Clone Detection in Reuse of Software Technical Documentation

Dmitrij Koznov[1], Dmitry Luciv[1], Hamid Abdul Basit[2], Ouh Eng Lieh[3], and Mikhail Smirnov[1]

[1] Saint Petersburg State University
d.koznov@spbu.ru,dluciv@math.spbu.ru,smnsmn1979@gmail.com
[2] Lahore University of Management Sciences
hamidb@lums.edu.pk
[3] National University of Singapore
issoel@nus.edu.sg

Abstract. As software documentation is becoming more and more complicated, efficiency of maintenance process could be increased through documentation reuse. In this paper, we apply software clone detection technique to automate searching of repeated fragments in software technical documentation to be reused. Our approach supports adaptive reuse, which means extracting "near duplicate" text fragments (repetitions with variations) and producing customizable reusable elements. We present a process and a tool, which can work with both DocBook documentation (widely used XML markup language) and DRL (DocBook extension with adaptive reuse features), as well as with plain text. Our tool is based on Clone Miner software clone detection tool, and integrated to DocLine environment (adaptive reuse documentation framework), providing visualization and navigation facilities on the clone groups found, and also supporting refactoring to extract clones into reusable elements.

Keywords: software technical documentation, documentation reuse, software clone detection, adaptive reuse, refactoring, DocBook, DocLine, DRL

1 Introduction

Software documentation is a significant component of modern software. It is supposed to help software engineers to comprehend a given software system and accomplish development and modification tasks more efficiently [1]. There are two types of software documentation: technical documentation (requirement specifications, design documents, etc.), and user documentation (e.g., user guides). Sometimes API documentation is considered, that is a special case of technical documentation and describes application programming interfaces of reusable code libraries [2]. In this paper, we consider technical software documentation only.

It should be noted, that technical documentation may have considerable size and complex structure, and like the software itself, is constantly changed during development process. The quality of technical documentation is a well-known problem that has not been resolved in the last decades [3]. One of the reasons that leads to essential decrease of documentation quality during maintenance process is that documents may contain numerous repetitions. If there is no traceability between duplicate text fragments, we need to modify each fragment manually while making changes. But in practice it is hard to keep the documentation updated because of huge volumes and lack of time. That leads to accumulation of mistakes and contradictions in documentation.

The situation is even more complicated because very often duplicate information is “near duplicate”, e.g., in one document the same software features may be described many times with different level of details. Also, there are sets of similar objects, which are described on the documentation: functions, interruptions, signals, etc. If objects belong to the same set then their descriptions have a lot of commonalities, but at the same time they differ from each other. This leads to repetitions with variations, and makes it difficult to apply usual text search techniques to find such repetitions. Moreover, it is necessary not only to search duplicate text fragments, but to manage them consistently.

Systematic reuse techniques attempt to simplify the software maintenance process. Different techniques of software reuse have been proposed [4, 5]. One of these techniques is XVCL [6], which is based on adaptive reuse introduced by Paul Bassett [7]. These ideas were applied for software documentation reuse in DocLine framework [8]. In this context, refactoring of XML documentation technique was also explored [9] to simplify the maintenance process of existing documentation extracting reusable text fragments. But the challenge of automatically searching for reusable document fragments still remained open.

This paper closes the gap using software clone detection technique [10, 11]. Our approach is designed for operating with XML documentation in DocBook [12] and DRL [8] markup languages, as well as with plain Text (i.e. ASCII/UNICODE format). DocBook is a wide-spread XML language for software documentation development in Linux/Unix community, while DRL is an extension of DocBook for implementation of adaptive reuse approach. We used Clone Miner [13] as a clone detection tool, filtering and correcting its outputs. Based on clones found by the tool, the approach supports refactoring of documentation, i.e. producing customizable reusable elements and inserting them into the text. We consider not only exact duplicates, but also “near duplicate" text fragments, applying adaptive reuse technique [6, 7]. We implemented the approach as a tool that incorporates visualization and navigation facilities on the detected clone groups and provides seamless invocation of DocLine refactoring operations. The paper includes the results of the evaluation the proposed approach whereby we applied our tool for DocBook documentation of several open source projects, and, in particular, for Linux Kernel Documentation (LKD) [14].

Acknowledgements. The authors thank the students Artem Shutak, Dmitry Kopin, Mikhail Smarzhevskij and Adeel Khan, who implemented the draft versions of selected parts of the solution, and participated in discussions.

2 Related works

Technical documentation development currently widely employs XML markup languages. The widely used standards are DocBook [12] and DITA [15], both supporting modular approach and enabling development of reusable documentation components. In [8] adaptive software reuse technique of Bassett-Jarzabek [6, 7], has been applied to documentation. But all of these approaches imply that documentation is developed as reusable modules from the very beginning, and they do not offer approaches and tools for searching and extracting repetitions. Meanwhile, documentation maintenance often requires eliminating inconsistencies, because previous corrections were local and made by different persons, in different manners. Searching "near duplicate" text fragments and extracting reusable text elements could simplify the maintenance process. Moreover, this may also lead to correction of descriptions of similar code objects (signals, functions of API, handlers, etc.) for better unification to facilitate future changes. The adaptive reuse technique of XVCL is helpful in this regard. In [9] refactoring of documentation was suggested to extract adaptive reusable elements. But no tools to search repeatable fragments were available.

A systematic review of the software documentation domain is presented in [16]. Below, we overview some of the studies, which provide automatic analysis and transformation of documentation.

Zhong et. al. [17] suggested an approach to infer resource specifications from API documentation. The approach overcomes the problem that developers tend to ignore information in API documentation. But if some part of the code is automatically generated on technical documentation, the problem is solved. The paper proposes to generate resource specification on documentation.

An approach to detect documentation errors comparing code samples and corresponding document fragments is proposed in [18]. The approach is based on comparing code objects that are mentioned in the text (data types, procedures, variables, etc.) with the ones in the samples.

Garousi et. al. [1] suggests to analyze the usage and quality of software projects' documentation during development and maintenance phases, based on projects' data and experts' opinion from a survey-based questionnaire.

Metrics to measure documentation quality are proposed in [19, 20]. The authors also adapt the VizzAnalyzer clone detection tool [21] to provide a measurement of a documents uniqueness. However, further use of found clones is only briefly discussed and their automatic transformation for future reuse is not done.

To summarize, little attention is given to search repetitions in software technical documentation to extract reusable elements. The issue is only touched upon in [19, 20], but no approach applies the idea of adaptive reuse to software technical documentation.

3 Background

3.1 DocBook

DocBook [12] is a collection of standards and tools for technical writing, particularly used for large and highly structured content. The key difference between DocBook and other structured formats (e.g., LaTeX) is that the style (bold, font size, italics etc.) is separated from the structured content. This allows one source document to have many presentations, such as HTML, PDF, etc. Unlike other document markup tools, DocBook is not WYSIWYG technology (What You See Is What You Get). It provides more flexibility, and allows to create more reliable documents, but demands for technical writers to be more experienced than Microsoft Word users (discussion about usage of markup languages by technical writers can be found in [22]). DocBook may be easily extended, and it is possible to use these extensions in practice: you only need to perform preprocessing specifications to eliminate extended constructs into plain DocBook, and after that you may use the standard DocBook utilities to get target document presentations (e.g., PDF).

3.2 DocLine

DocLine [8] is created for the development and maintenance of complicated software documentation basing on adaptive reuse [6, 7] to operate with duplicate documentation fragments. Adaptive reuse means that reusable text fragments can be configured for each context where they are inserted.

DocLine provides a new XML markup language DRL, a model of documentation development process, and a toolset integrated into Eclipse IDE. DRL (Documentation Reuse Language) extends DocBook [12] providing two mechanisms of adaptive reuse: customizable information elements and multi-view item catalogs.

Customizable information elements. This can be understood with the help of a simple example. Let us consider a news aggregator that provides news feed from different sources. A description of the module to refresh news from RSS and Atom feeds can be the following:

```
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)
```

Meanwhile, the news aggregator can also use Twitter as a news feed, and the description of corresponding module can be as follows:

```
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 provide reuse of duplicate text in (1) and (2) using an adaptive reuse technique, the corresponding information element must be specified in DRL:

```
<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)
```

In this example, we define an information element (`<infelement/>` tag) and an extension point inside it (`<nest/>` tag). When this information element is included in a particular context, the extension point can be removed, replaced or appended with custom content without having to modify the information element itself. The following customization transforms(3) into (2):

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

The example (4) shows a reference to the information element defined in (3) (`<infelemref/>`) and the replacement of the extension point defined in this information element by new content (`<replace-nest/>`).

Multi-view item catalogs. In the documentation of most Software products one can find descriptions of typical items of the same kind. To organize adaptive reuse for that case a multi-view item catalog is introduced in DRL. The catalog contains a collection of items represented by a set of attributes. When a technical writer includes a catalog item into a particular context, s/he must indicate the corresponding representation template and the item identifier. Then, the content of the template will be inserted into the target context and all the references to the attributes will be replaced by corresponding attribute values. A particular case of the catalog is a dictionary, which contains a set of terms without presentation templates. Dictionaries are useful for creating glossary to unify naming policy in documentation. More details about multi-view item catalogs can be found in [8, 9].

3.3 Refactoring Documentation

Refactoring is the process of changing a software system in such a way that it does not change the external behavior of the code, yet improves its internal structure [24]. In [9], refactoring was adapted to XML documentation maintenance. In this case, refactoring means the change of internal document specification (XML markup constructs), and preservation of output document presentation (e.g., pdf file). Based on this idea, a number of refactoring operations were designed for DocLine [9].The operations can be divided into the following groups:

1. Operations for extracting common assets, and, in particular, for transition to DRL from plain text or DocBook.

2. Operations to facilitate core assets tuning (extending their configurability).
3. Operations to facilitate the use of small-grained reuse constructions — dictionaries and multi-view item catalogs.
4. Operations for renaming various structural elements of documentation.

3.4 Software Clone Detection and Clone Miner

Very often software is reused by means of copy/paste. It produces duplicate code (software clones), and that may lead to serious maintenance problems. Clone detection methods and tools are aimed to find different kinds of duplicate code to perform refactoring based on reuse techniques. Systematic review of clone detection methods and tools can be found in [11], while interesting discussion about code cloning and clone detection is presented in [10].

This area is quite mature; there are a number of ready-to-use tools. We selected Clone Miner tool [13] as it is a simple command line tool that could easily integrate into the DocLine framework. Clone Miner is a token-based code clone detector. It converts the input source code into a string of lexical tokens and then applies suffix array based string matching algorithms to find the repeated parts of this string as clone groups.

The tool allows varying the minimal length of clones to be searched, measuring it in terms of the number of tokens. A token in the context of text documents is one single word separated from other words by some separator: '.', '(', ')', etc. For example, the following text fragment consists of 2 tokens: "FM registers".

Clone Miner was extended for this project to support plain text and Unicode inputs, which made it possible to apply the tool to Russian language documents as well.

4 The Process of Clone Detection and Refactoring

4.1 Overview

The general scheme of the process is shown in Fig. 1. The input of the process is a DRL file, which the user prepares for clone detection. After that s/he starts document clone detection by launching Clone Miner which generates the output results. Once the user gets the list of clone groups, s/he can execute the automated refactoring for any clone group. In refactoring, all occurrences of clone selected are replaced by references to reusable element definition.

4.2 Preparation for Clone Detection

DocLine operations are executed for DRL constructs, in particular, the searching of clones is applied to information elements. If the user wants to apply document clone detection for plain text or DocBook documents, s/he has to first perform refactoring operation "Transition to DRL". As a result, a new information element appears that includes the whole original text. Clone detection is then performed on this information element.

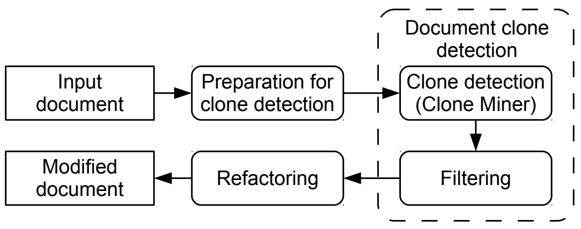


Fig. 1. Process overview

4.3 Clone Detection

We start Clone Miner in the "search in flat text" mode since actually we need flat text search: the repetitions in question might be found inside XML structures. Therefore, the found clones might violate XML markup. (5) shows a text fragment with a found clone *emphasized*. It includes the start tag but not the end tag. The clones become correct in terms of XML as a result of refactoring operations, and so does their context in the document.

```
<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

We use clones detected in refactoring, since it is the technical writer who is responsible for choosing the candidates for refactoring based on their semantic meaningfulness. Meanwhile the number of clone groups detected is so large that they need to be filtered. Our algorithm filters Clone Miner output by the following steps:

1. A clone group is rejected if clone length in the group is less than 5 symbols (e.g. "is a" contains 3 symbols): as a rule, such clones have no semantics, but usually a lot of such groups are found. Some terms can be lost, especially abbreviations, but this is the way to reduce considerably the number of insignificant clone groups. It should be reminded that we measure the clone length in number of tokens in the paper (it means the number of symbol sequences separated by the comma, space, etc.), but in this case we do it in terms of symbols, because the length is too small.
2. We eliminate the groups containing clones consisting only of XML constructs and do not contain output text: we have no task to organize XML markup constructs reuse.

3. We remove the clone groups consisting of phrases “that is”, “there is a”, etc.: these clones have no software semantics (this issue is discussed in section 6). To avoid such clones we have elaborated the dictionary of such expressions based on our own experiments, and we check every clone group if its clones belong to the dictionary. In the future more sophisticated analysis techniques can be used, considering natural-language patterns embedded [23] into strictly defined DRL markup.

4.5 Refactoring

After previous steps, we have a set of clone groups. But our aim is to use clones to extract reusable elements. It can be done using the refactoring process described below. The process uses refactoring operations which have been suggested in [9], but some additional activities have to be performed. The schema of the refactoring process is presented in Fig. 2.

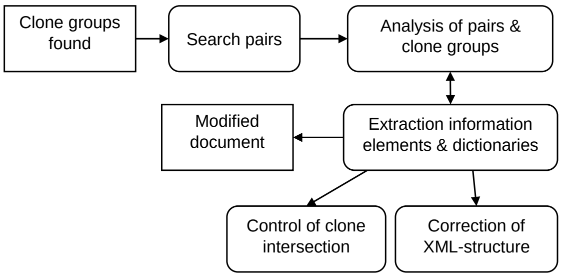

Clone groups found
Search pairs
Analysis of pairs & clone groups
Modified document
Extraction information elements & dictionaries
Control of clone intersection
Correction of XML-structure

Fig. 2. Refactoring process

Searching close pairs. We have the list of clone groups found by Clone Miner in (*SetG*). To provide adaptive reuse, we search clone groups from *SetG* where clones are located close to each other. For example, the following phrase can be found in the text 5 times with different variations (various port numbers): “inet daemon can listen on ... port and then transfer the connection to appropriate handler”. In this situation we have 2 clone groups with 5 clones in every group: one group include clones “inet daemon can listen on”, while the other includes “port and then transfer the connection to appropriate handler”. We want to combine these clone groups in a single information element with one extension point to capture different port numbers.

To find out such kinds of clone groups we propose an algorithm. The algorithm works only with two clone groups, because our observations show that this is the most popular case (we plan to extend the algorithm in the future for n clone groups). We define the distance between two clones as the number of symbols between them (we do not consider a case of intersected text fragments).

We define the distance between two clone groups G_1 and G_2 under the following constraints:

1. They have the same number of clones: $\#G_1 = \#G_2$.
2. We introduce an ordering for clones in a group based on their appearance in the document, and assign a number to each clone. As a result we have a set of clone pairs, where the first member belongs to one group, the second belongs to another, and for all pairs the first members belong to the same group, and the second members belong to the second one. Clones in the same pair are not intersecting, i.e. they are not overlapping in the text:
   $\forall k \in (1..\#G_1)\ g_1^k \ \cap \ g_2^k = \emptyset$,
   where g_1^k and g_2^k are members of G_1 and G_2 groups respectively.
3. For every pair of clones (one clone belongs to first group, another belongs to the second group, and both clones have the same number) clone from one group occurs before clone from the other group in the document:
   $(\forall k \in (1..\#G_1) Before(g_1^k, g_2^k)) \bigvee (\forall k \in (1..\#G_1) Before(g_2^k, g_1^k))$,
   where g_1^k and g_2^k are members of G_1 and G_2 groups respectively.

The distance between G_1 and G_2 is $dist(G_1, G_2) = \max(dist(g_1^k, g_2^k))$, where $dist(g_1^k, g_2^k)$ is a distance between clones (text fragments) g_1^k and g_2^k. We use this simple formula, because we have just one special requirement for distance between clone groups: if we choose a clone group it should be possible to compare a distances from this group to others to select the closest one. But we would not like to consider unreal distances, that is, variation of distances between clone pairs for selected groups should not be too big. For example, if the distance between the first clone pair is 1 symbol, and the distance between the second pair is 10 000 symbols then there is no chance, that these pairs are semantically connected, and it would not be sensible to create information element with extension point. Following our experiments, we have defined the maximum of distance variance between clones from two group as constant 2000: $\mathrm{Var}(\{dist(g_1^k, g_2^k) | k \in (1..\#G_1), g_1^k \in G_1, g_2^k \in G_2\}) \leq 2000$. If the variance is greater, we do not consider this pair.

The algorithm of searching close pairs considers all clone groups from $SetG$ and for every group finds the closest one. If it is successful, a new pair is added to the set $PairG$. If it is not successful for selected clone group (e.g. there is no other clone group with the same number of clones) the resulting list have no pair with this clone group.

Analysis of pairs & clone groups. In this step we combine the clone group pairs and the initial list of clone groups in a list L to present the information to the user for making decision: what text fragments should be extracted as information elements/dictionaries (elements of L we will call *candidates for refactoring* or shortly – *candidates*). The problem is that reusable text elements have to have some semantic, e.g. to be a typical description of a function or interruption. If reuse relies only on syntax and have no semantics, it looks useless. But it is hard to do such analysis automatically, that is why we provide browsing facilities to help the user to make the right decision.

L includes clone group pairs and single groups, which are not included in any pair: $L = PairG \bigcup \{G \mid G \in SetG \; \& \; \nexists P \in PairG : G = left(P) \vee G = right(P)\}$.

We order L by the length of elements measuring length in a symbols in descending order. The length of clone is the number of the symbols in a clone. The length of clone group is sum of lengths of all clones from a group: $\forall G \in L$ $length(G) = \#G \cdot length(g)$, where $g \in G$, and $\#G$ is a number of clones in G. It should be reminded that all clones from a group are duplicate text fragments, that is why all of them have equal length. The length of the clone group pair is the sum of the lengths of clone groups included in the pair: $length(Pair(G_1, G_2)) = length(G_1) + length(G_2)$. We can see elements at the top of the L , which contain most "amount" of text, and these elements are most preferable for reuse. The user should select manually a group or a pair to perform refactoring operations.

Extraction information elements & dictionaries. For each clone group or clone group pair selected before the user can apply the following refactoring operations (see section 3.3, group 1): *extracting information element, extracting information element with variations, or extracting to dictionary.*

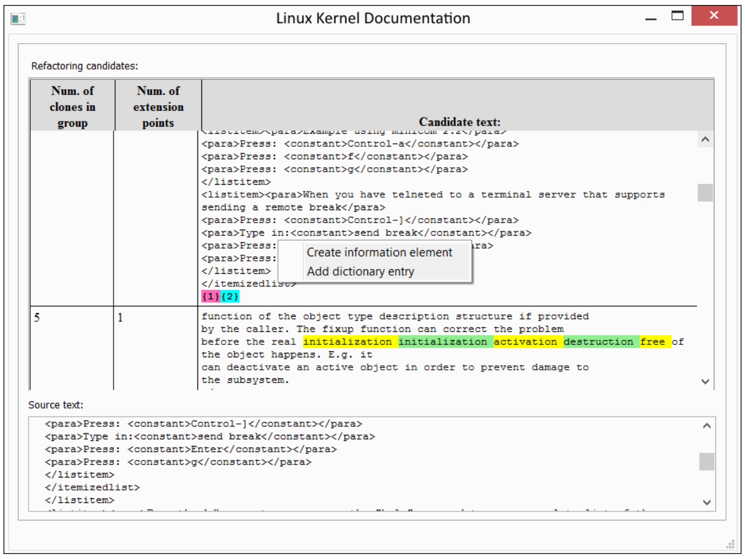
Before executing these operations, we check if the selected clone group intersects with other clone groups, which have already been used to extract information elements/dictionaries. Clone Miner allows intersection of clone groups, as it has no information what will happen to the detected clones further. But in our case, such intersection leads to mistakes in refactoring operations.

If this checking was successful we perform transformation of selected text fragments to be reused and the remaining context into correct XML. As mentioned earlier, Clone Miner outputs XML-incorrect results, but DocLine can operate only with the correct DocBook/DRL fragments. Generally speaking, our algorithm opens/closes all the necessary tags, both in the clone and in the context from which it is extracted. However, these open/close actions should be "clever". For example, if we close and reopen the tag `<para>` (it marks a new paragraph), then we would have two paragraphs instead of one in the resulting text (i.e. text on pure DocBook that is produced after the elimination of DRL constructions and, in particular, after the substitution of reusable information elements). This is the direction of the future work.

Once the refactoring operation for selected candidates is successfully completed, it is removed from L, and document coordinates of the another elements of L are recalculated. After that the user goes back to the "Analysis of pairs & clone groups" step.

5 The Tool

To support the process presented above we implemented a Documentation Refactoring Toolkit [25], and integrated it into DocLine/Eclipse. The tool is implemented in Python and can be invoked as a standalone application (i.e. outside of Eclipse and DocLine). The tool provides navigation over detected candidates, and text browsing facility to observe clones in the source text. It is possible to perform extracting all clones from selected groups into reusable elements, i.e. perform refactoring.



Refactoring candidates:

Num. of clones in group	Num. of extension points	Candidate text:

		`<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:`
		`<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

The main window is shown on the fig. 3. The tool is launched for a document, while the title of the document is displayed as the title of the window. The lines of table in the section «Refactoring candidates» correspond to clone groups or pairs found for that document. In the pop up menu for a candidate the user can select a refactoring type – to create either an information element or a dictionary element. If the candidate is a pair, then variations are highlighted as yellow/green pieces of text in the «Candidate text» column.

The «Source text» section shows clones in the source document. If a selected candidate is a clone group then the user needs to select the number of the clone element in the group (see color numbers at the end of the first cell in the «Candidate text» column, fig. 3). If a candidate is a pair (see the second cell in the «Candidate text» column), then the user needs to select a certain pair by clicking on the corresponding variation. In either case the «Source text» window will display the clone pair of clones in the source document.

6 Evaluation

We did our experiments using hand-made tests and third party DocBook documentation of open source industrial projects. The list of projects and corresponding documentation is presented in table 1.

Project	Documentation	Acronym	Size
Linux Kernel is an open operating system kernel, which is basis for Linux operating system	«Linux Kernel Documentation» is designed for programmers who use Linux Kernel [14]	LKD	892KB
Zend Framework is an open source framework for developing web applications and services using PHP	«Zend PHP Framework documentation» is a programming guide [26]	Zend	2924KB
Subversion is a versioning and revision control system	«Version Control with Subversion For Subversion 1.7» is a tool description for users and system administrators [27]	SVN	1810KB
DocBook is a framework for single source documentation development	«DocBook 4 Definitive Guide» is the complete official documentation on DocBook markup language 4.0 [28]	DocBook	686KB

Table 1. Documentation used in experiments

Following GQM approach [29] we selected a set of questions to characterize the way of the assessment in our experiments:

– question 1: quality of documentation clone detection
– question 2: effectiveness of filtering clones
– question 3: evaluation of refactoring facilities

Addressing **question 1** we did experiments with Clone Miner and DocLine clone search facilities. The first experiment was carried out on hand-made tests for which we know exactly the number and the locations of clones. We found that Clone Miner made some mistakes, e.g., it sometimes skipped the last token in clones. We fixed these errors. After that our tool found correctly all the clones in hand-made tests.

To assess **question 2** we used third party documentation listed in table 1. We used metrics, which are filtering types described in section 4.4. The results are presented in table 2. It should be noted, that filtering decreases the number of candidates by 13.2% on average.

Metrics	LKD	Zend	SVN	DocBook	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 2. Filtering results

Numbers of refactoring candidates after filtration are presented in table 3 for two cases: with the minimal lengths of clones of 1 and 5 (divided by slash in table). In the latter case the numbers of candidates are fewer and the situation looks more operable. However smaller clones, which were excluded in this case, can be used as dictionary elements or in other important situations. Therefore, we recommend that technical writer should work with candidates with the minimal length of clone equal to 1. To simplify operations with a large number of candidates our tool supports ordering by length.

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

Table 3. Number of candidates in case of minimal length of clone is 1 and 5

Let us consider **question 3**. We assessed the question using the metric called amount of reuse, which tracks percentages of reused text [30]. We calculate the metric by dividing the amount of reusable text by the total size of documentation. We take all refactoring candidates as reusable text and calculate the amount as $\sum_{C \in (\text{all candidates})} length(C)$, where $length(C)$ is the number of symbols in a clone or a clone pair multiplied by the number of clones in the group (see section 4.5). We measure documentation/text fragments size in symbols. It would be better to measure it in tokens but we had some technical problems with that. The average amount of reusable text for all tested documents is between 48% and 52.9%. The results show that when refactoring is carried out in an automatic (straightforward) way, reuse happens to be quite significant. But it is hard to estimate real reuse amount because, as it has been mentioned, technical writer performs additional semantic filtering of candidates for refactoring. To estimate the quality of refactoring more precisely, additional experiments with real project documentation are necessary.

Conclusions

Our experiments have shown that even after filtering we have a lot of insignificant clones. Some of them are easy to remove with improved filtering, but others can only be filtered manually. The precision of the algorithm is a baseline for our future work. Support of adaptive reuse should be also extended, e.g. proving extraction of information elements with n extension points, where $n > 1$.

During our experiments, it became clear that our tool should be improved to be more convenient in operating with clone groups, e.g. providing more facilities for construction of information elements.

The proposed approach can be useful in software product line documentation management environment to extract reusable document fragments for documentation of different product line members and organize reusable document structure. It simplifies document maintenance process and, of course, it is meaningful only if maintenance (product line member or/and its documentation) is significant. Our approach can also be used in the context of variability management [31] in software product line development.

Supporting semantic reuse can allow to integrate our approach with various software traceability techniques [32, 33], and mapping document fragments into other software artifacts: code, requirements, model entities, etc. In this case, reuse can improve the quality of this mapping, and semantic-oriented adaptive reuse could increase the granularity of the mapping.

Apart from software engineering, the proposed approach could also be used in such areas as Ontology Engineering [34] or Enterprise Architecture Modeling [35]: usually, models are stored in XML format, and irregular repetitions are also possible here, taking into account that a number of analysts can work with a large volume of information, and a lot of information is unstructured (documents and comments applied to models, long names of model entities, etc.).

References

1. Garousi, G., Garousi, V., Moussavi, M., Ruhe, G., Smith, B.: Evaluating usage and quality of technical software documentation: an empirical study. In Proceedings of EASE '13, pp. 24–35 (2013)
2. Watson, R.: Developing best practices for API reference documentation: Creating a platform to study how programmers learn new APIs. In Proceedings of IPCC'12, pp. 1–9 (2012)
3. Parnas, D. L. Precise Documentation: The Key To Better Software: The Future of Software Engineering, S. Nanz, Ed.: Springer, (2011)
4. Holmes, R., Walker, R. J.: Systematizing Pragmatic Software Reuse. ACM Transactions on Software Engineering and Methodology, vol. 21(4), 20, 44 p. (2013)
5. Czarnecki, K.: Software Reuse and Evolution with Generative Techniques. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, p. 575 (2007)
6. Jarzabek, S., Bassett, P., Zhang H., Zhang, W.: XVCL: XML-based Variant Configuration Language. ICSE 2003, pp. 810–811 (2003)
7. Bassett, P.: The Theory and Practice of Adaptive Reuse. SIGSOFT Software Engineering Notes, 22(3), pp. 2–9 (1997)
8. Koznov, D., Romanovsky, K.: DocLine: A Method for Software Product Lines Documentation Development. Programming and Computer Software, 34(4), pp. 216–224 (2008)
9. Romanovsky, K., Koznov, D., Minchin, L.: Refactoring the Documentation of Software Product Lines. CEE-SET 2008, Brno (Czech Republic), October 13–15, 2008. LNCS, vol. 4980, Springer 2011, pp. 158–170 (2011)
10. Akhin, M., Itsykson, V.: Clone Detection: Why, What and How? In Proceedings of CEE-SECR'10, pp. 36–42 (2010)
11. Rattan D., Bhatia, R. K., Singh, M.: Software Clone Detection: A Systematic Review. Information & Software Technology (INFSOF), 55(7), pp. 1165–1199 (2013)
12. Walsh, N., Muellner, L.: DocBook: The Definitive Guide. O'Reilly, 1999, 644 p. (1999)
13. Basit, H. A., Smyth, W. F., Puglisi, S. J., Turpin, A., and Jarzabek, S.: Efficient Token Based Clone Detection with Flexible Tokenization. In Proceedings of ACM

SIGSOFT International Symposium on the Foundations of Software Engineering, ACM Press, pp. 513–516 (2007)
14. Linux Kernel Documentation, snapshot on Dec 11, 2013 (2013), `https://github.com/torvalds/linux/tree/master/Documentation/DocBook/`
15. Darwin Information Typing Architecture (DITA) Version 1.2 Specification (2012), `http://docs.oasis-open.org/dita/v1.2/os/spec/DITA1.2-spec.pdf`
16. 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. of Systems and Software, Under Review, (2012).
17. Zhong, H., Zhang, L., Xie, T., Mei, H.: Inferring resource specifications from natural language API documentation. In Proceedings of 24th ASE, pp. 307–318 (2009)
18. Zhong, H., Su, Z.: Detecting API documentation errors. In Proceedings of SPASH/OOPSLA, pp. 803–816 (2013)
19. Wingkvist, A., Lowe, W., Ericsson, M., Lincke, R.: Analysis and visualization of information quality of technical documentation. In Proceedings of the 4th European Conference on Information Management and Evaluation, pp. 388–396 (2010)
20. Wingkvist, A., Ericsson, M., Lowe, W.: A Visualization-based Approach to Present and Assess Technical Documentation Quality. Electronic Journal of Information Systems Evaluation, 14 (1) (2011)
21. VizzAnalyzer Clone Detection Tool, `http://www.arisa.se/vizz_analyzer.php`
22. Cameron, H. G.: Wright: Technical Writing Tools for Engineers and Scientists. Computing in Science and Engineering, 12(5), pp.98–103 (2010)
23. Grigorev, S., Kirilenko, I.: GLR-based abstract parsing. In Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia (2013)
24. Fowler, M., et al.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
25. Document Refactoring Toolkit, `http://www.math.spbu.ru/user/kromanovsky/docline/index_en.html`
26. Zend PHP Framework documentation, snapshot on Apr 24, 2015 (2015), `https://github.com/zendframework/zf1/tree/master/documentation`
27. SVN Book, snapshot on Apr 24, 2015 (2015), `http://sourceforge.net/p/svnbook/source/HEAD/tree/trunk/en/book/`
28. DocBook Definitive Guide, snapshot on Apr 24, 2015 (2015), `http://sourceforge.net/p/docbook/code/HEAD/tree/trunk/defguide/en/`
29. Basili, V. R., Caldiera, G., Rombach H. D.: The Goal Question Metric Approach. Encyclopedia of Software Engineering: Wiley (1994)
30. Frakes, W., Terry., C.: Software reuse: metrics and models. ACM Comput. Surv., 28(2), pp. 415–435 (1996)
31. Krueger, C. W.: Variation Management for Software Product Lines. In Proceedings of SPL'02, San Diego, CA, USA pp. 37–48 (2002)
32. Abadi, A., Nisenson, M., Simionovici, Y.: A Traceability Technique for Specifications. In Proceedings of ICPC'08, pp. 103–112 (2008)
33. Terekhov, A. N., Sokolov, V. V.: Document Implementation of the conformation of MSC and SDL diagrams in the REAL technology. Programming and Computer Software. 33 (1), pp. 24–33 (2007)
34. Gavrilova, T. A.: Ontological engineering for practical knowledge work. 11th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, KES 2007, LNCS Vol.4693, pp. 1154–1161 (2007)
35. Grigoriev, L., Kudryavtsev, D.: ORG-Master: Combining Classifications, Matrices and Diagrams in the Enterprise Architecture Modeling Tool. Communications in Computer and Information Science (CCIS) Series, Springer, pp. 250–258 (2013)