Journal of Computer Science 10 (5): 755-762, 2014
ISSN: 1549-3636
© 2014 Science Publications
doi:10.3844/jcssp.2014.755.762 Published Online 10 (5) 2014 (http://www.thescipub.com/jcs.toc)

COMENTE+: A TOOL FOR IMPROVING SOURCE CODE DOCUMENTATION USING INFORMATION RETRIEVAL

[1]Julio Cezar Zanoni, [1]Milton Pires Ramos, [2]Cesar Augusto Tacla, [2]Gilson Yukio Sato, [3]Gregory Moro Puppi Wanderley and [3]Emerson Cabrera Paraiso

[1]TECPAR-Paraná Institute of Technology, Curitiba, Brazil
[2]CPGEI, UTFPR-Technological Federal University of Paraná, Curitiba, Brazil
[3]PPGIa, PUCPR-Pontifícia Universidade Católica do Paraná, Curitiba, Brazil

Received 2013-04-17; Revised 2013-05-23; Accepted 2014-01-4

ABSTRACT

Document source code is seen as a boring time consuming task by several developers. However, a well-documented source code, allow developers to have a better visibility into what was and is being developed, helping, for example, the reuse of the code. This study presents a semi-automatic method for documentation of source code from the existing artifacts in a software project under development. The method aims to reduce developer's workload, allowing them to work on other tasks of the project and/or ensure that the project deadlines will be met. The method, implemented in a tool, called Comente+, is capable of creating or updating comments into a source code from gathered information recovered from the project artifacts. To implement Comente+, we used an information retrieval approach. We performed some experiments with real data to validate this approach. For that, we created a special measure that estimates how well documented a source code is.

Keywords: Information Retrieval, Source Code Documentation, Small Teams

1. INTRODUCTION

The source code documentation is a valuable tool to detect and correct problems in software systems. With the support of a well-structured and organized documentation, a reduction in time spent to make changes or maintenance in a source code is expected. Paduelli and Sanchez (2006), argue that the difficult to maintain legacy systems, due to its complexity and size, is aggravated by the staff turnover and also, by an insufficient or nonexistent documentation. They also mentioned that in developing or modifying a source code, the developers do not produce this documentation in an appropriate manner, writing only brief notes of commentary without much meaning.

Indeed, the unwillingness of developers to document the source code exists and is probably related to the fact that this task requires the production of several pages of explanatory text. Write these texts lead them to stop developing codes to dedicate themselves to something to which they have difficulty in making or low interest.

The source code documentation is also important since it can give visibility to what was and is being developed by the participants of the development group. It may be used as a tool for source code reuse or provide information about development state.

In this research, we are interested in studying how code documentation may be increased, especially for projects been developed by small teams (teams with up to 10 members). In a small team, many times small companies with small budgets, the task of code documentation may be one among several tasks that a team member must perform.

Thus, it is helpful reduce the team members' workload caused by documentation by making the process more automatized as possible. In small teams, those who are involved in programming also perform activities such as company management, contact with costumers.

Corresponding Author: Julio Cezar Zanoni, TECPAR-Paraná Institute of Technology, Curitiba, Brazil

In this study we present a semi-automatic method for documentation of source code from the existing artifactsin a software project under development. The method, implemented in a tool called Comente+, is capable of evaluating the degree of documentation of a code and update it with information gathered from project artifacts as well as the source code itself.

This article is divided as follows. In section 2, we define the process of developing software in a small team and the process of documenting source code. In section 2, we also present our approach for source code documentation. In section 3 we show some results obtained from practical experiments. Section 4 presents some discussion. Finally, in section 5, we present our conclusions and suggestions for future work.

2. MATERIALS AND METHODS

The next paragraphs present some background needed to understand our approach.

2.1. Software Development in Small Teams

The software development comprises several steps usually supported by tools designed for that purpose. In general, it is produced collaboratively, with participation of several specialists. Researches in Computer Supported Cooperative Work (CSCW) applied to software development are widespread. Several studies have already been developed (Cook and Chumber, 2005; Teruel *et al*., 2012; Jiang *et al*., 2006; Duque *et al*., 2012). However, most of this work focuses two aspects: (1) improvements to the infrastructure to support distributed development (integrated environments or groupware) and (2) encourage the communication among participants of the collaborative project, aimed at large teams of software development.

Software development small teams have some needs and characteristics that should be taken into account (Campagnolo *et al*., 2009). As in general participants work on a common physical environment, the face to face communication is enough, which means that electronic messages systems are less important. Participants are in charge of specific activities, but they develop different activities during the project life cycle, i.e., in practice a participant plays different roles in spite of his formal function. For example, a developer can play the role of analyst and tester. Such a multiplicity of roles can quickly lead to an excessive workload; this fact contributes to members neglecting important activities such as project and software documentation. The source code documentation is a crucial task to facilitate reuse and software maintenance.

The documentation process may be more demanding if the project has to comply with standards such as ISO 9001 or models of maturity like the Capability Maturity Model (CMM) (Soomro and Hesson, 2012). Being in compliance does not necessarily mean that the group must be certified, but that it aims at ensuring the quality of software through the definition and standardization of development processes. All requirements imposed by rules or models, have significant impact on small teams. There are researchers such as Pollice *et al*. (2004); Land and Walz (2006) and Campagnolo *et al*. (2009) that propose approaches to minimize such an impact.

To Land and Walz (2006), small development teams have up to 20 participants, unlike Pollice and colleagues and Campagnolo and colleagues that consider up to 10 participants. As a basis for this work, a small team has up to 10 participants.

The method presented in section 2.6 is intended to gap some of the features previously mentioned. One of them and perhaps the most important, since it refers to the main research problem is the participants' excessive workload. The method aims to reduce this workload, allowing the developer to work on other tasks of the project and/or ensure that the project deadlines will be met.

It is important to highlight that this method and the tool associated to it are not intended to substitute programs to generate source code documentation like Javadoc. The main idea is to provide developers with indications on what is important to be documented, promoting a quality increase on source code comments.

2.2. Source Code Documentation

The software development tools, such as UML language and RUP process generate several documents, called artifacts, used in various stages of the process as a way to record its development (Shiki *et al*., 2004; Massoni *et al*., 2003; OMG, 2010). These documents are elaborated and refined during the project development until the delivery of the product to customers.

In general, there are two types of software documentation: Management documentation and user documentation. The management documentation includes all information about the project development, including the source code documentation. The user documentation is focused to the end user and normally is compounded by user's manuals.

The documentation of a source code is done by inserting comments into it. A comment is a fraction of code text identified by the compiler, but completely ignored by it since it does not represent a valid action in terms of code compilation. Moreover, these comments (**Fig. 1**) should be useful to the human reader and must contain at least the explanation of the code as a whole.

Figure 2 shows an example of a source code well documented, in which comments describe a part of a class. In this example, the developer uses the comments intensively, producing a source code easier to maintain.

The implementation of the method was possible thanks some techniques related to information retrieval and pattern matching. The next section briefly presents them.

2.3. Information Retrieval and Pattern Matching

This section briefly presents the techniques used in the method implementation.

2.4. Information Retrieval

Information Retrieval (IR) provides to users a set of possible documents that match to the terms of the search expression used to represent users' needs (Baeza-Yates and Ribeiro-Neto, 1999). IR can be used to search information on unstructured or structured texts. Unstructured texts are usually those free of any structure, like a user review, a letter (Barathi and Valli, 2011). Structured or semi-structured texts follow a standard format or pattern, as is the case of source code.

Another way of using IR is recovering passages rather than full texts (Callan, 1996). Passages are small pieces of a text. These pieces of texts could be indexed by an IR system. The users' searches will return as response such small portions of text that are more significant and sometimes could answer directly to the user needs.

Concerning our approach, the use of IR is directly related to retrieving passages information from the texts written in natural language. Such information will be used to complete comments in the source code.

2.5. Pattern Matching

Analyzing the source code shown in **Fig. 2**, some interesting information can be retrieved with the use of pattern matching techniques.

Pattern matching is another research domain concerned with the formulation of queries and searches based on a pattern. It allows the retrieval of words or parts of a text that have certain properties. A pattern is a set of syntactic features that must occur in a text segment. The text segments that meet the specifications of the pattern are called "matched". The patterns range from words to more complex structures. In order to recover those patterns one need well-formed rules (like regular expressions).

The most common types of patterns are: Words, prefixes and suffixes. The most basic patterns are words. Matching these patterns means finding the exact string with the word/pattern in the text been analyzed.

Among the existent pattern matching types, Regular Expressions (RE or regex) are the most powerful. Regex provide a flexible and efficient mechanism for processing texts, which uses a formal method to specify a text pattern. Through an extensive and rather complete notation, it is possible to analyze a large set of texts looking for patterns.

A regular expression describes a set of strings, concisely, without having to list all elements of the set. For example, a set containing the strings "Händel", "Handel" and "Haendel" can be described by the pattern H(ä|ae?)ndel. These constructs can be combined to form arbitrarily complex expressions as well as arithmetic expressions. In general, there are different regular expressions to describe a set of strings. Many of the Integrated Development Environments (IDEs) implement regular expressions using syntaxes that are similar to the ones found in programming languages. The exact syntax of a regular expression and the available operators vary according to the adopted implementation.

The regular expressions were used to extract the main elements from the code, such as: Class definition, method signature and variables declaration. Also, we used them to find existing comments in the source code.

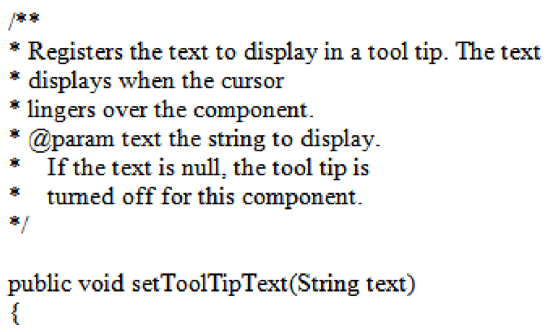
The next section present the method for semi-automatic source code documentation, based on pattern matching and IR.

2.6. A Method for a Semi-Automatic Source Code Documentation

In this section we describe the proposed method for source code documentation. It performs the extraction and analysis of information from source code and from the management documentation related to a specific software project.

The proposed method gathers information from the source code comparing it with comments and descriptions contained in every artifact related to the project (those written in natural language). In doing so, it is possible to check what can be automatically documented.

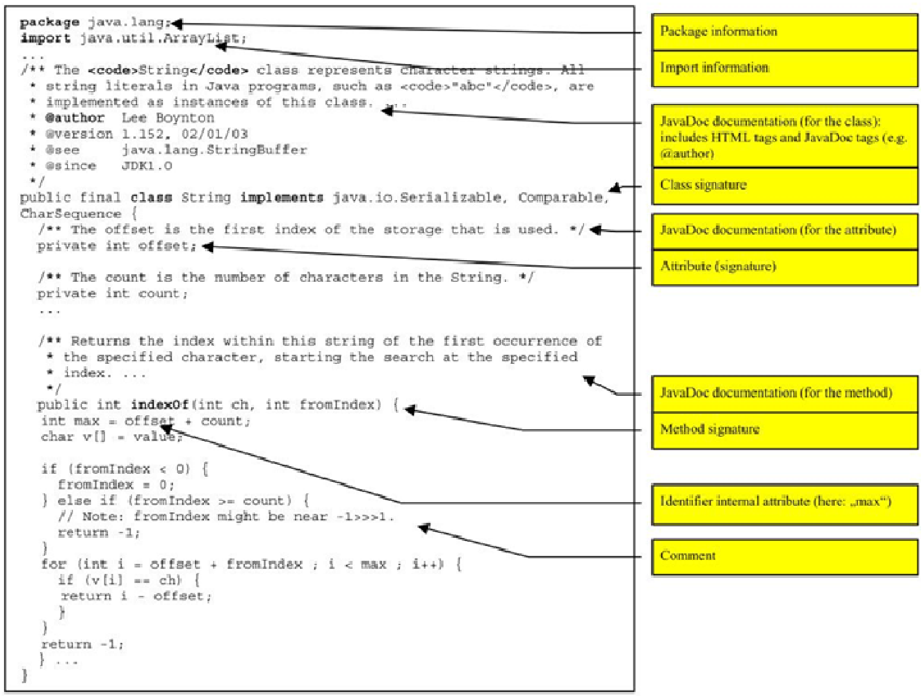
Figure 3 shows a diagram identifying the three main modules of the method.



```
/**
* Registers the text to display in a tool tip. The text
* displays when the cursor
* lingers over the component.
* @param text the string to display.
*    If the text is null, the tool tip is
*    turned off for this component.
*/

public void setToolTipText(String text)
{
```

Fig. 1. A fragment of Java code with some comments in Javadoc format (Javadoc, 2013)



```
package java.lang;
import java.util.ArrayList;
...
/** The <code>String</code> class represents character strings. All
 * string literals in Java programs, such as <code>"abc"</code>, are
 * implemented as instances of this class. ...
 * @author  Lee Boynton
 * @version 1.152, 02/01/03
 * @see     java.lang.StringBuffer
 * @since   JDK1.0
 */
public final class String implements java.io.Serializable, Comparable,
CharSequence {
  /** The offset is the first index of the storage that is used. */
  private int offset;

  /** The count is the number of characters in the String. */
  private int count;
  ...

  /** Returns the index within this string of the first occurrence of
   * the specified character, starting the search at the specified
   * index. ...
   */
  public int indexOf(int ch, int fromIndex) {
   int max = offset + count;
   char v[] = value;

   if (fromIndex < 0) {
     fromIndex = 0;
   } else if (fromIndex >= count) {
     // Note: fromIndex might be near -1>>>1.
     return -1;
   }
   for (int i = offset + fromIndex ; i < max ; i++) {
     if (v[i] == ch) {
      return i - offset;
     }
   }
   return -1;
  } ...
}
```

Package information
Import information
JavaDoc documentation (for the class): includes HTML tags and JavaDoc tags (e.g. @author)
Class signature
JavaDoc documentation (for the attribute)
Attribute (signature)
JavaDoc documentation (for the method)
Method signature
Identifier internal attribute (here: „max")
Comment

Fig. 2. Java Code (String.java from standard Java library) (Rech, 2005)

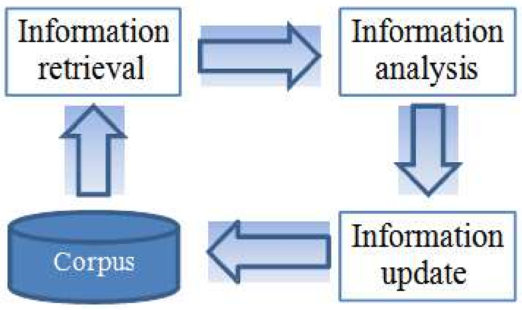


Fig. 3. The main method modules

The corpus is a predefined set of files to be read and updated. The corpus contains all the existing software documentation for a particular development project. Thus, it contains the project description files and the source code files written in Java (Java-Net, 2010).

2.7. Information Retrieval Module

The Information Retrieval module processes the files into the corpus. It first analyzes the source code files. This is due to the fact that in these files the code lines (except the comments) are used as queries when searching into documents written in natural language.

It means that the code lines written by developers are the main source for retrieving information, containing all the relevant information that we wish being documented by means of comments. For instance: A class name or a method signature (arguments, visibility,) is used to search relevant passages into artifacts related to the code. In section 3, we present some experiments that evaluated the effectiveness of this approach.

The process to extract information from the source code uses pattern matching by applying regular expressions taking into account that the source code is a well-structured document and its domain is well known. The programming language (Java in this case) has a grammar and a set of reserved words (or keywords according to the Java specification (Gosling, 2000) that can be used by the regex to easily match with its structures (language syntax) and extract information from them.

Files written in natural language (no matter what language is used) need to be preprocessed, in order to split them into passages. A passage in this case is a sentence. Figures, tables and diagrams are not used in the actual version of the system. After preprocessing, the passages are indexed by the IR system, written using (Lucene, 2011). Lucene provides all tools to indexing the passages and recovering them, using information from source codes as search terms.

Once the relevant information was found, it is stored in a MySQL database.

2.8. Information Analysis Module

The information obtained in the previous process is confronted in order to check which element (class, methods and variables) of the code is documented. In such a process, three cases may occur:

- The element of the source code is not documented
- The element is partially documented. In this case, there is an associated comment to the source code structure, but it not contains all the possible relevant information
- The element is well documented, in terms of information just gathered from the source code itself

In the first case, if there is no comment, then a new one is created, including all the relevant information founded in the source code itself (e.g., method parameters and method return type) and, if exists, it is incremented with passages found in the documents written in natural language (i.e., software requirements, software architecture,).

In the case a comment exists and its information is incomplete, in terms of source code information (e.g. method parameters and method return type) a new comment is created instead of deleting the existing one. This new comment is intended to show the missing information to the developer, that should validate de new one, deleting the old one (if he agrees with the new one):

- Finally, in the case a comment exists and it is complete, just the recovered passages are attached to it
- In all cases, the comment is created using the Javadoc's format
- It is important to highlight that a source code may have more them a comment for each element

2.9. Information Update Module

This last module finally updates the documentation, writing the comments into the source code.

2.10. The Comente+Implementation

The method just described was implemented in Java, generating a tool called Comente+. The Comente+ tool analyses every Java file of a project before and after the method application. In order to estimate how documented a source code is, we defined a C (for Comments) measure, according to the Equation (1):

$$C = \frac{\text{total of comments}}{\#\text{of classes} + \#\text{of methdods} + \#\text{of variables}} \quad (1)$$

where, total of comments is the sum of existing comments in the source code, # of classes is the sum of classes found by Comente+, # of methods is the sum of methods found by Comente+ and # of variables is the sum of variables found by Comente+.

In the actual version of the system, every Java file is copied in an auxiliary folder before Comente+ starts processing them. At the end, we have a set of modified Java files, enriched with new or updated comments.

It is important to highlight that the existence of a large number of comments does not mean that their quality is good enough to adequately document the source code. In section 3, we present some qualitative evaluation of this point.

In the next section we present the results of some practical experimentation we performed.

3. RESULTS

To evaluate Comente+ (and consequently the approach) a few experiments were carried out, intended to demonstrate its effectiveness. The experiments were performed using three different software projects. In each corpus there were source code files written in Java and text files written in Portuguese describing the system.

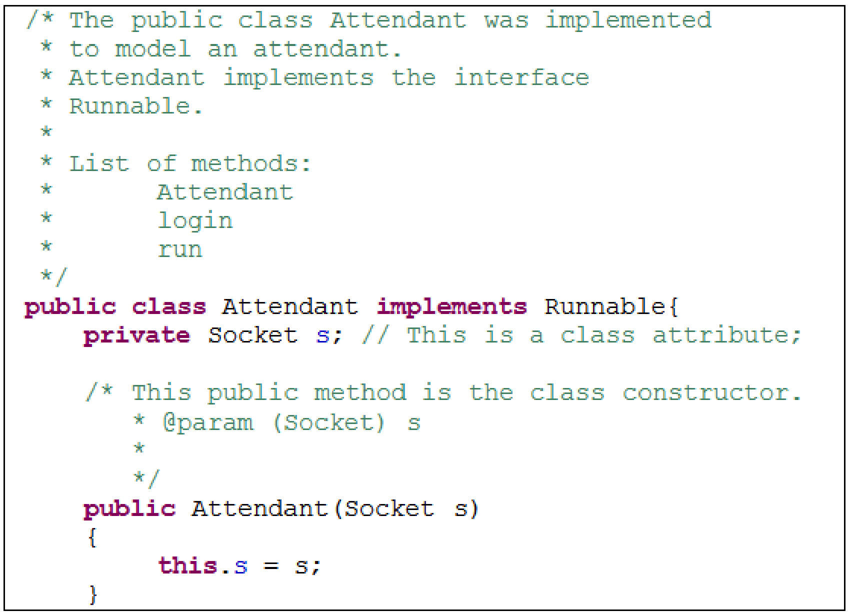
The first project, called SE Telecom, is focused on telecommunications and was developed by a small team of a Brazilian company. The other two projects (Emotion and MODUS-SD) are related to Human Computer Interaction and were developed by a small research team at Pontifícia Universidade Católica do Paraná (PUCPR) in Brazil. **Table 1** shows the main features of the corpora.

Table 2 shows the number of comments found before Comente+ processing and the number of new ones added or updated to source code files. It also shows the C measure calculated before and after Comente+ processing.

We also performed a qualitative study, asking developers to evaluate the passages recovered from text files and, consequently, the comments produced with them. Developers should classify each passage according to these three possibilities:

- No relation: The passage has no relation with the element been documented
- Some relation: The passage has some relation with the element been documented. This could happen if a passage has, for instance, information related to more than one element in the code
- Total relation: The passage is definitively related to the element been documented

The results of this qualitative evaluation, for project SE Telecom, are presented in **Table 3 and 4**.



```
/* The public class Attendant was implemented
 * to model an attendant.
 * Attendant implements the interface
 * Runnable.
 *
 * List of methods:
 *       Attendant
 *       login
 *       run
 */
public class Attendant implements Runnable{
    private Socket s; // This is a class attribute;

    /* This public method is the class constructor.
       * @param (Socket) s
       *
       */
    public Attendant(Socket s)
    {
        this.s = s;
    }
```

Fig. 4. An excerpt of a source code example from SE Telecom

Table 1. The corpora used in the experiments

Project	# of source code files	# of classes	# of methods	# of variables	# of documentation pages (after preprocessing)
SE telecom	24	24	142	335	7
Emotion	1	1	4	21	53
MODUS-SD	7	7	15	328	11

Table 2. Comente+ quantitative results

Corpus	# of comments written by developers	# of comments created (or updated) by Comente+	C Before Comente+	C After Comente+
SE Telecom	341	501	0,681	1,681
Emotion	85	25	3,269	4,231
MODUS-SD	327	350	0,934	1,934

Table 3. Qualitative evaluation: Retrieved passages using the AND operator

Project: SE Telecom	No relation	Some relation	Total relation
# of recovered passages = 48	7	19	22
% comparing with total recovered results	15%	39%	46%

Table 4. Qualitative evaluation: Retrieved passages using the search terms

Project: SE Telecom	No relation	Some relation	Total relation
# of recovered passages = 143	72	11	60
% comparing with total recovered results	50%	8%	42%

Figure 4 shows an extract of source code commented by Comente+. In this case, the information needed to compose the comments was mainly found in the source code itself.

4. DISCUSSION

Table 2 shows the number of comments found before Comente+ processing and the number of new ones added or updated to source code files. It also shows the C measure calculated before and after Comente+ processing. The number of comments is dramatically augmented for some projects. This is due to the fact that Comente+ retrieves passages and creates or updates a comment for each one.

The project Emotion has the best score in terms of comments. Only 25 new comments were added. This project has the highest C, before and after Comente+ processing. This is due the fact that the source code has a few number of code elements if compared to the number of comments. We remind again that each element in the code may have more than one comment.

We tested two different approaches for recovering passages using Lucene. The first one uses the AND operator among the search elements when recovering passages (**Table 3**). The second approach used every element in the query to search a passage, producing a greater number of recovered passages (**Table 4**). As expected, the first approach produced better results (reducing the number of false positives-no relation), since recovered passages have in their content information about all elements of the query. Adding “some relation” and “total relation” we have 85% of passages related to the element been documented. Almost 50% of the total passages recovered were classified as been completely related to the element been documented.

5. CONCLUSION

This study presents a method for documenting source code based on information recovered in the artifacts produced during software development. The results showed that the Comente+ is a promising tool in documenting source code. Comments in the source code are created or updated according to passages found in natural language texts.

We are planning to apply Comente+ since the beginning of a real project in order to collect some data to evaluate if, or not, the workload over developers was reduced.

We also planned to create an instigator agent (as the one presented in (Boz *et al*., 2011)) that will help developers to better document their code, giving insights and suggestion during codification time.

6. REFERENCES

Barathi, M. and S. Valli, 2011. Context disambiguation based semantic web search for effective information retrieval. J. Comput. Sci., 7: 548-553. DOI: 10.3844/jcssp.2011.548.553

Baeza-Yates, R. and B. Ribeiro-Neto, 1999. Modern Information Retrieval. 1st Edn., Addison-Wesley, ISBN-10: 020139829X, pp: 544.

Boz, J.G., M.P. Ramos, G.Y. Sato, C.A. Tacla and J.C. Nievola *et al*., 2011. A virtual catalyst in the knowledge acquisition process. Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, (EKE' 11), Miami, EUA., pp: 149-152.

Callan, J.P., 1996. Passage-level evidence in document retrieval. Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Jul. 3-6, ACM Press, Dublin, Ireland, pp: 302-310.

Campagnolo, B., C.A. Tacla, E.C. Paraiso, G. Sato and M.P. Ramos, 2009. An architecture for supporting small collocated teams in cooperative software development. Proceedigns of the 13th International Conference on Computer Supported Cooperative Work in Design, Apr. 22-24, IEEE Xplore Press, Santiago, pp: 264-269. DOI: 10.1109/CSCWD.2009.4968069

Cook, C. and N. Churcher, 2005. Modelling and measuring collaborative software engineering. Proceedings of the 28th Australasian Computer Science Conference, Research and Practice in Information Technology, (PIT' 05), Australia, pp: 267-276.

Duque, R., M.L. Rodriguez, M.V. Hurtado, C. Bravo and C. Rodriguez-Dominguez, 2012. Integration of collaboration and interaction analysis mechanisms in a concern-based architecture for groupware systems. Sci. Comput. Programm., 77: 29-45. DOI: 10.1016/j.scico.2010.05.003

Gosling, J., 2000. The Java Language Specification. 1st Edn., Addison-Wesley Professional, Boston, ISBN-10: 0201310082, pp: 505.

Javadoc, 2013. The java API documentation generator.

Java-Net, 2010. A brief history of the green project.

Jiang, T., J. Ying, M. Wu and M. Fang, 2006. An architecture of process-centered context-aware software development environment. Proceedings of the 10th International Conference on Computer Supported Cooperative Work in Design, May 3-5, IEEE Xplore Press, Nanjing, pp: 1-5. DOI: 10.1109/CSCWD.2006.253193

Land, S.K. and J.W. Walz, 2006. Practical Support for ISO 9001 Software Project Documentation. 1st Edn., IEEE Computer Society, New Jersey, ISBN-10: 0471768677, pp: 418.

Lucene, 2011. Welcome to apache lucene.

Massoni, T., A. Sampaio, P. Borba and A.L. Freire, 2003. A RUP-Based Software Process Supporting Progressive Implementation. 1st Edn., IGI Publishing, Hershey, EUA, pp: 13.

OMG, 2010. Unified Modeling Language (UML), version 2.0.

Paduelli, M.M. and R. Sanches, 2006. Maintenance problems: Characterization and evolution. Proceedings of the 3rd Workshop on Modern Software Maintenance, V Brazilian Symposium on Software Quality, (SQ '06), Vila Velha, Brazil, pp: 1-13.

Pollice, G., L. Augustine, C. Lowe and J. Madhur, 2004. Software Development for Small Teams: A RUP-Centric Approach. 1st Edn., Addison-Wesley, ISBN-10: 0321199502, pp: 272.

Rech, J., 2005. Preprocessing of object-oriented source code for code retrieval. Citeseer.

Shiki, N., Y. Ohno, A. Fujii, T. Murata and Y. Matsumura, 2004. Unified Modeling Language (UML) for hospital-based cancer registration processes. Asian Pac. J. Cancer Prev., 9: 789-96. PMID: 19256778

Soomro, T.R. and Hesson, 2012. Mihyar. Supporting best practices and standards for information technology infrastructure library. J. Comput. Sci., 8: 272-276. DOI: DOI: 10.3844/jcssp.2012.272.276

Teruel, M. A., E. Navarro, V. Lopez-Jaquero, F. Montero and J. Jaen *et al*., 2012. Analyzing the understandability of requirements engineering languages for CSCW systems: A family of experiments. Inform. Softw. Technol., 54: 1215-1228. DOI: 10.1016/j.infsof.2012.06.001