Nigerian Journal of Technology (NIJOTECH)
Vol. 34 No. 2, April 2015, pp. **392 – 398**

University of Nigeria, Nsukka, ISSN: 1115-8443
www.nijotech.com
http://dx.doi.org/10.4314/njt.v34i2.25

MINIMIZATION OF RETRIEVAL TIME DURING SOFTWARE REUSE

H. O. Salami*
DEPARTMENT OF COMPUTER SCIENCE, FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA, NIGERIA
E-mail address: *ho.salami@futminna.edu.ng*

ABSTRACT
Software reuse refers to the development of software using existing software. Reuse of software can help reduce software development time and overall cost. Retrieval of relevant software from the repository during software reuse can be time consuming if the repository contains many projects, and/or the retrieval process is computationally expensive. This paper describes pre-filtering, which is a method of minimizing retrieval time during software reuse. Pre-filtering can be applied while reusing object-oriented software, whose requirement specifications contain Unified Modelling Language (UML) diagrams. Pre-filtering involves quickly identifying a subset of repository projects which are potentially similar to a query model. The candidate projects are subsequently compared with the query during retrieval to determine their actual degree of similarity to the query. The query and repository projects are represented by n-dimensional feature vectors, where each feature is a metric which provides a quantitative measure of properties of a software project. Experimental results show that the proposed technique leads to a significant reduction in retrieval time, even though it causes a slight decrease in mean average precision.

Keywords: *software reuse; retrieval time; pre-filtering; software repository; UML*

1. INTRODUCTION
Software reuse is the use of existing software to develop new software. It minimizes *reinvention of the wheel* during software development. The benefits of software reuse include accelerated development, reduced overall costs and risks, increased dependability and effective use of specialists[1]. Reusable software artifacts are usually kept in a components library or repository, from where they can be retrieved during reuse. As the repository increases in size, there is a corresponding rise in retrieval time which can lower the expected savings in development time. Moreover, retrieval time can be high if the retrieval process is computationally expensive.

This paper describes pre-filtering, which is a fast way of identifying a subset of repository projects which are potentially similar to a query model. The shortlisted repository projects are subsequently compared with the query model in a retrieval stage to ascertain their actual degree of similarity with the query model. The pre-filtering technique proposed in this paper can be applied while reusingobject-oriented software, whose requirement specifications contain Unified Modelling Language (UML) diagrams. UML is the *de facto* standard language for modelling object-oriented software.

There are two ways of gaining speed by selecting a first set of projects from the repository prior to the retrieval stage during the *pre-filtering stage.* First, it leads to a reduction in retrieval time because not all repository projects are compared with the query when a retrieval stage is preceded by pre-filtering. The decrease in retrieval time is significant when the retrieval stage is time consuming and/or the repository contains many projects. Second, if the repository projects are indexed beforehand, additional speed is gained because it eliminates the need to load each repository project into primary memory during pre-filtering as well as retrieval. Indexing entails computing and saving the pre-filtering features for each project as the project is being stored in the repository.

The remainder of this paper is organized as follows: Section 2 briefly discusses related work. In Section 3, an overview of the software reuse process is provided. Section 4 describes the pre-filtering process.

*** Corresponding Author, Tel: +234-903-036-1832**

Experimental results are presented in Section 5 and the paper is concluded in Section 6.

2. RELATED WORK

Channarukul et al. [2] performed pre-filtering while retrieving class diagrams for reuse. During their pre-filtering stage, a fixed number of projects are selected from the repository depending on the number of class names the query and repository class diagrams have in common. However, because this filtering approach checks for exact matches in class names, two similar class diagrams containing classes whose names are non-identical but similar (e.g. college and school) may not be considered as similar.

While retrieving class diagrams for reuse, authors in [3] performed a computationally inexpensive selection (or pre-filtering),in which a fixed number of relevant cases are chosen by using the WordNet lexicon to index cases. WordNet is a large lexical database developed at Princeton University. Because WordNet is utilized during pre-filtering, repository class diagrams which contain classes whose names are similar in meaning to those of the query diagram are likely to be shortlisted.

Park and Bae [4]adopted a two-stage approach for retrieving repository artifacts. In the first stage (i.e. the pre-filtering stage), they determined the similarity score of two class diagrams using the Structure Mapping Engine (SME). The SME software works based on the structure mapping theory, which allows knowledge to be mapped in two domains by considering relational commonalities of objects in the domains regardless of the objects involved in the relationships. Based on the similarity scores obtained, a subset of repository projects is selected. During the second stage, sequence diagrams in the shortlisted projects are converted to Message-Object-Order-Graphs which are then compared using a graph matching algorithm.

The pre-filtering approach presented in this paper differs from those in existing works in two aspects. Firstly, this work compares software systems using software metrics that describe some properties of the software systems. Secondly, unlike the previous software reuse works which performed pre-filtering using information collected from only class diagrams, this work uses information from UML sequence diagrams and class diagrams during pre-filtering. Both class diagrams and sequence diagrams are commonly used to model software systems early in the software development life cycle.

3. REUSE SYSTEM

Our reuse approach is hinged on the intuition that similar software systems have similar requirements. Thus, the requirement specifications of new projects (software systems) to be built serve as queries that are compared with requirement specifications of existing projects stored in a repository. Once the most similar requirements are found, the corresponding artifacts can be adapted to meet the needs of the new software system. Moreover, since UML is the *de facto* language for modeling software requirements, the requirement specifications to be compared are described using UML. This section gives an overall picture of our reuse system by describing the sequence of steps needed to reuse software contained in a repository. More details about the reuse system can be found in [5]. The prerequisite for using the reuse system is that requirement specification for the query and repository projects should contain class and sequence diagrams. Figure 1 illustrates the steps involved in the reuse process. These steps are described below:

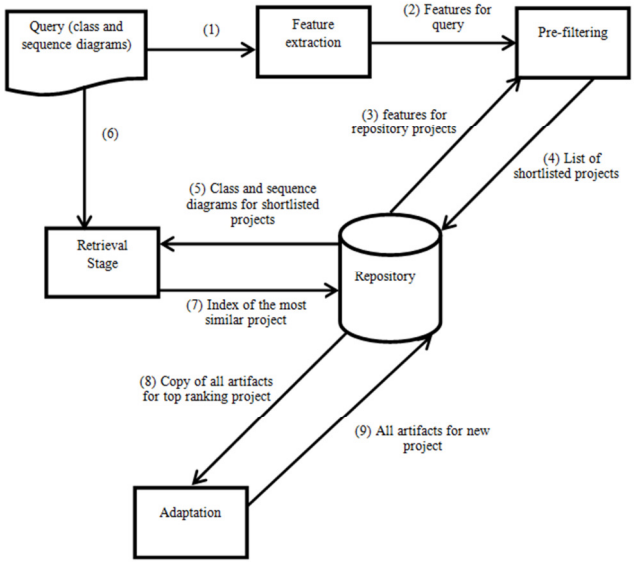

Figure 1: Concept of operation of reuse system

1. The user presents a query requirement specification which contains class and sequence diagrams.
2. The pre-filtering features for the query are computed. These features consist of size and complexity metrics such as total number of classes in a class diagram, number of messages exchanged by objects in a sequence diagram, and the number of attributes and operations of classes. These metrics can be used to filter out repository projects whose sizes are significantly different from that of the new system.

3. Pre-computed features for each project in the repository are retrieved. By comparing the features of query and repository projects, a tentative similarity score between the query and each repository project is obtained.
4. Based on the initial similarity scores, a list of repository projects that are potentially similar to the query is created.
5. The requirement specifications for each shortlisted repository project are retrieved from the repository.
6. The query (i.e., the requirement specifications for the new software system) is presented to the retrieval stage.
7. During retrieval, the actual degrees of similarity between the query and requirement specifications of shortlisted repository projects are computed. From the similarity scores, the most similar existing project to the query is determined.
8. A copy of all the artifacts for the most similar repository project is returned to the user. These artifacts include design, source code, documentation and test data.
9. The user adapts the artifacts to suit the needs of the new software system being developed. S/he stores all the artifacts for the new project in the repository, so that the project can be reused in the future. Note that at this time, the set of pre-filtering features for the new project is also stored in the repository.

4. PRE-FILTERING APPROACH

The proposed method of selecting a subset of repository projects involves comparing the set of features of the query to those of all repository projects. The features for repository projects are obtained when the projects are saved to the repository, while those of the query are gotten as soon as the query is presented to the reuse system. The features comprise a set of software metrics that capture information about the size and complexity of UML models. It is expected that corresponding metrics for similar software should not differ significantly. The set of metrics for query and repository models are represented by *n*-dimensional feature vectors, where *n* is the number of metrics. Each dimension of the vector holds the value of a particular metric. The pre-filtering similarity score for two software systems is the Euclidean distance between their feature vectors. In order to ensure that features contribute equally to the similarity score, values of each feature are normalized by dividing by the maximum value of the feature.

Table 1: Description of metrics used for pre-filtering

Metric No	Metric Description	Reference	Metric is applicable to
1	Total number of classifiers in a class diagram.	NC [6]	class diagram
2	Total number of methods in a class diagram	NM [6]	class diagram
3	Total number of attributes in a class diagram		class diagram
4	Total number of associations in a class diagram	NAssoc[6]	class diagram
5	Total number of aggregation relationships diagram (each whole-part pair in an aggregation relationship) in a class diagram	NAgg[6]	class diagram
6	Total number of dependency relationships in a class diagram	NDep[6]	class diagram
7	Total number of generalization relationships (each parent-child pair in a generalization relationship) within a class diagram	NGen[6]	class diagram
8	Total number of generalization hierarchies in a class diagram	NgenH[6]	class diagram
9	Maximum of the Depth of Inheritance Tree (DIT) values obtained for classes in the class diagram. The DIT value for a class within a generalization hierarchy is the longest path from the class to the root of the hierarchy.	MAXIMUM DIT[6]	class diagram
10	Maximum of the HAgg values obtained for classes in the class diagram. The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.	MAXIMUM HAGG[6]	class diagram
11	Number of messages in a sequence diagram		Sequence diagram[b]
12	Number of classes per use case[a]	[7]	Sequence diagram[b]
13	Number of use cases[a] per class	[7]	Sequence diagram[b]

a: use case refers to sequence diagram in this paper
b: values need to be averaged over number of classifiers or number of sequence diagrams, as the case may be, in order to obtain a single value for the metric

Table 2: Details of query and repository projects for a hypothetical example

	Actual similarity with query	Rank after retrieval stage (no pre-filtering)	Pre-filtering feature vector	Pre-filtering similarity score	Selected after pre-filtering?	Rank after retrieval stage (with pre-filtering)
Query	-	-	<0.8, 0.3, 1.0, 0.8, 0.7, 0.7, 0.7, 1.0, 0.7, 1.0, 0.8, 0.7, 0.8>	-	-	-
Repository Project 1	0.29	3	<0.9, 0.5, 0.5, 1.0, 1.0, 0, 0.3, 0.8, 1.0, 0.5, 0.3, 0.1, 0.2>	1.56	No	-
Repository Project 2	0.22	1	<0.7, 0.6, 0.8, 0.7, 0.7, 0.5, 1.0, 0.7, 0.5, 0.8, 0.6, 0.6, 1>	0.73	Yes	1
Repository Project 3	0.56	5	< 0.9, 1.0, 0.1, 0, 0.4, 0, 0, 0.5, 0.7, 0.3, 0.7, 0.3, 0.3>	2.04	No	-
Repository Project 4	0.25	2	< 0.6, 0.2, 0.7, 0.8, 0.7, 0.9, 0.6, 0.8, 0.7, 0.8, 0.9, 1.0, 0.5>	0.68	Yes	2
Repository Project 5	0.31	4	< 1.0, 0.6, 0.9, 0.9, 0.7, 1.0, 0.5, 0.9, 0.5, 0.7, 1.0, 0.3, 0.5>	0.84	Yes	3

Table 1 describes the 13 metrics that form the features of each software system. Typically, there are several sequence diagrams for each software system. However, because the number of dimensions for the feature vector (i.e., n) is fixed, metrics that are applicable to individual sequence diagrams are averaged to obtain single values. Similarly, metrics that measure values for classifiers need to be averaged over the number of classifiers in the class diagram. A classifier refers to a class or interface in a class diagram.

In the remainder of this section, a hypothetical example is presented to illustrate how pre-filtering works. Assume that a repository contains five software projects, and three of them are to be selected at the end of pre-filtering.

Table 2 provides details of the query as well as repository projects. The second column of the table shows the similarity scores computed in a retrieval stage, between the query and each repository project. It is worth mentioning that throughout this paper, lower similarity values indicate better degrees of similarity. Note that the manner in which similarity scores are computed during the retrieval stage is not described in this paper. The third column shows the ranking of repository projects based on the similarity values in the previous column. Feature vectors for the query and repository projects are shown in the fourth column. The pre-filtering similarity scores on the fifth column are the Euclidian distances between feature vectors on the preceding column. Based on the pre-filtering similarity scores, three of the five repository projects are shortlisted as shown on the sixth column. The last column of Table *2* shows that the three selected projects from the repository are ranked based on the similarity scores from the retrieval stage (i.e., from the second column of the table).

The following observations can be made from the above example:

a. Based on the pre-filtering similarity scores, the fourth, second and fifth projects are the most similar to the query, whereas from the actual similarity scores obtained during the retrieval stage, the second, fourth and first projects are the most similar to the query. These differences arisebecause the computations in the retrieval stage are more detailed than during pre-filtering. This also explains why repository project 1, which is the third most similar project to the query is not among the three shortlisted projects.
b. As shown in the last column of Table 2, Projects 1 and 3 from the repository are not compared with the query, because they were not selected after pre-filtering. Thus, pre-filtering helps to minimize retrieval time by limiting the number of repository projects that are compared with the query.

5. EXPERIMENTS

This section describes experiments for assessing the proposed pre-filtering technique. All experiments were carried out on a personal computer having the following configuration: 2.67 GHz Intel Core 2 Quad processor; 4 GB RAM; and 32-bit Windows 7 operating system.

Table 3: Description of software systems used for experiment

Software Family	Brief Description	Versions	Label in repository	No. of classifiers in class diagrams	No. of sequence diagrams	No. of messages in all sequence diagrams
Java Game Maker (JGM)	game engine for developing java games	1.9, 2.1, 2.2, 2.9, 3.1	$R_1 - R_5$	27 – 37	54 – 98	581-1838
Plot Digitizer (PD)	For digitizing data points off of scanned plots, scaled drawings, etc.	2.3.0, 2.4.1, 2.5.0, 2.6.0, 2.6.2	$R_6 - R_{10}$	44 – 66	48 – 69	648 – 942
Open Stego (OG)	steganography tool	0.2.0, 0.3.0, 0.4.0, 0.5.0, 0.5.2	$R_{11} - R_{15}$	11 – 59	15 – 92	172 – 3895
JOrtho (JO)	Java based spell checker	0.2, 0.3, 0.4, 0.5, 1.0	$R_{16} - R_{20}$	22 – 56	66 – 88	419 – 2175
51 Degrees (51D)	For detecting mobile devices that browse a website	2.2.8.5, 2.2.8.6, 2.2.8.7, 2.2.8.8, 2.2.8.9	$R_{21} - R_{25}$	52	77 – 84	1331 - 1366
Jcurses (JC)	Console toolkit for Windows®	0.91, 0.92, 0.94, 0.95, 0.95b	$R_{26} - R_{30}$	57 - 66	178 - 254	4991 – 9291

5.1 Evaluation Criteria

The following criteria would be used to assess the pre-filtering approach described in this paper.

5.1.1 Mean Average Precision (MAP)

The average precision (AP) for a query is obtained using precision values calculated at each point when a new relevant document is retrieved (using precision = 0 for each relevant document that was not retrieved). Mean Average Precision, also referred to as *mean precision at seen relevant documents* for a set of queries is the mean of the AP scores for each query [8]. The formula for MAP is given in (1).

$$MAP = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{Q_j}\sum_{i=1}^{Q_j}P(rel=i) \quad (1)$$

In (1), N is the number of queries, Q_j is the number of relevant documents for query j and $P(rel = i)$ is the precision at the i^{th} relevant document. MAP was chosen to measure retrieval efficiency because it is widely used to evaluate ranked information retrieval systems.

5.1.2 Retrieval Time

Since the main reason for employing pre-filtering is to minimize retrieval time, it is important to measure retrieval time with and without pre-filtering.

5.1.3 Correlation between Similarity Scores and Estimated Reuse Effort

Even though a reuse system is able to retrieve relevant projects from a repository with high MAP, it is possible that it is only good at ranking the repository projects but the similarity scores themselves are meaningless. To address this possibility, we shall examine the degree of correlation between the similarity scores returned by the reuse system and estimated modification (reuse) effort. Since a significant amount of reuse effort is dedicated to programming, code-based sizing metrics will be used to estimate reuse effort. The formula employed by Basili et al. [9] for predicting software maintenance effort will be used to predict reuse effort in this paper. We reasoned that maintenance effort is more or less proportional to reuse effort since they both involve efforts to modify existing software to meet some current needs.(Reuse) effort in man hours is estimated as follows [9]:

$$\text{Effort} = 0.36 * \text{effective SLOC} + 1040. \quad (2)$$

In (2), effective source lines of code (SLOC) is the sum of added, deleted and modified SLOC. A strong degree of correlation between similarity scores and estimated reuse effort shows that similarity scores returned by the reuse system can provide a reuser with a rough estimate of the amount of effort needed to adapt

retrieved software artifacts to suit the needs of the software system being developed.

5.2 Experimental Data

Data scarcity is a common problem for software engineering research [10]. Due to the unavailability of software reuse repositories containing UML diagrams, we reverse engineered class and sequence diagrams for six families of open source software using Altova® UModel®. The software was obtained from SourceForge, a popular web-based source code repository. The repository contained five versions of each software family, making a total of 30 projects. Furthermore, 30 queries Q_1 ... Q_{30} were formed by using each of the repository models in turn (i.e., $Q_i = R_i$, $1 \leq i \leq 30$). The similarity between each query and every repository project was determined. Intuitively, R_i is relevant to Q_i ($1 \leq i \leq 30$) only if Q_i and R_i are versions of the same software family. For example, $R_1 \ldots R_5$ are relevant to $Q_1 \ldots Q_5$, while $R_{26} \ldots R_{30}$ are relevant to $Q_{26} \ldots Q_{30}$. A brief description of the various software families is presented in Table 3.

5.3 Results

The first experiment was aimed at determining the performance of the pre-filtering stage without any retrieval stage. Table 4 shows shows the retrieval time as well as correlation between pre-filtering similarity scores and reuse effort when pre-filtering is considered as a stand-alone retrieval stage. In another experiment, we measured the MAP as the number of projects returned by the pre-filtering stage is varied. The horizontal axis of Figure 2 shows that the number of projects returned after pre-filtering is varied from 5 to 30. The vertical axis shows the MAP.

The final experiment studied the effect of pre-filtering on MAP and retrieval time, by comparing these values when retrieval is performed with and without pre-filtering. The number of projects returned after pre-filtering was set to 10. Figure 3 and Figure 4 show how pre-filtering affects MAP and retrieval time, respectively.

5.4 Discussion of Results

As can be inferred from and Figure 2, the proposed pre-filtering approach meets its objective; it is computationally inexpensive. Moreover, it returns many of the relevant repository projects in response to the queries, since the MAP is 76.34% when only five projects are shortlisted (see Figure 2). However, the correlation between the pre-filtering similarity score and predicted reuse effort is only 0.61, compared to a correlation coefficient of 0.78 when the similarity scores from the retrieval stage were compared with estimated reuse effort. It is inconsequential that pre-filtering similarity scores do not have a high correlation with reuse effort, because the actual similarity scores between query and repository projects will be determined in the retrieval stage. Nonetheless, these results indicate that it is not advisable to rely on pre-filtering alone (i.e., without a retrieval stage) during software reuse.

Figure 3 shows that pre-filtering caused MAP to drop from 92.74% to 84.94%. This decrease in MAP is expected since pre-filtering inadvertently omits some relevant repository projects as a result of its shallow comparison, which is based on only 13 metrics. Furthermore, it can be observed from Figure 4that pre-filtering led to a sharp decline in retrieval time. The retrieval time dropped from 2,473 seconds to 888 seconds. This represents a speed-up of approximately 2.8.

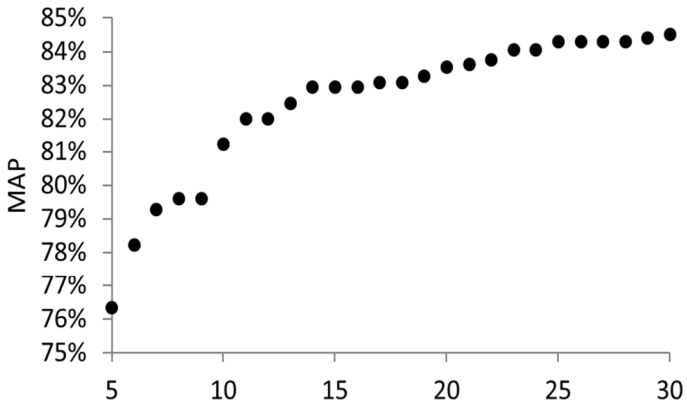

Figure 2: Relationship between MAP and number of projects selected after pre-filtering

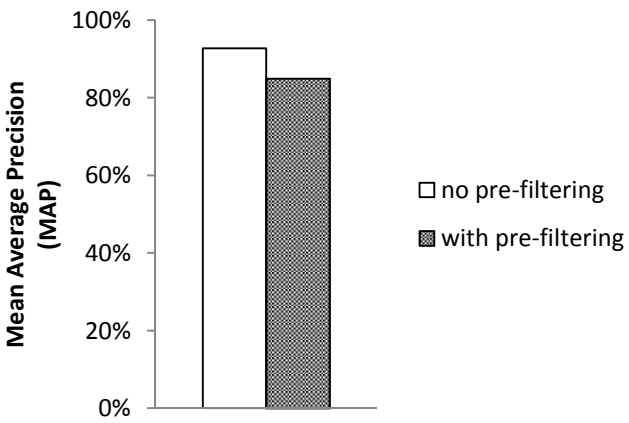

Figure 3: Effect of pre-filtering on MAP

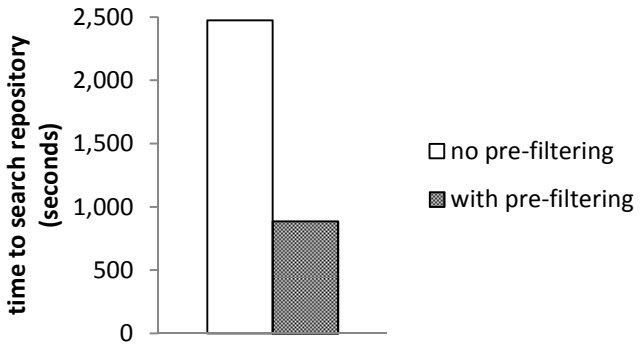

Figure 4: Effect of pre-filtering on retrieval time

Table 4: Performance of pre-filtering stage without a retrieval stage

Correlation with reuse effort	Time to search repository (milliseconds)
Correlation Coefficient = 0.6130 Significance Level = 2.56×10^{-49}	3.99

6. CONCLUSION AND FUTURE WORK

In order to minimize retrieval time during software reuse, this paper proposed a technique for pre-filtering repository projects prior to retrieval. Experimental results show that pre-filtering results in very substantial decrease in retrieval time. Pre-filtering led to a slight reduction in MAP, owing to its use of a superficial approach for selecting projects' based on supposed similarity. It does not matter that there is a low degree of correlation between pre-filtering scores and estimated reuse effort since pre-filtering is expected to be followed by the retrieval stage, which has been shown to produce a strong degree of correlation with reuse effort.

Our pre-filtering technique can be extended in several ways. Firstly, if it takes into account the names of concepts that occur in the models, it will help to sieve out projects that belong to application domains different from that of the query. Secondly, the number of repository projects returned after pre-filtering was fixed in the experiment. More research is needed to automatically determine the proportion of repository projects to be returned after pre-filtering. On one hand, selecting a large fraction of repository projects after pre-filtering defeats the aim of pre-filtering. On the other hand, choosing a small proportion of repository projects may lead to a significant decrease in MAP, since many relevant projects may not be shortlisted at the end of pre-filtering. Thirdly, more experiments can be performed to determine if the effect of pre-filtering can be improved by adding or removing from the metrics listed in Table 1.

7. ACKNOWLEDGMENT

The author would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM), Saudi Arabia under Research Grant 11-INF1633-04.

8. REFERENCES

[1] I. Sommerville, *Software Engineering*, 9th ed.: Pearson Addison Wesley, 2010.

[2] S. Channarukul, S. Charoenvikrom, and J. Daengdej, "Case-based reasoning for software design reuse," in *Aerospace Conference, 2005 IEEE*, 2005, pp. 4296-4305.

[3] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Case-Based Reuse of UML Diagrams," in *The Fifteen International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*, 2003, pp. 335-339.

[4] W.-J. Park and D.-H. Bae, "A two-stage framework for UML specification matching," *Inf. Softw. Technol.,* vol. 53, pp. 230-244, 2010.

[5] H. O. Salami and M. A. Ahmed, "A framework for reuse of multi-view UML artifacts," *International Journal of Soft Computing and Software Engineering,* vol. 3, pp. 156 - 162, 2013.

[6] M. Genero and M. Piattini, "Empirical validation of measures for class diagram structural complexity through controlled experiments," in *5th International ECOOP Workshop on Quantitative Approaches in object-oriented Software Engineering (QAOOSE 2001)*, 2001.

[7] J. Muskens, M. Chaudron, and C. Lange, "Investigations in applying metrics to multi-view architecture models," in *30th Euromicro Conference*, 2004, pp. 372 - 379.

[8] S. Teufel, "An overview of evaluation methods in TREC ad hoc information retrieval and TREC question answering," *Evaluation of Text and Speech Systems,* pp. 163-186, 2007.

[9] V. Basili, L. C. Briand, S. Condon, Y.-m. Kim, W. L. Melo, and J. D. Valettt, "Understanding and Predicting the Process of Software Maintenance Releases," in *in proceedings of the 18th international conference on software engineering*, 1996, pp. 464-474.

[10] D. Zhang and J. J. P. Tsai, *Advances in Machine Learning Applications in Software Engineering*: IGI Global, 2007.