ISSN 0361-7688, Programming and Computer Software, 2016, Vol. 42, No. 4, pp. 216–224. © Pleiades Publishing, Ltd., 2016.
Original Russian Text © D.V. Luciv, D.V. Koznov, H.A. Basit, A.N. Terekhov, 2016, published in Programmirovanie, 2016, Vol. 42, No. 4.

On Fuzzy Repetitions Detection in Documentation Reuse

D. V. Luciv[a], D. V. Koznov[a], H. A. Basit[b], and A. N. Terekhov[a]

[a] *St. Petersburg State University, Universitetskaya nab. 7/9, St. Petersburg, 199034 Russia*
[b] *Lahore University of Management Sciences, Opposite Sector U, DHA, Lahore, 54792 Pakistan*
e-mail: d.lutsiv@spbu.ru, d.koznov@spbu.ru, a.terekhov@spbu.ru, hamidb@lums.edu.pk

Received February 12, 2016

Abstract—Increasing complexity of software documentation calls for additional requirements of document maintenance. Documentation reuse can make a considerable contribution to solve this problem. This paper presents a method for fuzzy repetitions search in software documentation that is based on software clone detection. The search results are used for document refactoring. This paper also presents Documentation Refactoring Toolkit implementing the proposed method and integrated with the DocLine project. The proposed approach is evaluated on documentation packages for a number of open-source projects: Linux Kernel, Zend Framework, Subversion, and DocBook.

DOI: 10.1134/S0361768816040046

1. INTRODUCTION

Documentation is an important integral part of modern software. There are two types of documentation: technical documentation (requirements, project specifications, test plans and reports, API documentation, and so on [1]) and user documentation. Technical documentation helps software developers to read and understand software products, as well as to facilitate their development and modification [2]. And yet, technical documentation can be of considerable size and complexity. Like software itself, software documentation is steadily improved in the process of software development and maintenance. Improving quality of technical documentation is a well-known problem that has not yet been solved [3, 4].

One of the reasons for the poor quality of documentation is a large number of uncontrolled repetitions. This makes document maintenance more complicated, since, to keep consistency, one has to update a number of parts of documents that contain replicated data. This is often neglected due to a lack of time and technical facilities. As a result, the number of errors in documentation gradually increases. The situation seems to be even more difficult because of near text duplicates: the same software features can be described from different points of view with different details. Moreover, software documentation contains descriptions of similar objects (functions, interrupts, classes, and messages); thus, the corresponding text fragments involve both similarities and differences. It should also be noted that uniform documentation is a good form, where the description of similar objects has a common structure and matches in the text parts that describe the same data.

There are various software reuse techniques [5, 6]. The DocLine environment [9], which is a basis of the research presented in this paper, uses Bassett and Jarzabek's adaptive reuse technique [7, 8], which is employed in the DocLine documentation development technology. This technology supports XML documentation refactoring (changing the XML representation of a document, while preserving its appearance) [10]. However, a certain problem arises concerning automatic search for repeating text fragments. In [11, 12], a software clone detection approach based on the Clone Miner tool [13] was proposed for finding textual repetitions. Yet, only crisp repetitions were considered.

This work continues the investigation [9] by addressing the problem of finding fuzzy repetitions that are composed of crisp repetitions if corresponding clone groups are positioned close to one another in text. The developed algorithm is implemented as a part of the Document Refactoring Toolkit [14] and is evaluated on documentation packages for some well-known open-source projects: Linux Kernel [15], Zend Framework [16], Subversion [17], and DocBook [18].

2. OVERVIEW

An overview of the field of technical documentation development (problems and solutions) can be found in [19]. Below, we confine ourselves to some works devoted to automatic analysis and processing of documents.

In [20], an approach to automatic generation of software resource specifications based on API documentation is proposed. This approach addresses the following problem: developers often misread software documentation and, therefore, make lots of mistakes when developing software resources, although resource description is fully contained in the documentation and can be used as input data for automatic generation.

In [21], an approach is proposed to finding errors in software documentation by comparing program code entities described in the documentation (data types, procedures, variables, classes, etc.) with code examples also contained in it.

In [2], it is proposed to estimate the quality of project documentation on the basis of expert polls.

In [22, 23], some metrics for documentation quality evaluation are proposed. Authors adapted the VizzAnalyzer tool designed for software clone detection [24] to finding repetitions in technical documentation.

XML languages are widely used to develop technical documentation. The best known XML languages are DocBook [25] and DITA [26], which support the modular approach and allow constructing reusable text modules. However, these languages weakly support documentation reuse. In the DocLine technology [12], the Bassett–Jarzabek adaptive reuse technique [7, 8] is applied to documentation; this technology supports reuse parameterization and planning. It should be noted, however, that all these approaches assume that software documentation is originally developed as a set of reusable modules and provide no means for repetitions detection.

Thus, we can draw the following conclusion. Repetitions detection in technical documentation is used only in [22, 23, 27, 28]. Yet, in [22, 23], repetitions found are used only for determining the quality of documentation, but not for document transformation. In [28, 29], repetitions are used for automatic refactoring of documents according to the refactoring method described in [10]. And still, in all these works, only crisp repetitions are considered.

3. TECHNOLOGIES IN USE AND THE DOCLINE PROJECT

3.1. DocBook

DocBook [25] is a set of standards and tools for technical documentation development in XML.[1] DocBook separates text representation (boldface font, italic font, alignment, etc.) from content of documents, thus implementing the idea of single source: the same document can be represented in HTML, PDF, etc. DocBook is easy to extend, particularly, by introducing additional constructions and preprocessing operations.

[1] Strengths and weaknesses of XML documentation development tools are discussed in [26].

3.2. DocLine

The DocLine technology [12] is designed for developing and maintaining complex software documentation on the basis of adaptive reuse [7, 8], which allows configuring text fragments depending on their context. DocLine contains a model of documentation development process and Eclipse-based software tools, as well as supplements DocBook with the decision representation language (DRL). The DRL supports two mechanisms of adaptive reuse: configurable information elements and Multi-view item catalogues.

Information elements. Consider an example. Suppose that we have a news aggregator that downloads news data from various news feeds. Below is a fragment of its documentation (receiving news from RSS and Atom feeds).

`When module instance receives refresh_news call, it updates its data from RSS and Atom feeds it is configured to listen to and pushes new articles to the main storage.` (1)

In addition, the aggregator can receive news from Twitter feeds.

`When module instance receives refresh_news call, it updates its data from Twitter feeds it is subscribed to and pushes new articles to the main storage.` (2)

To reuse the repeating text from (1) and (2), an information element is created:

`<infelement id="refresh_news"> When module instance receives refresh_news call, it updates its data from <nest id="SourceType">> </nest> and pushes new articles to the main storage. </infelement>` (3)

The tag `<nest/>` denotes an extension point (possibility of parameter substitution). When using the information element in a certain context, the extension point can be deleted, replaced, or supplemented with some text. Below is an example of using (3) to form text (2):

`<infelemref infelemid="refresh_news">> <replace-nest nestid="SourceType">>Twitter feeds it is subscribed to </replace-nest> </infelemref>` (4)

In (4), there is a link to the information element defined in (3) `(<infelemref/>)` with its extension point being replaced by a new text fragment `(<replace-nest/>)`.

Multi-view item catalogues. Generally, documentation of software products contains descriptions of entities that have the same functions but represented differently. A special case of a catalogue is a dictionary containing descriptions of terms. Dictionaries are useful for constructing glossaries and for unifying terms contained in documentation. In [10, 12], Multi-view item catalogues are described in more detail.

3.3. Documentation Refactoring

Refactoring is a process of restructuring program code to improve its internal structure without affecting its functionality [29]. In [10], the concept of refactoring was expanded for XML documentation to enable modification of an internal XML representation while preserving finite text representations (for example, PDF). The following set of refactoring operations was developed:

1. selecting common entities (particularly, to transfer from plain text and DocBook to the DRL);

2. tuning key entities;

3. fine-grain reuse with the application of dictionaries and Multi-view item catalogues;

4. renaming various structure elements.

3.4. Software Clone Detection and Clone Miner

Software clone detection is now being intensively developed, and a great number of clone detection tools are presently available [30, 31]. In [11, 12], it was proposed to use clone detection methods and instruments for finding textual repetitions. The Clone Miner tool [13] was used, which is a token-based detector of clones with a command line interface that transforms program code into a list of tokens. The detector is based on suffix arrays [32]. Clone Miner allows adjusting the minimum length of clones (number of tokens). For instance, a text fragment "FM registers" consists of two tokens. For our purposes, Clone Miner has been upgraded to support Unicode, which allows experimenting with texts in different languages.

4. REPETITION SEARCH PROCESS AND DOCUMENT REFACTORING

4.1. Process Flowchart

The general flowchart of the repetition search process is shown in Fig. 1. First, a documentation file is preprocessed and, then, is passed to Clone Miner; the results are filtered, and the user can automatically refactor any clone groups detected. The refactoring procedure yields a reusable information element or dictionary entry.

4.2. Preparation for Clone Detection

DocLine can work only with DRL constructions (particularly, clone detection is carried out for information elements). If a source file contains plain text, then DocLine transforms it into one information element ("transfer to DRL" operation). Thereafter, the document can undergo further processing.

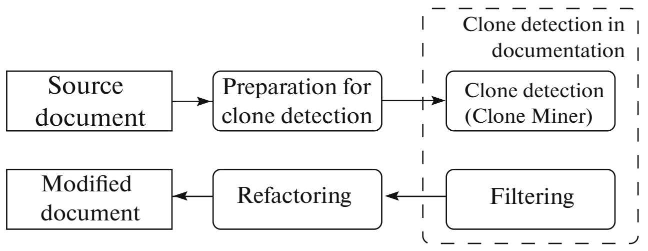

Clone detection in documentation
Source document
Preparation for clone detection
Clone detection (Clone Miner)
Modified document
Refactoring
Filtering

Fig. 1. General flowchart of the repetition search process.

4.3. Clone Detection

Clone Miner searches through plain text, since found repetitions may violate the XML markup. Below is an example in which a detected clone that includes an opening tag (but not a closing one) is italicized. In the process of refactoring, the violated markup of the clones and surrounding text is corrected.

```
<section id="file-tree-isa-directory>>
<title>Reviving incoming calls </title>
<para>
Once you receive an incoming call,
the phone gets CallerID information
and reads it out. But if…</para>
</section> (5)
```

4.4. Filtering

The technical writer makes the final decision about which of the detected clones should be refactored. The list of clones detected by the Clone Miner, however, is quite large (thousands of groups of clones) and must be filtered to get rid of junk. The filtering procedure involves the following steps.

1. Reject clone groups containing less than five symbols (for example, a group "is a" consists of three symbols); generally, such clones have no independent meaning but prove to be rather numerous.

2. Reject groups that contain only XML markup without any text.

3. Reject groups of clones containing common language phrases like, for example, "that is" or "there is a." In the process of document analysis, a dictionary of phrases is constructed, and all clone groups are checked whether their components are included into this dictionary. If there is a match, then the group is rejected.

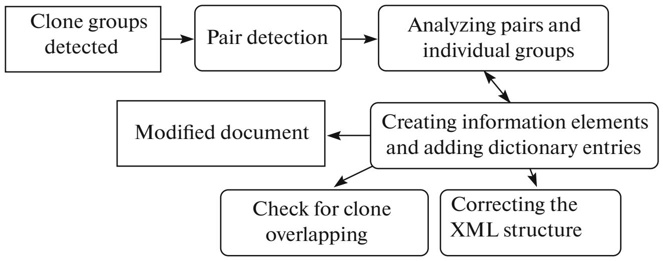

Clone groups detected
Pair detection
Analyzing pairs and individual groups
Modified document
Creating information elements and adding dictionary entries
Check for clone overlapping
Correcting the XML structure

Fig. 2. Refactoring process.

4.5. Refactoring

The filtering procedure yields a set of clone groups. Our final goal consists in using the clones to obtain reusable elements. This goal can be achieved by following the process described below (see Fig. 2). In this process, some operations are executed automatically, but selection of refactoring candidates and application of a particular refactoring operation are performed by the user.

Detection of clone pairs. The set of clone groups detected by Clone Miner is denoted by *SetG*. Pairs of clone groups that contain clones positioned close to one another in the text are selected from *SetG* for refactoring. For example, the following phrase occurs in the text in five different variants with different port numbers: "inet daemon can listen on ... port and then transfer the connection to appropriate handler." This example consists of two groups with five clones in each: the first group contains clones "inet daemon can listen on," while the second group contains clones "port and then transfer the connection to appropriate handler." These two groups can be combined to form one information element with a single extension point that can take values of different ports.

Below, we propose an algorithm for finding pairs of clone groups, which can easily be extended to work with *n* groups.

The distance between clones is a number of text symbols between them (overlapping clones are not considered). Hence, we can find the distance between clone groups G_1 and G_2 if the following conditions are met.

1. The groups contain the same number of clones: $\#G_1 = \#G_2$.

2. The clones in both the groups are ranked in the order of occurrence in a document with a serial number being assigned to each clone. Thus, a set of clone pairs is formed (in all pairs, the first clone belongs to the first group, and the second, to the second group). All pairs contain no overlapping clones: $\forall k \in [1, \#G_1] g_1^k \bigcap g_2^k = \varnothing$, where g_1^k and g_2^k are the clones of the groups G_1 and G_2, respectively.

3. If we have two clones from two groups, and these clones have the same numbers, then one clone proceed to another, and this is true for any such clone pairs:

$$\forall k \in [1, \#G_1] Before(g_1^k, g_2^k) \vee \forall k \in [1, \#G_1] Before(g_2^k, g_1^k),$$

where g_1^k and g_2^k are clones of the groups G_1 and G_2, respectively.

The distance between the groups G_1 and G_2 is defined as $dist(G_1, G_2) = \max_{1 \le k \le G_1} dist(g_1^k, g_2^k)$, where *dist* is a distance (in symbols) between the clones g_1^k and g_2^k. This definition makes it possible, upon selecting a clone group, to compare distances from this group to other groups in order to select the nearest one. In addition, we reject the pairs in which distances between the corresponding clones of the first and second groups vary considerably. For instance, if the distance between clones of the first pair is one symbol, while that between clones of the second pair is 10 000 symbols, then there is almost no chance that these groups can somehow contribute to a common meaning. Having experimented, we came to a conclusion that, for pairs of clone groups, the variance of distances between their clones should be bounded from above at 2000: $Var(\{dist(g_1^k, g_2^k) \vee k \in [1, \#G_1], g_1^k \in G_1, g_2^k \in G_2\}) \leqslant 2000$. Thus, we eliminate the pairs with greater variance from consideration.

The algorithm searches through all groups from *SetG* and finds the nearest one for each of them. With such groups found (if any), a new pair of groups is included into the set *PairG*; henceforth, these groups are not considered.

Analysis of pairs and individual groups. At this step, pairs of clone groups and individual groups are included into a list *L*:

$$L = PairG \bigcup \{G | G \in SetG \vee \nexists P \in PairG : G = left(P) \vee G = right(P)\}.$$

From this list, the user selects desired groups to form reusable information elements or dictionary entries (hereinafter, we refer to the elements of this list as refactoring candidates or, simply, candidates). It is important for reusable text fragments to be meaningful (for example, be a part of a description for a function or interrupt). Reuse that is based only on syntax and ignores semantics is quite ineffective. But it seems impossible to analyze meaningfulness of candidates automatically, and this is a reason why the user is provided with a browser interface for viewing and executing refactoring operations.

The candidates in the list L are ranked in the descending order of their lengths (in symbols). The length of a clone is the number of its symbols, while the length of a group is a sum of lengths of clones in this group: $\forall G \in L length(G) = \#G \cdot length(g),\, g \in G$. The length of a pair is found as a sum of lengths of groups constituting this pair: $length(Pair(G_1, G_2)) = length(G_1) + length(G_2)$. The candidates containing the maximum amount of text (candidates that are most preferable for reuse) are placed at the top of the list L.

Creation of information elements and dictionary updating. For a selected candidate, the user can execute the following refactoring operations (see Subsection 3.3): create an information element, create a variative information element, and add a dictionary entry.

Before executing these operations, the proposed algorithm checks whether the clones of a candidate are overlapped with the clones that have already undergone refactoring. Clone Miner allows clone groups to overlap, since it does not imply the further processing. In our case, such overlapping leads to refactoring errors. The clones overlapping with already used ones are eliminated from the candidate list.

Then, the algorithm corrects XML markup of the candidate and the corresponding document context. As noted above, Clone Miner yields results that are incorrect in terms of XML, while DocLine can work only with correct DRL and DocBook documents. Our toolkit balances the violated XML markup (opens and closes all missing tags). Note that a full-fledged implementation of such balancing is quite a complex problem. For instance, if the tag `<para>` (sets a paragraph) is unnecessarily opened and closed one more time, then the resultant DocBook document will contain two paragraphs instead of one. Handling such situations is one of the problems to be addressed in further works.

Once the refactoring procedure for the selected candidate is complete, coordinates of the other candidates in text are recalculated according to the changes introduced into the document. The user can also return to the analysis of pairs and groups (candidates).

5. DOCUMENTATION REFACTORING TOOLKIT

To implement the process described above, the documentation refactoring toolkit is proposed [14], The toolkit is integrated with DocLine, but can be used independently of Eclipse and DocLine, since it is implemented in Python. The documentation refactoring toolkit allows browsing candidates and original documentation text simultaneously, as well as selecting candidates for refactoring.

Figure 3 shows the candidate browser. Rows of the table "Refactoring candidates" correspond to particular candidates. The context menu allows the user to choose a refactoring operation for a selected candidate (create an information element or add a dictionary entry). For an individual group, links to its clones (under the candidate text) are highlighted with alternating colors (see numbers in braces under the clone text for the first candidate in Fig. 3); for a pair of clones, alternating colors indicate different variants of text in the extension point (see information about the second candidate in Fig. 3). The section "Source text" contains the source text of a document. When clicking on a variant in the table, a source text fragment, which consists of two clones with a selected text variant between them, is highlighted. When clicking on a clone link in the table, the corresponding clone is highlighted in the section "Source text." Highlighting allows the user to see not only the selected text fragment but also its context.

6. EVALUATION

To evaluate the effectiveness of the proposed approach, a series of experiments were carried out on specially prepared tests and on DockBook documentation for some open-source projects (see the list of the projects and documents in Table 1).

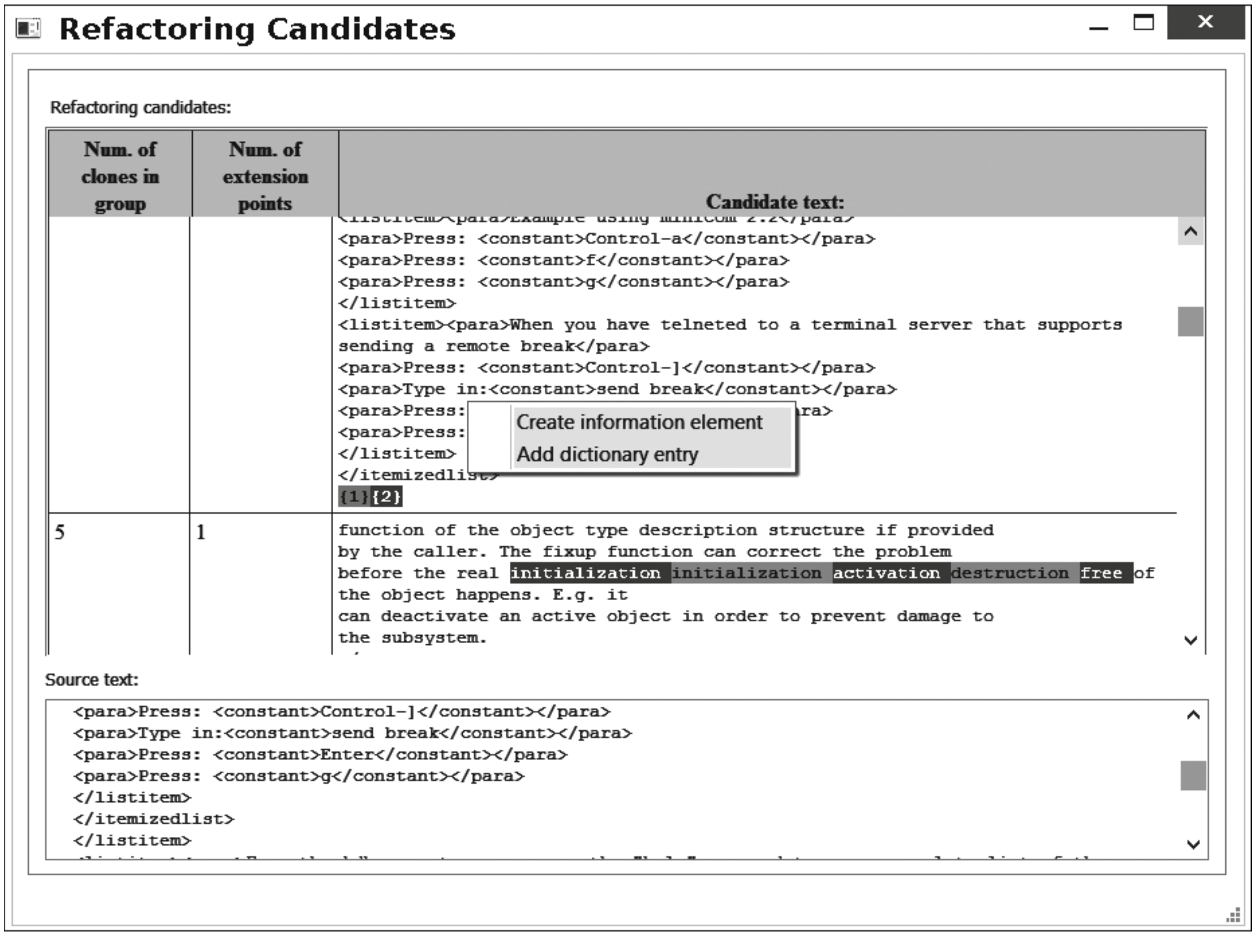
We used the Goal-Question-Metric (GQM) approach [33] with a goal to evaluate the proposed approach and tool. The following questions correspond to the goal:

- question 1: quality of clone detection;
- question 2: effectiveness of clone filtering;
- question 3: refactoring capabilities.

To answer each of these questions, a special series of experiments were conducted.

The first series of experiments concerned the use of Clone Miner. A small set of documents with repetitions (in the number of 10) was created manually. It was found that Clone Miner ignored the last repeating token in the clones. Upon making some improvements, the proposed toolkit successfully detected all repetitions.

The experiments to answer the second question involved the documentation from Table 1. As a filtering effectiveness metric, we used the number of clones fil-



Refactoring candidates:

Num. of clones in group	Num. of extension points	Candidate text:
		<listitem><para>Example using minicom 2.2</para> <para>Press: <constant>Control-a</constant></para> <para>Press: <constant>f</constant></para> <para>Press: <constant>g</constant></para> </listitem> <listitem><para>When you have telneted to a terminal server that supports sending a remote break</para> <para>Press: <constant>Control-]</constant></para> <para>Type in:<constant>send break</constant></para> <para>Press: ra> <para>Press: </listitem> </itemizedlist> {1}{2}
5	1	function of the object type description structure if provided by the caller. The fixup function can correct the problem before the real initialization initialization activation destruction free of the object happens. E.g. it can deactivate an active object in order to prevent damage to the subsystem.

Source text:

```
<para>Press: <constant>Control-]</constant></para>
<para>Type in:<constant>send break</constant></para>
<para>Press: <constant>Enter</constant></para>
<para>Press: <constant>g</constant></para>
</listitem>
</itemizedlist>
</listitem>
```

Fig. 3. Documentation Refactoring Toolkit.

tered based on the criteria described in Subsection 4.4. The results are shown in Table 2; note that, on the average, filtering reduces the number of clones by 13.2%.

The number of candidates after filtering is shown in Table 3 for the minimum clone length (a Clone Miner parameter) of one and five tokens. For five tokens, the number of candidates is far less. The rejected short clones, however, can be useful for constructing clone pairs and for completing the dictionary, as well as in some other important cases. On this evidence, we recommend the technical writer to work with clones more than one token long. Ranking the candidate list will place most of short candidates at the bottom.

In the experiments to answer the third question, the "amount of reuse" metric [34] was employed. It is calculated as a ratio between the volume of the reusable text and the overall volume of the documentation:

Table 1. Documents used in the experiments

Project	Documentation	Short form	Size
Linux Kernel (open-source basis of the Linux OS)	Linux Kernel Documentation [15]	LKD	892 KB
Zend Framework (open-source PHP framework for web site and service development)	Zend PHP Framework documentation [16]	Zend	2924 KB
Subversion (centralized version control system)	Version Control with Subversion For Subversion 1.7 [17]	SVN	1810 KB
DocBook (language and toolset for documentation development)	DocBook 4 De_nitive Guide [18]	DocBook	686 KB

Table 2. Filtering results

Filtering method	LKD, %	Zend, %	SVN, %	DocBook, %	On average, %
Rejecting clones under 5 symbols in length	7.3	4.8	4.4	7.2	5.9
Rejecting pure XML markup clone groups	3.3	5.8	2.4	6.0	4.4
Rejecting common language phrases	3.2	2.2	2.9	3.4	2.9
Total	13.8	12.8	9.7	16.6	13.2

Table 3. Number of candidates for clones from 1 and 5 tokens long

Number of candidates	LKD	Zend	SVN	DocBook
Individual	12819/1034	33400/5213	27847/3119	8228/870
Pairs	351/108	1400/613	616/249	232/50
Total	13170/1254	34800/5826	28463/3368	8460/920

$$\frac{c \in \sum_{all\,candidates} length(C)}{length(T)},$$

where $length(C)$ is the number of symbols in a clone (or in a clone pair) multiplied by the number of clones in the corresponding group(s) (see Subsection 4.5), while $length(T)$ is the length of the whole text in symbols.

The size of the documents used was measured in symbols. On the average, for all documents (see Table 1), the amount of reuse was from 48% to 53% for the clones more than 1 token long and from 21% to 28% for the clones more then 5 tokens long. These results suggest that, in the case of automatic refactoring, reuse can affect documentation quite considerably. However, it is difficult to evaluate actual amount of reuse, since it is reasonable to select only semantically significant repetitions. To obtain more accurate estimates, more experiments are required.

7. CONCLUSIONS

The approach proposed in this paper can be applied to processing software product lines documentation. It allows selecting reusable text fragments to restructure documentation of various software products, thus simplifying its maintenance and improving its quality. In addition, the proposed approach can be used for variation management [35, 36].

When conducting the experiments, we also came to the conclusion that our toolkit needs more flexible construction of information elements and a more user-friendly interface.

The results of our experiments show that, even after filtering, we have to deal with a great number of "junk" clones. Improving the accuracy of filtering is one of the main problems to be addressed in further works. It is also required to enhance the adaptive reuse to allow working with candidates having more than one extension point.

In addition, support of semantically oriented reuse can make it possible to combine our approach with various software traceability techniques [37–40], as well as to establish connections between documentation and other artifacts: program code, requirements, models, etc. In this case, reuse can improve the quality of these connections, while semantically oriented reuse can increase their accuracy.

In addition to the field of software engineering, the proposed approach can find its application in knowledge management, ontology engineering [41–44], and enterprise architecture modeling [45, 46]. In toolsets being designed and used in these fields, models generally have an XML structure and often contain repetitions, since, when constructing these models, analysts have to work with a large amount of weakly structured data (documents, comments to models, logical names of entities, and so on).

REFERENCES

1. Watson, R., Developing best practices for API reference documentation: Creating a platform to study how programmers learn new APIs, *Proc. IPCC,* 2012, pp. 1–9.

2. Garousi, G., Garousi, V., Moussavi, M., Ruhe, G., and Smith, B., Evaluating usage and quality of technical software documentation: An empirical study, *Proc. EASE,* 2013, pp. 24–35.

3. Parnas, D.L., Precise documentation: The key to better software, *The Future of Software Engineering,* Nanz, S., Ed., Springer, 2011.

4. Shalyto, A.A., New initiative in programming: Drive for open project documentation, *PC Week RE,* 2003, no. 40, pp. 38–42.

5. Holmes, R. and Walker, R.J., Systematizing pragmatic software reuse, *ACM Trans. Software Eng. Methodol.,* 2013, vol. 21, no. 4, p. 44.

6. Czarnecki, K., Software reuse and evolution with generative techniques, *Proc. IEEE/ACM Int. Conf. Automated Software Engineering,* 2007, p. 575.

7. Bassett, P., The theory and practice of adaptive reuse, *SIGSOFT Software Eng. Notes,* 1997, vol. 22, no. 3, pp. 2–9.

8. Jarzabek, S., Bassett, P., Zhang, H., and Zhang, W., XVCL: XML-based variant configuration language, *Proc. ICSE,* 2003, pp. 810–811.

9. Koznov, D. and Romanovsky, K., DocLine: A method for software product lines documentation development, *Program. Comput. Software,* 2008, vol. 34, no. 4, pp. 216–224.

10. Romanovsky, K., Koznov, D., and Minchin, L., Refactoring the documentation of software product lines, *Lect. Notes Comp. Sci.,* 2011, vol. 4980, pp. 158–170.

11. Koznov, D.V., Shutak, A.V., Smirnov, M.N., and Smazhevskii, M.A., Clone search for technical documentation refactoring, *Komp'yuternye instrumenty v obrazovanii,* 2012, no. 4, pp. 30–40.

12. Lutsiv, D.V., Koznov, D.V., Basit, H.A., Li, O.E., Smirnov, M.N., and Romanovskii, K.Yu., Method for repeating text fragments search in technical documentation, *Nauchno-Tekh. Vestn. Inf. Tekhnol. Mekh. Opt.,* 2014, vol. 4, no. 92, pp. 106–114.

13. Basit, H.A., Smyth, W.F., Puglisi, S.J., Turpin, A., and Jarzabek, S., Efficient token-based clone detection with flexible tokenization, *Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering,* 2007, pp. 513–516.

14. Mathematics and Mechanics Faculty of the St. Petersburg State University, Document Refactoring Toolkit. http://www.math.spbu.ru/user/kromanovsky//docline/index_en.html.

15. GitHub, Linux Kernel Documentation. https://github.com/torvalds/linux/tree/master//Documentation/DocBook.

16. GitHub, Zend PHP Framework documentation. https://github.com/zendframework/zf1/tree//master/documentation.

17. SourceForge, SVN Book. http://sourceforge.net/p/svnbook/source/HEAD/tree/trunk/en/book.

18. SourceForge, DocBook Definitive Guide. http://sourceforge.net/p/docbook/code/HEAD/tree/trunk/defguide/en.

19. Zhi, J., Garousi, V., Sun, B., Garousi, G., Shahnewaz, S., and Ruhe, G., Cost, benefits and quality of technical software documentation: A systematic mapping, *J. Syst. Software,* 2012, pp. 1–24.

20. Zhong, H., Zhang, L., Xie, T., and Mei, H., Inferring source specifications from natural language API documentation, *Proc. 24th ASE,* 2009, pp. 307–318.

21. Zhong, H. and Su, Z., Detecting API documentation errors, *Proc. SPASH/OOPSLA,* 2013, pp. 803–816.

22. Wingkvist, A., Lowe, W., Ericsson, M., and Lincke, R., Analysis and visualization of information quality of technical documentation, *Proc. 4th Eur. Conf. Information Management and Evaluation,* 2010, pp. 388–396.

23. Wingkvist, A., Ericsson, M., and Lowe, W.A, Visualization-based approach to present and assess technical documentation quality, *Electron. J. Inf. Syst. Eval.,* 2011, vol. 14, no. 1, pp. 150–159.

24. Applied Research in System Analysis, VizzAnalyzer Clone Detection Tool. http://www.arisa.se/vizz_analyzer.php.

25. Walsh, N. and Muellner, L., *DocBook: The Definitive Guide,* O'Reilly, 1999.

26. Darwin Information Typing Architecture (DITA) Version 1.2. http://docs.oasis-open.org/dita/v1.2/os/spec/DITA1.2-spec.pdf.

27. Koznov, D.V., Shutak, A.V., Smirnov, M.N., and Smazhevskii, M.A., Clone search for technical documentation refactoring, *Komp'yuternye instrumenty v obrazovanii,* 2012, no. 4, pp. 30–40.

28. Lutsiv, D.V., Koznov, D.V., Basit, H.A., Li, O.E., Smirnov, M.N., and Romanovskii, K.Yu., Method for repeating text fragments search in technical documentation, *Nauchno-Tekh. Vestn. Inf. Tekhnol. Mekh. Opt.,* 2014, vol. 4, no. 92, pp. 106–114.

29. Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.

30. Rattan, D., Bhatia, R.K., and Singh, M., Software clone detection: A systematic review, *Inf. Software Technol.,* 2013, vol. 55, no. 7, pp. 1165–1199.

31. Akhin, M. and Itsykson, V., Clone detection: Why, what and how, *Proc. CEE-SECR,* 2010, pp. 36–42.

32. Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E., Replacing suffix trees with enhanced suffix arrays, *J. Discrete Algorithms,* 2004, vol. 53.

33. Basili, V.R., Caldeira, G., and Rombach, H.D., The Goal Question Metric Approach, Wiley, 1994, vol. **1**, pp. 528–532.

34. Frakes, W. and Terry, C., Software reuse: Metrics and models, *ACM Comput. Surv.,* 1996, vol. 28, no. 2, pp. 415–435.

35. Krueger, C.W., Variation management for software product lines, *Proc. SPL,* San Diego, 2002, pp. 37–48.

36. Koznov, D.V., Novitskii, I.A., and Smirnov, M.N., Variation management tools: Ready for industrial application, *Tr. S.-Peterb. Inst. Inf. Avtom. Ross. Akad. Nauk,* 2013, no. 3, 297–331.

37. Abadi, A., Nisenson, M., and Simionovici, Y.A, Traceability technique for specifications, *Proc. ICPC,* 2008, pp. 103–112.

38. Terekhov, A.N. and Sokolov, V.V., Document implementation of the conformation of MSC and SDL diagrams in the REAL technology, *Program. Comput. Software,* 2007, vol. **33**, no. 1, pp. 24–33.

39. Koznov, D.V., Smirnov, M.N., Dorokhov, V.A., and Romanovskii, K.Yu., WebMLDoc: An approach to automatic change tracking in user documentation of Web applications, *Vestn. S.-Peterb. Univ., Ser. 10 Appl. Math. Inf. Protsessy Upr.,* 2011, no. 3, pp. 112–126.

40. Smirnov, M.N., Koznov, D.V., Dorokhov, V.A., and Romanovskii, K.Yu., WebMLDoc software environment for automatic change tracking in user documentation of Web applications, *Sist. Program.,* 2010, vol. 5, no. 1, pp. 32–51.

41. Gavrilova, T.A., Ontological engineering for practical knowledge work, *Proc. 11th Int. Conf. Knowledge-Based*

and Intelligent Information and Engineering Systems, 2007.

42. Kudryavtsev, D. and Gavrilova, T., Diagrammatic knowledge modeling for managers: Ontology-based approach, *Proc. Int. Conf. Knowledge Engineering and Ontology Development,* 2011, pp. 386–389
43. Bolotnikova, E.S., Gavrilova, T.A., and Gorovoy, V.A., To a method of evaluating ontologies, *J. Comput. Syst. Sci. Int.,* 2011, vol. 50, no. 3, pp. 448–461.
44. Gavrilova, T.A., Gorovoy, V.A., and Bolotnikova, E.S., Evaluation of the cognitive ergonomics of ontologies on the basis of graph analysis, *J. Sci. Tech. Inf. Process.,* 2010, vol. 37, no 6, pp. 398–406.
45. Grigoriev, L. and Kudryavtsev, D., ORG-Master: Combining classifications, matrices, and diagrams in the enterprise architecture modeling tool, *Communications in Computer and Information Science,* Springer, 2013, pp. 250–258.
46. Koznov, D.V., Arzumanyan, M.Yu., Orlov, Yu.V., Derevyanko, M.A., Romanovskii, K.Yu., and Sidorina, A.A., Specificity of projects in the field of enterprise architecture design, *Biznes-informatika,* 2005, no. 4., pp. 15–26.

Translated by Yu. Kornienko