the field's importance, defined by Formula (4).

$$Sim = \sum(Sim_i \times W_i) \qquad (4)$$

$$i \in (summary, desc, keywords, correlating, severity)$$

In Formula (4), *Sim* denotes the similarity of two defect patterns, *i* denotes the indexes of different fields of defect patterns, Sim_i denotes the similarities of the *i*-th field, and W_i denotes the importance of the *i*-th field.

So far, Sim_i could be calculated by our approach according to Section 3.1, to complete the similarity calculation, we need a metric to measure the importance of each field. Different from our previous work [3], which merely draws a magnitude relationship of these fields, we present a novel approach based on information theory (or entropy) [21] to determine the importance values quantitatively.

Intuitively, the more diverse the contents of a certain field are, the more powerful this field is in differentiating different defect patterns, thus the more important this field is. Therefore, it is reasonable to weigh each field by the degree of its content diversity. The field importance is defined by Formula (5).

$$W_i = \frac{\sum_{j=1}^{N}(1 - H(a_{ij}))}{N} = 1 - \frac{\sum_{j=1}^{N} H(a_{ij})}{N} \qquad (5)$$

In Formula (5), *N* is the number of words appearing in all the instances of the *i*-th field. a_{ij} is a word appearing in one instance of the *i*-th field, and $H(a_{ij})$ is the information entropy of a_{ij} within the context of the *i*-th field, defined by Formula (6); "$1- H(a_{ij})$" actually indicates the amount of information carried *by* a_{ij} (the reason is explained later). In this formula, by averaging the amount of information carried by each word appearing in the *i*-th field, our approach obtains the relative importance of this field. The information entropy of a_{ij} is defined by Formula (6).

$$H(a_{ij}) = -\sum_{k=1}^{m}(p_{jk} \, log_m \, p_{jk}) \qquad (6)$$

In Formula (6), m is the number of instances of the *i*-th field, and p_{jk} is the probability that a_{ij} appears in the *k*-th instance of the *i*-th field. Note that, the entropy, *H*, of a variable a_{ij} is a measure of the degree of uncertainty associated with the value of a_{ij}. The more uniformly a_{ij} is distributed among different instances, the more uncertain it becomes, whereas the less information it carries. In extreme cases, if a word a_{ij} appears in all instances of a certain field, the p_{jk} will be *1/m* constantly, and thus it gets the maximal entropy, 1; nevertheless this word actually carries no useful information, because it couldn't help to distinguish between different instances.

We now present an example to illustrate the calculation of field importance.

Example 2

Suppose our repository contains only three defect patterns, whose summaries are displayed below, and we want to compute the importance of the field "summary".

Summary 1 equals() used to compare incompatible arrays.

Summary 2 Call to equals() comparing different types.

Summary 3 equals() always returns false.

After the three word processing steps described in Section 3.1, the summaries become

Summary 1 equal, use, compare, incompatible, array

Summary 2 call, equal, compare, different, type

Summary 3 equal, always, return, false

The entropies for each word are

$$\mathrm{H(equal)} = -3 \times ((1/3) \times \log_3 (1/3)) = 1$$

$$\mathrm{H(compare)} = -(2 \times (1/2) \times \log_3(1/2) + 0) = \log_3 2$$

$$\mathrm{H(others)} = -(0 + 0 + 1 \times \log_3 1) = 0$$

According to Formula (5), the weight of field *summary* is

$$\mathrm{W_{summary}} = (1/14) \times (3 \times (1-1) + 2 \times (1 - \log_3 2) + 9 \times (1-0)) \approx 0.696$$

After calculating the importance values for each field, each value needs to be normalized by their summation, as described in Formula (7), so that the final similarity of two defect patterns stays between 0 and 1.

$$W_i = \frac{W_i}{\sum W_j} \qquad (7)$$

$$i, j \in (summary, desc, keywords\ , correlating, severity)$$

After setting up the initial values of the weights, we adopt two strategies to make our approach adaptive to some special situations.

Strategy 1 When a defect pattern involved in comparison has some blank field, the corresponding field similarity is deemed invalid (denoted by *NaN*), at the same time, our approach increases the weight of the other fields except for *Severity*. After the adjustment, the summation of all field weights remains 1.

This strategy is adopted to make our approach adaptive to defect patterns with incomplete information. Various reasons can lead to the incompleteness, e.g., *"Library Independent"* defect patterns don't have "correlating classes, methods, fields", and defect patterns which are currently being entered have unfinished fields.

The formal description of the adjustment is as follows

$$if\ Sim_i = NaN, then\ W_i = 0, W_j = W_j' \times \beta \qquad (8)$$

$$i, j \in (summary, desc, keywords\ , correlating) \wedge (i \neq j)$$

In Formula (8), W_i denotes the weight of the blank field, and W_j' and W_j denote the weights of the other fields before and after the adjustment respectively. β is the boosting factor which indicates how many times the weight is enlarged, and its value can be calculated by Formula (9), where $W_{severity}$ represents the weight of *severity*, which stays the same in the adjustment.

$$\beta = \frac{1 - W_{severity}}{1 - W_{severity} - W_i} \qquad (9)$$

Strategy 2 When some field similarity is high sufficiently, our approach enlarges its field weight. After the adjustment, the summation of all field weights remains 1.

The rationale is that among the three fields, the "summary", "description", and "keywords", each one could be capable of demonstrate the content of a defect pattern independently;

meanwhile multiple ways exist to express the same meaning in natural language. Therefore, if the summaries or descriptions or keywords of two patterns are highly similar, this pair is probably potential duplicates, even if the other fields vary. In this sense, increasing the weight of fields which happen to be highly identical is instrumental to identify semantically similar defect patterns with different appearances. The excerpt of two defect patterns below is an example.

Example 3

Summary 1: Ignorance of the return value of String.replace().

Description 1: The return value of String.replace() is ignored, thinking it will update the object on which the method is invoked.

Summary 2: Ignorance of the return value of String.replace().

Description 2: The effect of this method takes place by returning a new String, instead of modifying the parameter.

The summary similarity of the two patterns is 1, however, their descriptions vary a lot, although in fact the two patterns are semantically identical. Thus, by enlarging the summary weight, our approach successfully avoids a false negative.

The formal description of the adjustment is as follows

$$if\ Sim_i > max_i, i \in (summary, desc, keywords)$$

$$then\ W_i = 1 - \frac{1 - W_i'}{\mu}, W_j = \frac{W_j'}{\mu} \qquad (10)$$

$$j \in (summary, desc, keywords, correlating, severity)$$

$$\wedge\ (i \neq j), \mu > 1$$

In Formula (10), i denotes the index of the field with sufficiently high similarity, j denotes the other fields, max_i denotes the threshold of determining whether the i-th field is identical, W_i' and W_i denote the weight of the *i-th* field before and after adjustment respectively. μ denotes how much the weight of the j-th field shrinks.

3.3 Data Clustering

Ideally, duplicate defect patterns are the ones with the highest similarities. Therefore, our approach makes suggestion based on the result of clustering, whose goal is to group similar defect patterns together, on the premise that potential duplicates must exist in the same cluster.

To perform data clustering, our approach adopted Agglomerative Hierarchical Clustering (AHC) [6], which is a commonly used clustering method. An AHC procedure produces a series of partitions of the data, each step joins together the two clusters with the highest similarities. To reduce computation complexity, we suppose only patterns with the same "platform" and "content type" have the possibility to be grouped together.

After clustering, defect patterns that are most similar to each other are grouped into clusters. Thus, given a defect pattern, our approach suggests the ones residing in the same cluster as the potential duplicates.

4. Experiments

According to our approach described in Section 3, we implemented a prototype tool of detecting duplicates and embedded it into Defect Pattern Repository for Java [14]. The prototype tool could be applied in two ways, either during a new pattern is being entered, or in the process of examining patterns in the repository. In the former situation, the tool recommends similar patterns to the user who is currently typing a defect pattern, based on the information already fulfilled, so as to help the user to decide whether to continue the submission. In the latter one, the triager could use this tool as an assistant when trying to identify duplicates.

In our experiment, 289 defect patterns were identified as duplicates by triagers among 3709 defect patterns in the repository. We ran the prototype tool against these 3709 patterns and it could report 72% of the actual duplicates. A user who submits a new pattern has to wait for an average of 180ms to receive a list of suggested duplicates (with the processor of Intel Pentium M, 1.73GHz, and a memory of 768M).

However, the limitation of our approach lies in its incapability of discovering semantically identical patterns with unlike appearance. E.g., defect pattern "The modulo is 1 in remainder operation" and "The operator % is followed by 1" are actually duplicates; however the similarity calculated by our approach is too low to stand out.

5. Conclusion and Future Work

This paper first designs the composition of a defect pattern, so that it could be expressed clearly and also makes it easy to detect duplicates. Then we present our approach of detecting duplicates based on natural language processing. Through our experience of applying the prototype tool, our approach is proved to be capable of detecting potential duplicates with acceptable recall and precision rate, however there still exists room of improvement.

Future work could be conducted in two directions. First, the vector space model needs to be enhanced. By combining semantic and syntactic analysis, our approach is hopeful to discover identical defect patterns with unlike appearance. The other direction is to exploit the example code analysis. The application of code clone technique [20] in duplicate defect pattern detection could be of interest to research.

Acknowledgement

Our work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703 and the Science Fund for Creative Research Groups of China under Grant No. 60821003.

References

[1] Jlint, available at http://Jlint.sourceforge.net/.

[2] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. Proceedings of the eighteenth ACM symposium on Operating systems principles, pages 57-72, 2001.

[3] Q. Wu, Q. Wang. An Approach to Detect Duplicate Defect Patterns based on Information Retrieval (In Chinese). Proceedings of the 7th National Software and Application Conference in China, 2008.

[4] D. Hovemeyer and W. Pugh. Finding bugs is easy. Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 132-136, 2004.

[5] T. Kremenek, K. Ashcraft, J. Yang and D. Engler. Correlation exploitation in error ranking. Proceedings of the 12th international symposium on Foundations of software engineering, pages 83-93, 2004.

[6] Hastie, Trevor/ Tibshirani, Robert/ Friedman, Jerome H.. The Elements of Statistical Learning (2nd ed.). Springer, pages 520-528, 2009.

[7] L. Yu, J. Zhou, Y. Yi, P. Li, Q. Wang. Ontology Model-based Static Analysis on Java Programs. Proceedings of the 32nd IEEE International Computer Software and Applications Conference, pages 92-99, 2008.

[8] B. Livshits, T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. Proceedings of the 10th European software engineering conference, pages 296-305, 2005.

[9] Z. Li, Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. Proceedings of the 10th European software engineering conference, pages 306-315, 2005.

[10] C. Goues, W. Weimer. Specification Mining with Few False Positives. Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2009.

[11] H. Zhong, L. Zhang, T. Xie, H. Mei. Inferring Resource Specifications from Natural Language API Documentation. Proceedings of the 24th International Conference on Automated Software Engineering, pages 307-318, 2009.

[12] G. Ammons, D. Mandelin, R. Bodík, J. R. Larus. Debugging temporal specifications with concept analysis. Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 182-195, 2003.

[13] Common Weakness Enumeration, http://cwe.mitre.org/.

[14] Code Defect Pattern Repository for Java, http://cdp.seforge.org/.

[15] Hiew L. Assisted Detection of Duplicate Bug Reports. Master's thesis, University of British Columbia, 2006.

[16] N. Meng, Q. Wang, Q. Wu, H. Mei. An Approach to Merge Results of Multiple Static Analysis Tools. Proceedings of the 8th International Conference on Quality Software, 2008.

[17] G. Salton, M. J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[18] P. Runeson, M. Alexanderson, O. Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. Proceedings of the 29th international conference on Software Engineering, pages 499-510, 2007.

[19] C. Buckley, C. Cardie, S. Mardis, M. Mitra, D. Pierce, K. Wagsta, J. Walz. The smart/empire tipster ir system. Proceedings of TIPSTER Phase III, pages 107-121, 1999.

[20] D. M. German, M. D. Penta, Y. Gueheneuc, G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. Proceedings of the International Working Conference in Mining Software Repositories, pages 81-90, 2009.

[21] Y. Lan, L. Bing, and L. Xiaoli. Eliminating noisy information in Web pages for data mining. Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 296-305, 2003.

[22] X. Wang, L. Zhang, T. Xie. An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information. Proceedings of the 30th international conference on Software engineering, pages 461-470, 2008.

[23] Bugzilla, available at http://www.bugzilla.org/.

[24] JIRA, available at http://www.atlassian.com/software/jira/.