International Journal of Energy, Information and Communications
Vol.6, Issue 2 (2015), pp.1-16
http://dx.doi.org/10.14257/ijeic.2015.6.2.01

Detecting High Level Similarities in Source Code and Beyond

Dhavleesh Rattan[1], Rajesh Bhatia[2] and Maninder Singh[3]

[1]*Punjabi University, Patiala*
[2]*PEC University of Technology, Chandigarh*
[3]*Thapar University, Patiala*
[1]*dhavleesh@rediffmail.com*, [2]*rbhatiapatiala@gmail.com*, [3]*msingh@thapar.edu*

Abstract

This paper presents a novel method to compute similarity across object oriented programs at different levels of granularity. The tool is able to detect concept level similarities by applying latent semantic indexing and principal component analysis. Previous research has shown that detection of high level similarities can help in comprehending the design of the system for better maintenance. Moreover, model driven development has become a standard industry practice; we have extended our tool to detect similarities for UML diagrams by measuring the distance between two class models. In addition, we mined important change patterns at method level using multi-version program analysis by applying the proposed technique throughout the evolutionary history of the software. We have validated the similarity score by applying the tool at function level in the source code. We assess the usefulness and scalability of the proposed technique by empirical evaluation on source code of open source subject systems and multi-version program analysis on 8 releases of dnsjava.

Keywords: *principal component analysis, latent semantic indexing, class diagrams, multi-version program analysis, concept clones*

1. Introduction and Motivation

Fragments of code are usually copied from one location to another in software development. This copied code is called a software clone. The process of software cloning may lead to bug propagation thus a bug detected in one section of code therefore requires correction in all the replicated fragments of code [12,15]. Therefore, clone detection has emerged as an active research area. A large number of clone detection tools/ techniques developed till date detect clones with granularities like number of tokens, lines of code, methods, *etc.*, [12]. It is true that similarities exist at higher level of abstraction in software.

In this paper, we present a tool to detect concept clones. Concept clones are defined as similar program structures within one software system or across versions reflecting similar domain concepts. Marcus and Maletic [18] identified a scenario in which the software developer writes fresh solution to a problem (knowingly or unknowingly) for which the solution is already existing. This scenario leads to different solutions for the same problem. Identification of high level concept clones helps the software developer in understanding and comprehending the system at abstract level [19]. We realized the importance of similar domain concepts for sharing and interoperability.

During evolution, high level similarities emerge. The presence of structural similarities as the software system evolves helps in improving the understanding of the system. Detection of high level similarities throughout the history of the subject system helps in improving the different attributes of quality of the software like maintainability, reusability, extensibility. As an important application of the proposed technique, we

ISSN: 2093-9655 IJEIC

applied it in coarse-grained multi-version program analysis to detect changes in methods across versions of popular Java open source system *dnsjava* [20].

To understand and analyze the concept clones, we detected clones at method level too. A manual analysis of 219 sample methods within one version of software system is done to validate the technique.

Our work goes beyond source code leveraging the same tool and detects high level similarities for UML class models. Since the graphical UML (*Unified Modeling Language*) [16] is increasingly replacing conventional programming languages for developing and modeling software systems. The presence of design level similarities in form of concept clones cannot be precluded in UML models. In our work, we define **concept model clones** as two different class models representing the same conceptual/ domain model or design model [13]. The class diagrams may be created by different people pertaining to same domain.

The remainder of this paper is structured as follows: Section 2 presents the related work pertaining to high level concept clones. Section 3 describes our approach of clone detection. Empirical evaluation and implementation is shown in Section 4. Finally Section 5 concludes and shows the future directions.

2. Related Work

We observed that research in code clone detection is a well established field. In this section we will discuss only the techniques which detect conceptual/ structural clones.

Marcus and Maletic [18] gave a technique using latent semantic indexing to search for similar concepts extracted from comments and source code. They applied latent semantic indexing (LSI) on the textual representation of source code to identify semantic similarities across functions/ files/ programs. LSI is vector based statistical method to represent meanings of comments and identifiers of source code. They tried to detect similar high level concept clones e.g. abstract data types.

Basit and Jarzabek [19] developed Clone Miner using frequent itemset mining which works on the output of token based clone detection tool , Repeated Token Finder. The study analyzed the presence of simple clones in structural clones. Across a comprehensive set of 11 subject systems, structural clones are analyzed to identify their location, similarities, etc. The analysis of structural clones i.e. high level similarities is intended to help in understanding, refactoring and maintenance of the software system. The study concluded that over 50% simple clones were a part of structural clones.

Fanta and Rajlich [11] propose different types of clones in their taxonomy which is primarily aimed at eliminating clones in object oriented systems. The study defined class clones representing identical concepts and function clones sharing common code.

Grant and Cordy [17] introduced a technique based on independent component analysis to analyze vector space representations of methods in software systems.

The novelty of the proposed work lies in detection of concept model clones across UML models. Another highlight of the proposed technique is to detect concept clones across 8 releases of *dnsjava.*

3. Approach

The proposed method compute the similarity between source code elements/ UML class models at three levels. We detected high level concept clones in UML class models, similarity between source code software elements across versions, clones in software code fragments at method level as shown in Figure 1. We will refer to all the three types of input as source files. Currently the tool gives the similarity score between 0 and 1 at every level automatically once the file/files are given as input. Within a single class file as input, it compares every method with other method and generates the output. We detected the

high level similarities first and then explored the files deeper inside at method level to verify the results.

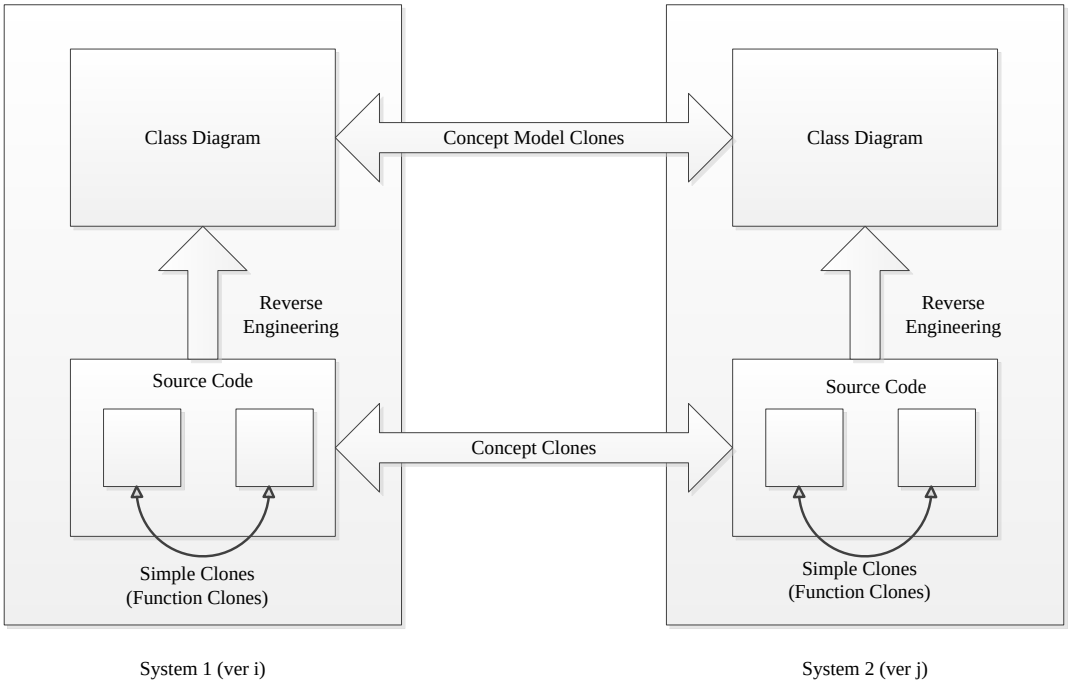

Figure 1. Different Levels of Clone Representation

We applied the well known techniques which are widely applied in information retrieval field to search for similarities of concepts extracted from source code elements/ class model element. Further, we explore the details of our technique.

3.1 Vector Space Model

A vector space model [7] is a mathematical model having a consistent mathematical structure which advances conceptual understanding. In a data centric approach of vector space model, the data is typically represented as a matrix where vector is used to represent each item or document in a collection (database). Each component of the vector reflects a particular concept, key word, or term associated with the given document. The value assigned to that component reflects the importance of the term in representing the semantics of the document. Typically, the value is a function of the frequency with which the term occurs in the document or in the document collection as a whole. In our work, a database is a collection of two or more class diagrams/ package/ source code files. Terms are non-unique tokens extracted from different class models in the database or tokens extracted from methods in source code. So a database containing a total of d documents described by t terms is represented as a term-by-document matrix A.

$$A = t \times d$$

The d vectors representing the d documents form the columns of the matrix. Thus, the matrix element a_{ij} is the weighted frequency at which term i occurs in document j. In the parlance of the vector space model, the columns of A are the document vectors, and the rows of A are the term vectors. The semantic content of the database is wholly contained in the column space of A, meaning that the document vectors span that content. We try to exploit geometric relationships between document vectors to model similarities and differences in content. A variety of schemes are available for weighing the matrix

elements. The elements a_{ij} of the term-by-document matrix A are often assigned two-part values $a_{ij} = l_{ij}\ g_i$. In this case, the factor g_i is a global weight that reflects the overall value of term i as an indexing term for the entire collection. The factor l_{ij} is a local weight that reflects the importance of term i within document j itself. Local weights range in complexity from simple binary values (0 or 1) to functions involving logarithms of term frequencies. Global weighting schemes range from simple normalizations to advanced statistics-based approaches [3, 17].

The existence of a vector space implies that we have a system with linear properties: the ability to add together any two elements of the system to obtain a new element of the system and the ability to multiply any element of the system by real number [6]. This is a model in which documents are mapped as real-valued vectors $d_i = (w_{i1}, w_{i2}. . . w_{im})$ where w_{ij} corresponds to the j^{th} key (j = 1, 2, . . .m) of the i^{th} document (i = 1, 2, . . ., n). Thus, the collection is represented as an n* m document-term matrix D of n documents and m keys. To weight the keys, we applied the well-known tf*idf scheme

$$W_{ij} = tf_{ij} \times \log \frac{N}{df_j}$$

Where the inverse of the document frequency df_j (number of documents which include the jth key) is multiplied by the key frequency tf_{ij} in a document. There are variations in the equation in addition to the former basic formula. Although tf.idf focuses on key frequencies in individual documents and the generality of keys in the collection, they are sensitive to the lengths of documents or numbers of words in them: the longer the document the more key occurrences. The normalization of document vectors adjusts the effect of both increasing word frequencies and increasing the numbers of word matches for increasing document lengths [1, 6].

3.2 Latent Semantic Indexing

Latent semantic indexing (LSI) was introduced in 1990, and is able to predict some underlying latent semantic structure that is partially not visible due to randomness of selected tokens with respect to retrieval [17, 21]. LSI has been used to detect clone in source code [18]. Vector Space Model (VSM) forms the basis for LSI [8]. We are implementing LSI using singular value decomposition (SVD). Here, we take large matrices of diagram-token association data and generating a semantic space wherein tokens and diagrams that are closely related to each other are placed near one another. SVD is a method for transforming correlated variables into a set of uncorrelated ones that better exposes the major associative patterns in the data, and ignores the smaller and less important data elements. Although, it reduces the dimension of original matrix, but at the same moment it also reflects data points which exhibit the most variation. That is, to identify similar patterns of data, similarity measures can be applied to the reduced, synthetic features rather than to original dataset. This approach has advantages in terms of speed, memory and even accuracy. One particularly nice application of this idea is to combine information retrieval with the use of principal components in dimension reduction.

3.3 Principal Component Analysis

The main idea of using principal component analysis (PCA) is to reduce the dimensionality of input data set in which there are a large number of interrelated variables, while maintaining as much as possible of the variation present in the data set. This variation and reduction in input data set is achieved by transforming it into a new set of variables, the principal components, which are uncorrelated, and which are ordered so that the first few retain most of the variation present in all of the original variables. It is a way of identifying patterns in data, and expressing the data in such a way as to highlight

their similarities and differences. Since patterns in data can be hard to find in data of high dimension, where the luxury of graphical representation is not available, PCA is a powerful tool for analyzing data [9]. Firstly, data are typically mean corrected by subtracting its mean from each variable [4, 5, 21]. Secondly, if large differences in the variances of the original variables do not reflect their relative importance, the mean-corrected data may be standardized by dividing each variable by its standard deviation. The objective of linear PCA is to use a linear mapping to find a set of d orthogonal basis vectors that maximally captures the relationship between original dimensions. The process of determining most influential features having maximum eigen values is called SVD.

Singular value decomposition (SVD) is a decomposition of a matrix. It helps in finding principal components. Then we find covariance matrix from the normalized matrix. Then we apply SVD on covariance matrix of both data sets, which picks out structures in one data set which are best correlated with the structures in the other data set.

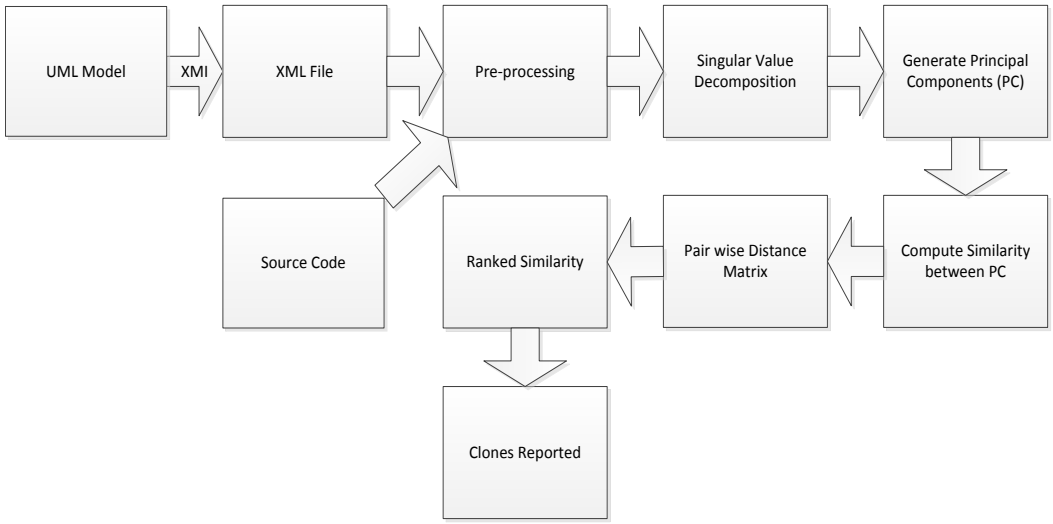

Figure 2. Overview of the Process

3.4 Similarity Metric

When documents are represented as term vectors, the similarity of two documents corresponds to the correlation between the vectors. This is known as the cosine of the angle between vectors, that is, cosine similarity. Cosine similarity is one of the most popular similarity measure applied to text documents, such as in numerous information retrieval applications and clustering too. In this experiment documents are classes or class diagrams. Similarity of two document vectors will be calculated as:

$$Cosine\ Similarity\ (d_1, d_2) = \frac{d_1 \times d_2}{|d_1|.|d_2|}$$

Where d_1 and d_2 represents their own term weights and $|d_1|$ and $|d_2|$ represents length of vectors.

4. Implementation and Empirical Evaluation

Software developers may be interested to know the similarity between software elements at different levels depending upon the need. The developed tool works at different levels in a top down fashion. Firstly we detected larger clones at class model level in UML models. Then we checked the similarity across classes in the source code to

check identical concepts. At the lower level, we checked simple clones at method level for source code file.

Once the similarities between source code elements have been computed at every level, software developer (maintainer) may cluster input class models/ files/ methods depending on score. One cluster at the abstract level of model or class will represent similar concepts implemented. Similar clusters of methods may also be grouped into different classifications depending upon the interval in which the score lies. The files which are very close (score between 0.9 and 1) significantly describe similar high level concept at all levels.

4.1 An Example

Figure 2 shows the overview of the process. In this technique, firstly, input source code will be searched for tokens using a java API *i.e.*, javaparser. The technique works in three phases as mentioned below:

1) **Parsing and identification of non-unique tokens**

2) **Computation - involves SVD and PCA**

3) **Comparison and reporting - involves computing cosine similarity between principal component vectors**

Three classes are made for this purpose:-

(1) FileParserManager (2) ClassFileParser (3)ClassFileComputation

1) Parsing and identification of non-unique tokens:

We used javaparser, a freely available tool for parsing. It consists of a class FileParserManager. Once we drag and drop the target file to the tool, this class will immediately call its subordinate class *i.e.*, ClassFileParser that will further request java parser to find methods and tokens file contains. It will return counted methods and tokens to class FileParserManager. Here we are considering only those tokens that at least consists of 3 letters and have a frequency count >2.

2) Computation :

It involves calculating local and global weights of individual tokens within a given target file and computing singular value decomposition of given vector space. SVD helps in removing insignificant tokens. At last, principal component vectors will be identified.

3) Comparison and Reporting

This is the last step of the process in order to detect clones. As in second step, we have identified principal component vectors. Similarity value will be computed within file and in-between files.

4.2 Real World Concepts

It has been planned to identify examples from related concepts stemming from real world scenarios as concepts are the building blocks of any information system. In one example, we took small class diagrams for library management system modeled by different developers. Figure 3 shows both the diagrams. We understand that UML class model is more like a graph but in our work we are not considering two dimensions of models. Our approach is motivated by following key points:

- UML class model is a graph but source code is a graph too considering control and data flow.
- Source code is written linearly but model is code too [10]. In Mark Harman's [10] keynote address in SCAM 2010, he articulated the idea of high level code for UML class models. We have textual representation for UML class models in the form of XMI files [2]. We used the XMI file to extract terms *i.e.*, name of class, name of the attribute and name of the operation from the class diagram.
- Conceptual models are usually developed independently by different persons. Identification of structural patterns i.e. name of class, name of attribute/ method in UML class diagrams and tokens in source code helps in comparison of conceptual models. Further it may aid model integration and merging. Such a relationship between conceptual models promotes interoperability.

Applying our approach, we came to the similarity score of 0.7365 for library management system. The similarity score is a clear indication of proximity of both conceptual models.

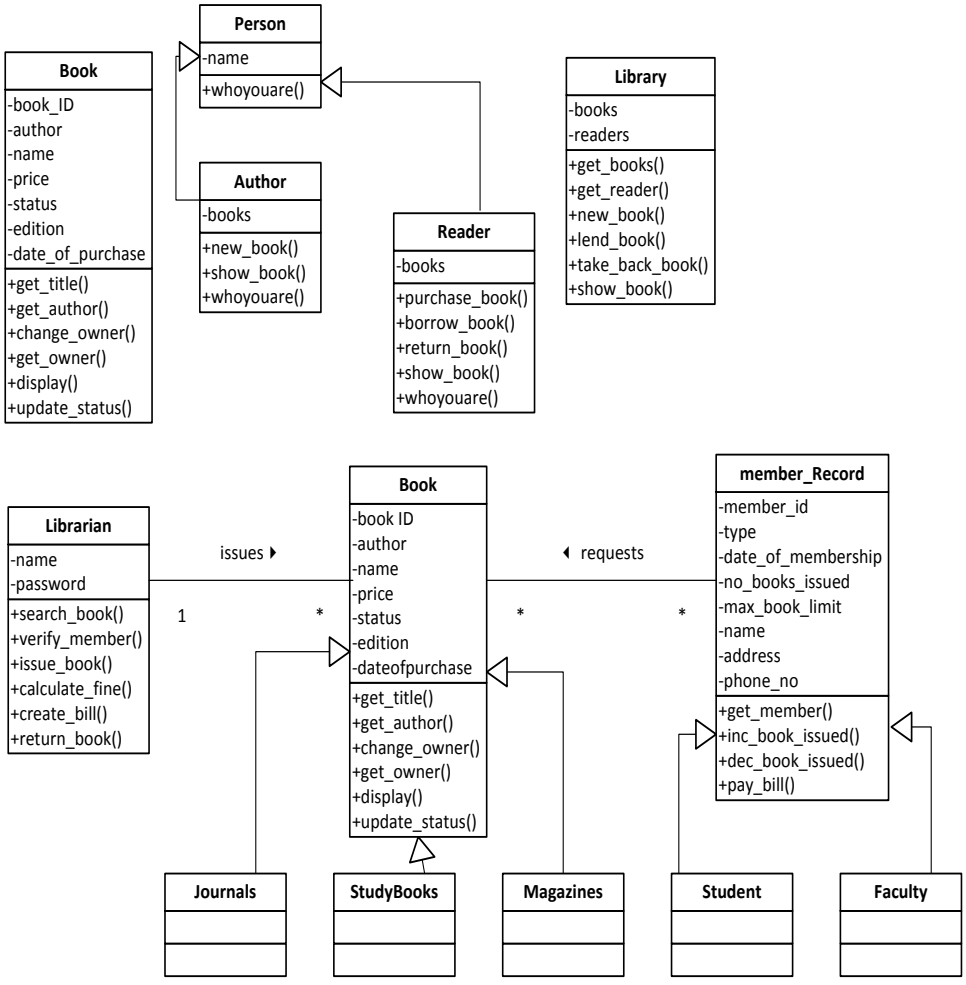

Figure 3. Class Diagrams Showing Library Management System

4.3. Conceptual Clones Across Version Control Systems

Comparison of class models/ source code across versions helps in identifying conceptual clones. We may have several related models/ source code files describing the

same concept. The related files may be representing different variants of the same software. In another application of the technique, in *copy-modify-merge* model of version control system, many users simultaneously modify their private copies. We can apply this technique to identify similar concepts across projects being developed by users working on same software. Whether one intends to merge/differ two models/ source code files, our tool is helpful in checking the proximity of models/ source code files. To demonstrate the usefulness of our approach, we have taken the example of open source system in java. We used the version information to track high level clones across versions of the software system. We are working at class model/ source code class level of granularity across versions. Similar files across versions represent similar intentions.

Multi-version Program Analysis/ Software Evolution/ Origin Analysis

Understanding the history of a software system helps the software developer in quick program analysis. As new versions of software appear, the software elements tend to change to keep it more useful as per the current requirements. We have tried to learn the evolution history by using techniques of high level concept clone detection. Using multi-version program analysis, we compared the number of functions between subsequent versions of the software. We have mined important change patterns from the evolution history of software system.

We have performed a case study on *dnsjava* [20]. It is open-source software system, an implementation of Domain Name Server in Java. The *dnsjava* project started in September 1998. It is still an active project and continues to evolve. For our case study, we selected 8 releases of *dnsjava* starting from 2.0.6 (Jan. 2008) to 2.1.4 (Jan. 2013). We decided to study the class files in org/xbill/dns sub package which is carrying out most of the DNS functions.

Table 1 summarizes the releases of *dnsjava* and its growth. The number of files is the number of Java class files in org/xbill/dns sub package. It has increased from 107 to 117 in the last five years.

Table 1. Growth of dnsjava

Sr. No.	Release	Date	# files	# LOC	# SLOC
1.	dnsjava-2.0.6	24 Jan, 2008	107	18536	9918
2.	dnsjava-2.0.7	25 Sep, 2009	112	19503	10325
3.	dnsjava-2.0.8	21 Nov, 2009	112	19514	10328
4.	dnsjava-2.1.0	07 Sep, 2010	112	20308	10865
5.	dnsjava-2.1.1	09 Feb, 2011	112	20387	10914
6.	dnsjava-2.1.2	24 July, 2011	116	20991	11178
7.	dnsjava-2.1.3	24 Oct, 2011	116	21085	11226
8.	dnsjava-2.1.4	04 Jan, 2013	117	21454	11441

We applied the tool across subsequent versions of *dnsjava*. We compared class files to get the score between 0 and 1, where 1 denotes a perfect match. Value close to 1 indicates high proximity than the value close to 0. As shown in the Table 2 most of the class files

are unchanged as they are exact duplicates i.e. value 1. We picked up all those class files where the score is less than 1. Those class files in both the versions are analyzed to detect method duplicates. It helps in knowing whether the change across versions in induced by cloning or not. We have chosen program element matching at file level of granularity to check for the number of methods to aid program understanding for the programmer. Our technique is robust against file renaming and method renaming. This is due to extraction of terms from files and methods and similarity computation based on non-unique tokens.

Table 2. Concept Clone Detection across Version of dnsjava

Sr. No.	Vold → Vnew	Name of class file	Change in number of methods	Addition/ Deletion	Cosine similarity result
1.	2.0.6 → 2.0.7	APLRecord	11 → 12	+1	1.00
2.		LOCRecord	16 → 18	+2	0.61
3.		Name	35 → 34	-1	0.62
4.		NSECRecord	10 → 8	-2	0.48
5.		ResolverConfig	16 → 17	+1	0.62
6.		Tokenizer	32 → 34	+2	0.63
7.		UDPClient	6 → 7	+1	0.61
8.	2.0.8 →2.1.0	DNSSEC	2 → 31	+26	0.48
9.		KEYBase	8 → 5	-3	0.74
10.		ResolverConfig	17 → 18	+1	1.00
11.		SIGBase	13 → 14	+1	0.74
12.		UDPClient	7 → 8	+1	0.66
13.	2.1.0 →2.1.1	Lookup	10 → 11	+1	0.62
14.		ResolverConfig	18 → 21	+3	0.58
15.	2.1.1 →2.1.2	Address	17 → 18	+1	0.60
16		DNSOutput	13 → 14	+1	0.62
17.		DNSSEC	31 → 32	+1	0.68
18.		OPTRecord	12 → 11	-1	0.65
19.		Record	40 → 41	+1	0.50
20.		WireParseException	2 → 3	+1	0.00
21.	2.1.2 →2.1.3	ZoneTransferIn	27 → 34	+7	0.67
22.	2.1.3 →2.1.4	DNSSEC	32 → 36	+4	0.63

We evaluated the proposed tool for all the class files in which there is change in number of methods. Figure 4 shows the number of class files added or removed across versions of *dnsjava*.

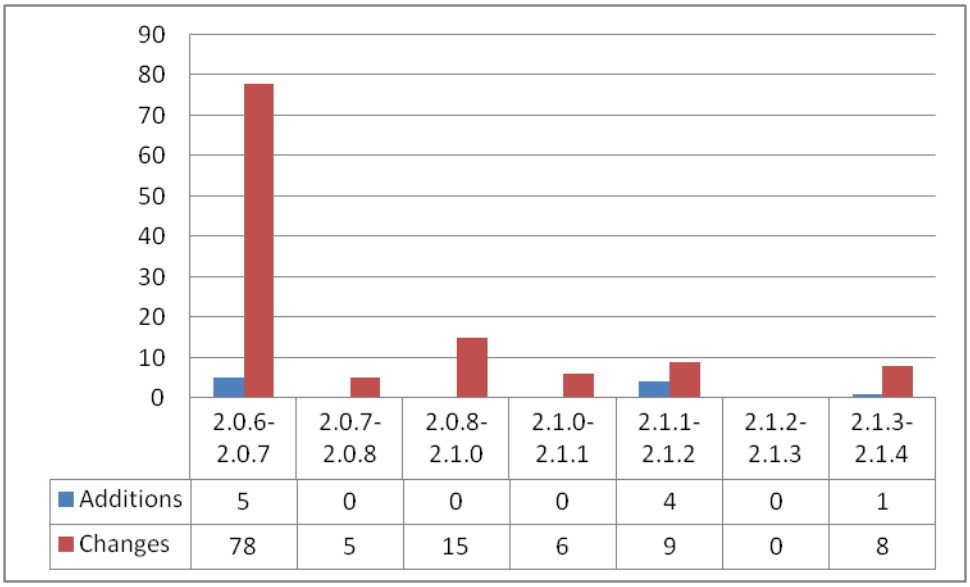

Figure 4. Number of Class Files Added/ Removed Across Versions

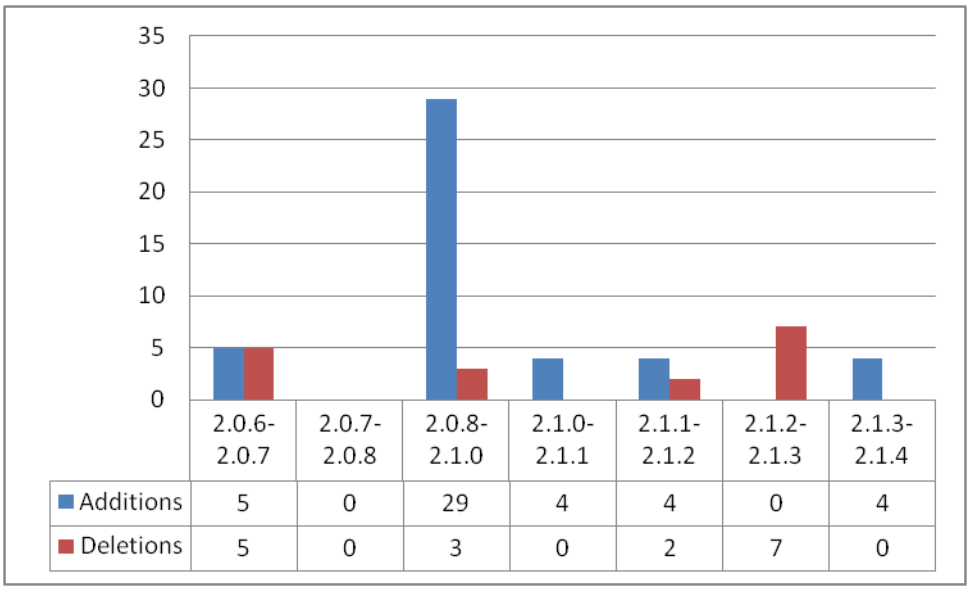

Figure 5. Number of Methods Added/ Removed Across Versions

To measure the performance of the proposed technique based on all the comparisons between Java class files for two versions, we manually verified the change in number of methods for every pair of comparison. Figure 5 shows the number of methods added or removed across versions of *dnsjava*. The total number of comparisons is given by the number of files which are to be compared for both the versions in the sub package of *dnsjava* we have taken for study as mentioned in Table 2. We chose to measure precision, recall and F-measure for the proposed technique. Is the technique working for all possible pair of files is described by recall measure? Are the generated pairs actually reflected evolutionary change is described by precision measure? Let A be the set of actual file pairs to be compared. Let D be the set of generated pairs by the technique. Let C is the set of possible candidate pairs. The

Precision = D/C

Recall = D/A

F-measure is the harmonic mean of precision and recall. It describes overall performance of the proposed technique and is calculated as:

F-measure = 1/ ((1/precision) + (1/recall))

We have 22 class files in which the number of methods has been changed. Upon manual analysis by checking software change management repositories, it is confirmed that there are 22 class files across different releases which have undergone change. So actual file pairs which are compared and possible candidate pairs is same i.e. A=22 and C=22. The technique fails to work in one case i.e. wireparseexception.java between 2.1.1 and 2.1.2 releases as there is single non-unique token only.

So Recall = 21/22=95.45

As far as precision is concerned, there are three comparisons in which the technique outputs wrong result. In two class files viz. APLRecord.java across 2.0.6 and 2.0.7 releases and ResolverConfig.java between 2.0.8 and 2.1.0 releases, result is 1 but there are changes in both the files as one new method is added in both the class files. So the tool should output less than 1. The third class file is mentioned above in which technique fails to deliver.

So Precision=19/22=86.36

F-measure = 2/ ((22/20) + (22/21))=2/2.15=0.93

In this paper, we have not extended our technique to detect changes at high level of abstraction *i.e.*, class diagram reverse engineered from the corresponding code for *dnsjava*. This is due to large time involved in generating class diagrams for all class files using any UML modeling tool supporting reverse engineering. In future, we may wish to extend current multi-version program analysis at class diagram level representation.

4.4 Clones within the File at Method Level

We also applied the technique to trace proximity across methods in class files changed over time. It is done to detect method clones i.e. whether cloning has led to change in files over versions. Simple clones are part of conceptual clones at file level in source code. This is verified by applying the tool to detect code clones at function level of granularity within one version. We took dnsjava-2.1.3 to evaluate the effectiveness of our approach to detect function clones. We applied the technique on 30 java class files in subfolder dnsjava-2.1.3/org/xbill/dns randomly having 219 functions. Within every class file, first method is compared with the second. Undoubtedly, methods represent a coherent piece of code performing a particular function. As the technique is based on extracting common tokens within two methods, we observed the methods for which the comparison lies between 0.9 and 1.0. We checked such methods manually to verify if the score between 0.9 and 1.0 is really a clone.

We verified the output manually for a sample set of 219 functions in single version of *dnsjava*. We validated the technique by measuring the precision and recall for percentile values lying between 0.9 and 1.0. Upon comparison, we get a ranked list of scores between methods. We observed the methods for which the score lies between 0.9 and 1.0.

While it is true that the applied technique doesn't compare all the methods across files with each other, yet the proximity value within first method and the next clearly depicts viability of the approach.

4.5 Other Applications

- We view our system as an application in software product line analysis. Software product line is based on reusing similar concepts by applying common means of production. Some of the studies [12] have applied clone detection in software product lines. We may apply our technique to identify similar concepts belonging to one family, thereby promoting product line design for that domain.
- The technique has got an added advantage that it is not defined for any specific modeling language but will work for all languages supporting XMI standard [22]. It also overcomes the tight integration of the UML with any modeling tool. XMI is a well known exchange representation by OMG for UML models.
- We may extend the tool to detect plagiarism in UML class models. In one typical scenario, the professor assigned the task to students to create UML class models related to same domain. Our technique can be used to measure the degree of similarity among all the class models created by students to detect the degree of plagiarism. The tool helps in exporting the *.mdl* files of all the students to XMI representation and comparing all the models with each other in one go. The degree of similarity is a clear indication of level of plagiarism.

4.6 Comparison with Existing Work

Clone detection in source code is an active research area and a number of techniques have been developed [12]. But, the focus of the proposed work is to detect high level similarities in source code and UML class models. Therefore, in Table 3 we compare the proposed technique with those existing techniques which aim to detect high level similarities/ concept clones in source code.

We understand that the goal of all the studies mentioned in Table 3 is to detect higher level similarities but the techniques and methods used are different.

Table 3. Comparison of Proposed Work with Existing Methods

	Marcus & Maletic [18]	**Basit & Jarzabek [19]**	**Grant & Cordy [17]**	**Proposed Method**
Intermediate representation/ transformation technique	comment removal and token regularization	tokens	vector space representation	vector space representation
Match detection algorithm	vector representation using LSI	frequent itemset mining	independent component analysis	principal component analysis and cosine similarity
Clone granularity level	code segments, files	files, methods	methods, blocks	class diagram, class file and methods
Subject system(s)/ Language	Mosaic 2.7/ C	Eclipse Graphical Editing Framework/ Java,	Linux/ C	dnsjava/ Java

		Eclipse Visual Editor/ Java, OpenJGraph/ Java J2ME Wireless Toolkit 2.2/Java Java Pet Store 1.3.2/ Java		

Marcus and Maletic [18]'s approach is based on examining identifiers and comments to find similar high level concepts but our technique is based on extracting non-unique tokens from input file and ignores identifiers and comments. Another work closest to ours is possibly that of Grant and Cordy [17]. Their technique applied independent component analysis and evaluated C programs at method level of granularity using raw distance metric. Our technique uses principal component analysis and works for object oriented programs at three different levels of granularity by calculating distance using cosine similarity.

Basit and Jarzabek [19] gave the concept of structural clones which are made up of simple clones that co-exist and relate to each other. We have successfully validated our approach in detecting concept clones across 8 releases of *dnsjava* by calculating precision and recall. Detecting high level concept model clones across UML models is another novelty of our approach.

5. Conclusion and Future Work

The user may choose different implementations in the form of class model or source code for same real world concept.

To the best of our knowledge, this is the first attempt to apply principal component analysis to identify concept clones within single software system or across versions. It is more important to identify concept clones at abstract model level than at lower levels. Thus, we have followed a top down approach starting from UML class models down to method level within same source code file. We understand the importance of user intervention to fully automate the presence of concept clones in a single version and across versions. The tool will assist the software engineer (maintainer) to understand as to why two different domain concepts are so close).

The application of the tool on UML class models does not take the structure of the class diagram into account. Calculation of similarity between two graphs is a lot more computationally expensive than between two vectors. We understand that considering structures, relations and node types of UML entities, we are able to fully address the challenge of model clone detection, but (sub) graph isomorphism is computationally demanding. Also, the difference between model and other kind of source code is that the latter is written linearly while a model is organized in two dimensions. Our approach is based on token extraction from the model and source code in vector space. Moreover, the applied technique searches for similar concepts based on extracted token matching. It does not consider semantically equivalent concepts where similar concepts are implemented and identifier names are different. A mapping of UML constructs from syntactical to semantic domain may help in detecting clones having same behavior but different syntactical structure. We are extending our current technique by applying formal methods to detect similar concepts implemented in different ways.

Further work in this area is required. A detailed mapping highlighting changes and similarities across versions of similar software will be immensely helpful for software developer. In the area of model clone detection, a comprehensive model clone detection tool for UML models and other data flow languages is missing. Different forms of models

have individual features which need to be exploited for clone detection. We noticed vagueness in the definition of model clones which hinders understanding of the topic area. Different domains perceive conceptual models as a key to good design. We expect to extend our work to other domains by extracting vocabulary terms.

References

[1] Han and M. Kamber, "Data Mining: Concepts and Techniques", 2nd edition, Morgan Kaufmann, **(2006)**.
[2] U. Kelte, J. Wehren and J. Niere, "A generic difference algorithm for UML models", Proceedings of SE 2005, Innsbruck, Austia, **(2005)** February 15-17, pp. 105-116.
[3] T. K. Landauer, P. W. Foltz and D. Laham, "Introduction to latent semantic analysis", Discourse Processes, vol. 25, **(1998)**, pp. 259-284.
[4] J. Liu, J. T. L. Wang, W. Hsu and K. G. Herbert, "XML clustering by principal component analysis", Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence, Boca Raton, FL, USA, **(2004)** November 15-17, pp. 658-662.
[5] G. Guerrini, M. Mesiti and I. Sanz, "An overview of similarity measures for clustering XML documents", Web Data Management Practices: Emerging Techniques and Technologies, A. Vakali and G. Pallis, Eds., PA, USA, IGI Global, **(2007)**, pp. 56-78.
[6] G. Salton and M. J. McGill, "Introduction to Modern Information Retrieval", McGraw Hill, **(1986)**.
[7] M. W. Berry, Z. Drmac and E. R. Jessup, "Matrices, Vector Spaces, and Information Retrieval", Society for Industrial and Applied Mathematics Review, vol. 41, no. 2, **(1999)**, pp. 335-362.
[8] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas and R. Harshman, "Indexing by latent semantic analysis", Journal of the American Society for Information Science, vol. 41, no. 6, **(1990)**, pp. 391–407.
[9] G. D. Clifford, "Singular Value Decomposition & Independent Component Analysis for Blind Source Separation", Biomedical Signal and Image Processing, **(2005)**, pp. 49.
[10] M. Harman, "Why source code analysis and manipulation will always be important", Keynote address at 10th International Working Conference on Source Code Analysis and Manipulation (SCAM'10), Timisoara, Romania, **(2010)** September 12-13.
[11] R. Fanta and V. Rajlich, "Removing Clones from the code", Journal of Software Maintenance: Research and Practice, vol. 11, **(1999)**, pp. 223-243.
[12] D. Rattan, R. Bhatia and M. Singh, "Software Clone detection: A systematic review", Information and Software Technology, vol. 55, no. 7, **(2013)**, pp. 1165-1199.
[13] D. Rattan, R. Bhatia and M. Singh, "Model Clone Detection based on Tree Comparison", Proceedings of Annual IEEE India Conference (INDICON), Kochi, India, **(2012)** December 7-9, pp. 1041-1046.
[14] Y. Sharma, R. Tekchandani and R. Bhatia, "Hybrid Technique for Object Oriented Software clone detection", Master's Thesis, Thapar University, Patiala, India, **(2011)**.
[15] M. Kaur, D. Rattan, R. Bhatia and M. Singh, "Comparison and evaluation of clone detection tools: An experimental approach", CSI Journal of computing, vol. 1, no. 4, **(2012)**, pp. 44-54.
[16] T. Weilkiens, "Systems Engineering with SysML/ UML", Morgan Kaufmann, **(2007)**.
[17] S. Grant and J. R. Cordy, "Vector Space Analysis of Software Clones", Proceedings of 17th IEEE International Conference on Program Comprehension, Vancouver, Canada, **(2009)** May 17-19, pp. 233-237.
[18] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code", Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'01), San Diego, USA, **(2001)** November 26-29, pp. 107-114.
[19] H. A. Basit and S. Jarzabek, "A Data mining approach for detecting higher-level clones in software", IEEE Transactions on Software, vol. 35, no. 4, **(2009)**, pp. 497-514.
[20] DNS Java < http://www.xbill.org/dnsjava/> **(2014)**.
[21] T. Tran, R. Nayak and P. Bruza, "Combining structure and content similarities for XML document clustering", Proceedings of 7th Australasian Data Mining Conference (AuSDM 2008), Glenelg, South Australia, **(2008)**, November, pp. 219-226.
[22] XMI Guide Version 2.4.1. Tech. rep., Object Management Group. http://www.omg.org/spec/XMI, document number formal/ 2011-08-09, **(2011)**.

Authors

Dhavleesh Rattan received his B. Tech. and M. Tech. degrees in Computer Science and Engineering from Punjab Technical University in 2003 and 2008 respectively. He is currently pursuing Ph. D. from Thapar University, Patiala. He is working as assistant professor at Punjabi University, Patiala. His research interests include software clone detection and model driven development. He has many research publications in reputed journals and conferences.

Dr. Rajesh Bhatia obtained his BE in Computer Science and Engineering in 1994, ME in Computer Science in 2001 and Ph. D. in Computer Science and Engineering in 2007 respectively. His research interests are Software Reuse, Software Testing, Software Clone Detection and Component-Based Software Engineering. Presently, he is working as Professor and Head, Computer Science and Engineering at PEC University of Technology, formerly Punjab Engineering College, Chandigarh. He is Certified Software Quality Professional from STQC, Ministry of Information Technology, Govt. of India. He has published about 50 research papers in reputed journals and conferences. He has been associated with various professional bodies such as Senior Member CSI, ACM SIGSE, IEEE, ISTE, IAENG etc.

Maninder Singh received his Bachelor's Degree from Pune University in 1994, and holds a Master's Degree, with honors in Software Engineering from Thapar Institute of Engineering & Technology, as well as a Doctoral Degree specialization in Network Security from Thapar University. He is currently working as Associate Professor in Computer Science and Engineering Department at Thapar University. He is on the Roll-of-honor @ EC-Council USA, being certified as Ethical Hacker (C|EH), Security Analyst (ECSA) and Licensed Penetration Tester (LPT). Dr. Singh has successfully completed many consultancy projects (network auditing and penetration testing) for renowned national bank(s) and corporate. His research interest includes Network Security, Grid Computing and is a strong torchbearer for Open Source Community.