TEMPLATE MINING IN SOURCE-CODE DIGITAL LIBRARIES

Yuhanis Yusof and Omer F. Rana
School of Computer Science, Cardiff University, PO Box 916, Cardiff CF24 3XF, UK
{y.yusof, o.f.rana}@cs.cardiff.ac.uk

Abstract

As a greater number of software developers make their source code available, there is a need to store such open-source applications into a repository, and facilitate search over the repository. The objective of this research is to build a digital library of Java source code, to enable search and selection of source code. We believe that such a digital library will enable better sharing of experience amongst developers, and facilitate reuse of code segments. Information retrieval is often considered to be essential for the success of digital libraries, so they can achieve high level of effectiveness while at the same time affording ease of use to a diverse community of users. Four different matching mechanism: exact, generalization, reduction and nameOnly is used in retrieving Java programs based from information extracted through template mining.

1. Introduction

A Digital Library (DL) may be regarded as a managed collection of information (in digital format) with associated services. These services enable access to this information over a network, and can range from support for selecting and browsing material in the collections, organising and archiving it, and making it available in different visual formats. Often DLs contain a diverse collection of information for use by many different users, and the size of information contained within it can vary. Examples of documents kept in DLs include journal papers, chapters of electronic books and magazines, and product documentation. Various techniques can be used in understanding the content of the DLs – such as classification, association discovery, clustering, visualization of data, information extraction, etc.

Information extraction is the process of capturing structured information from a particular document. Natural language processing techniques can be used to extract data directly from text if either the data, or the text surrounding the data, form recognizable patterns [2]. A key motivation for our work is to facilitate software reuse through information extraction, whereby a software engineer or programmer could make use of existing software packages to create new programs. Software reuse has been shown through empirical studies to improve both the quality and productivity of software development. Our thesis is that software reuse should not just be restricted to reusing software libraries in their entirety, but also enable software developers to understand the process associated with solving a problem encoded in the software library. A software developer may be interested in understanding how a particular feature has been coded in a particular language – rather than perhaps make full use of code that has been implemented by someone else.

Recent efforts in object oriented programming (such as the Common Object Request Broker Architecture (CORBA), and recently Web Services) indicate the significance of writing software as independent services to enable reuse. As software reuse is becoming more important, there is a need to store open-source applications in the format of a searchable repository. Such a repository will store all documents related to an application, which includes the source code, comments in versioning systems (such as CVS), documentation provided with the source code (such as a Javadoc document with Java source code), and details about the structure of the library (such as a class hierarchy), etc.

2. Related Work

There is limited literature in the area of applying template mining for extracting information: Cowie and Lehnert [3] have successfully extracted proper names from documents. Extraction of facts from press releases within a company's financial information system has also been undertaken in a few systems, such as ARTANS [9], JASPER [10] and FIES [1]. It has also been proven that template mining is able to build an abstract of scientific papers [8] and also the extraction of citation from digital documents [4]. Four different template are utilised, one for extracting information about articles, while the others are used for extracting information from citations.

Even though there has been much research done into extracting information from text documents, a significant effort has not been put into extracting information from software packages – especially program source code. Most of the research done in the area of understanding source code is mainly focused in categorizing the programming language used into a particular software component or source code achieve [11]. Ugurel et al. [13] classified source code into appropriate application domains and also programming languages using three components, namely the feature extractor, vectorizer and Support Vector Machine classifier. Paul and Prakash [12] have produced a framework which uses pattern languages to specify interesting code features. The pattern languages were derived by extending the source programming language with pattern-matching symbols. They transformed the source code into specific symbols by including a set of symbols that can be used to substitute syntactic entities in the programming language. In this paper, we discuss the usage of template mining in retrieving relevant Java programs stored in software packages, within a Digital Library of source code.

A related area that has been investigated by others is the submission of queries to retrieve particular source code by name. One example of this is the retrieval of particular numerical algorithms via email. Dongarra and Grosse [5] demonstrate this with reference to their Netlib Digital Library. Our approach differs from this in that we are interested in a variety of search techniques – and not just an exact match. Furthermore, many such approaches are restricted to particular types of applications (numerical algorithms in this case), and therefore are restricted in their scope. We also see limited use of existing search engines for this particular problem, as search engines such as `Google.com` or `Altavista.com` do not provide any support for formulating a query based on program structure. Therefore template search mechanisms provide users, especially programmers, with specific search capabilities without neglecting the use of keyword search as offered by Unix utilities like `grep`.

3. Java Template Miner

In developing our Template Mining approach, we are making the following assumptions:

- Users have some indication of the types of source code they are interested in. This could be in terms of the keywords they assume to be present within such source code, or the likely method names that such source code could contain. Although not likely to be valid in a general case, we have found this assumption to hold true based on the existing source code archives such as `Sourceforge.net`. Perhaps one reason for this is that developers who offer their source code for use by others often also attempt to describe their data structures or method names with comments that could be relevant for others.
- Users are familiar with the likely structure of the source code they are trying to find. This may be particularly true for numerical approaches (where nested loops are often used over arrays or similar data structures). Often many programming languages targeted towards the scientific computing community provide specialist support for such data structures (examples include OpenMP and High Performance Fortran).

A prototype system is currently under development – and primarily focused at Java developers. The general architecture of the prototype is described in Figure 1.

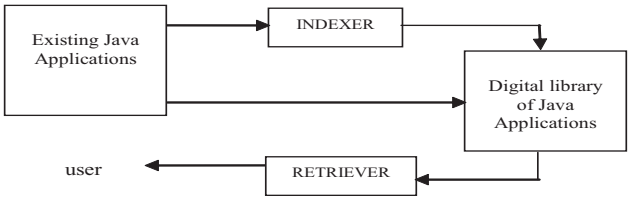

Figure 1. General architecture of source-code digital libraries

Two main components are included in the architecture, the INDEXER and the RETRIEVER. Existing Java applications are fed into the INDEXER, which will finally produce two text files: *index.txt* and *indexMethod.txt* file. Each source code file is analyzed individually, and is used to build a set of keywords. The index file now contains relevant keywords which represent each of the source files contained in the archive - and divided into the application from which the source file has been obtained. The index file thus establishes the rules to generate the internal representation of both the queries and content of software packages.

An index I is defined as a set of terms from each software package or user query: $I =< S_1, S_2, ...S_n >$, where S_i is a set of terms obtained from a software package, and P_j is the number of software packages in the repository. Therefore, $S_i \subseteq P_j$, such that $S_i = \{t_{i1}, t_{i2}, ..., t_{ik}\}$, where $t_{ij} \in P_j$, $(1 \leq j \leq k)$, $(k \leq \mid P_j \mid)$.

On the other hand, RETRIEVER extracts relevant Java source code from the DL that fully or partially matches the requested queries posed by a user. In general, it is defined as follows: let *C* be the set of all possible portions of Java source code for a given query. Let *A* be the set of all available portions of Java source code in the repository. Let $f : C \rightarrow PA$ where *PA* is the power set of *A*. Given a target query: $I_t =< S_1, S_2, ...S_n >$ the function(*f*) returns a set of Java programs from *A* that are the same or similar to I_t. We may specify the function as: $f(I_t) = A'$, where

$A' \subseteq A$, and $A' = \{I_{x1}, I_{x2}, ..., I_{xy}\}$, where $I_{xi} \in A$, and $(0 \leq y \leq \mid A \mid)$.

Users are given two choices, either to use the keyword/phrase query or the program template query. Here we will only be focusing on retrieving relevant Java files based on a program template. Several different types of information are automatically generated from the template query which will be mapped against the index files. In-

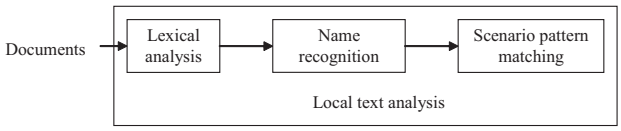

Figure 2. Structure of an information extraction system

formation extraction in this context involves two stages: extracting individual facts from the text, and integrating these facts to create a coherent understanding of the program. In doing so, we follow some elements contained in the structure of an information extraction system [7]. From Figure 2, lexical analysis is done by dividing the plain text into tokens which will then be used to treat each word separately. Each tokens is then referenced to a Java dictionary to determining its usage. For example token of `import, public, static, class` etc are recognized and accepted, while tokens containing `the, method, processing` etc are not. Following the analysis is the name recognition process where class name and method signature are analysed. Finally, pattern matching is implemented to integrate both facts captured through lexical analysis and name analysis. An example of a program template used in our experiment is provided below:

```
public class Matrix {
public Matrix getMatrix (int a, int b, int[]c) {
;
// body of the method //
}
public Matrix transpose () { }
public Matrix plus (int B) { }
public Matrix minus (double B) { }
}
```

Mining from the above template, several information can be extracted and mapped against the *index.txt* and *indexMethod.txt* files. These two files were initially created based on the software packages contained in the source code repository. They contain keywords captured from all text documents and source code programs of the repository. These include the class and method signatures which can then be used to retrieve similar Java programs from the repository. Class name is recognized by the Java keyword `class` while method signatures are recognized by three identifier: name, return type and the first parameter (if it exists). If there are no parameters provided in a particular method, the system will automatically replace it with the string 'none'. Currently, this system only recognizes seven main data types: `String, char, Boolean, int, double, float, long, byte` and `short`. If any other data types were found, the system will then automatically replace it with 'none'.

Based on these, the search process is undertaken using four different methods (described in more detail below): (1) exact matching, (2) generalization, (3) reduction, and nameOnly. In the exact matching procedure, class name and method signature are compared. If the system fails to find an exact match (equivalent to a Boolean AND) where all requirements (class and method signature) are successfully matched, it will then use the generalization method. In this method, the system will change some of the parameters in the method signature. For example, consider a Program Template:

```
public Matrix minus(double B) -> public Matrix
minus (float B)
```

where the data type double for argument B will be changed to data type float. Java files will be retrieved only if it matches the method name and either the return type or the first parameter. Alternatively, the search mechanism will retrieve Java programs based on the reduction technique (based on a Boolean OR) during the matching process. If either the return type or the first parameter is matched then the program is considered relevant to the query. Compared to the previous matching mechanism, reduction matching does not replace any data types before processing. Finally, the search mechanism will retrieve all Java programs which have the same method name while ignoring its signature. The final outcome of the template mining in source-code digital libraries is a list of Java programs, and this list is returned in a text file – *resultTemplate.txt*. These four approaches can be used as a first step in selecting suitable programs from an archive – with increasing degree of closeness to a template.

4. Result Analysis

Currently a Java repository of 114 MB has been built by storing 30 Java packages in it. Two index file are generated initially before template mining is done: *index.txt* and *indexMethod.txt*. `Table 1` shows two factors (time and file size) being observed during the process of generating index files and undertaking template mining in the Java source-code repository.

Table 1. Time and file size of generated index files

FILE (name)	TIME (ms)	FILE SIZE (bytes)
Index.txt	3140144	110590248
IndexMethod.txt	89591	2365555
resultTemplate.txt	7703	573

The processing time and file size of *index.txt* is longer and bigger as it contains all relevant words captured from the Java repositories. On the other hand, *indexmethod.txt*, only contains relevant keyword for method signatures. On completing the template mining and search process, the result is written into file *resultTemplate.txt*. Based on the program template used in the experiment (Section 3), the processing time taken to find and retrieve relevant Java programs is 7703 millisecond.

A precision and recall analysis is undertaken for comparing the effectiveness of source-code retrieval (`Table 2`). According to Frakes and Baeza-Yates [6], "Recall" is defined as the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in a repository. Similarly, "Precision" is defined as the ratio of the number of documents retrieved that were found to be relevant (as identified by a human expert) over the total number of documents retrieved from the repository via a given query. We use the same definition here for tabulating results of our experiment based on the program template mentioned in Section 3. Where the above definition says documents and database, the word Java programs and repository respectively can be substituted. We similarly adapt some of the other concepts identified when defining Recall and Precision for text documents. When evaluating Recall, we therefore investigate that out of all the relevant Java files in the repository, how many did our search actually retrieve? For evaluating Precision, we determine out of all the Java files that were extracted during a search, how many were relevant. According to Frakes [6], "both Recall and Precision take on values between 0 and 1" (or between 0 and 100 when expressed as percentages). `Table 2` presents the recall and precision figures for each search performed normalized to lie between 0 and 100.

Each of the extracted information (class name and method) is analyzed separately for an exact string and a sub-string match. Exact string matching is used to retrieve programs with the exact class name or method name, while sub-string matching is used to retrieve programs with the target string being a part of one string or if there exist any string as a substring of the target string. For example, if a user is searching for `Matrix.Java`, the file name `createMatrix.java` will also be retrieved through sub-string matching. Let M_i be the method name in the index file, and M_q be the target (query) method name. If the length of target string is less than the index term, we will accept Java programs which produces $(0 \leq (M_i.length - M_q.length) \leq x)$, otherwise the accepted string matching must fulfill $(0 \leq (M_q.length - M_i.length) \leq y)$. The values of x and y are chosen to be 10 and 5 respectively – these are heuristics based on our current data set.

Table 2. Precision and recall analysis of sub-string matching

Class or Method	SUBSTRING	
	Precision	Recall
Matrix	100%	80%
Matrix getMatrix(int)		
Exact matching	100%	100%
Generalization	-	-
Reduction	0%	-
nameOnly	100%	100%
Matrix transpose()		
Exact matching	100%	100%
Generalization	-	-
Reduction	0%	0%
nameOnly	100%	80%
Matrix plus(int)		
Exact matching	-	-
Generalization	-	-
Reduction	100%	50%
nameOnly	100%	50%
Matrix minus(Matrix)		
Exact matching	0%	0%
Generalization	-	-
Reduction	100%	33.33%
nameOnly	100%	33.33%

From the result, retrieved documents based on string matching produced the result of 100% for both precision and recall analysis. However, this does not happen in sub-string matching where recall analysis produces an average of 64.92%. The issue here is whether we would like to consider sub-strings within a sub-string to also be relevant during the search process. Although we see benefits of undertaking such an analysis in particular instances, the general benefit of this approach is not obvious to us. In this case, recall analysis would be 0% as the system is not able to produce any result. The current system is not able to extract sub-strings contained within a string automatically, and search for these as query terms. As long as the Java program (source string) contains the searched string (ignoring the sub-string length and location of each element), currently we still consider it as a precise retrieval. This leads us to obtain 100% precision as all the retrieved documents are found to be relevant to the query. Meanwhile, as for recall analysis, the system fails to retrieve some relevant java programs which then produced an inconsistent recall percentage. (`Table 2`) also contains elements of '-' as the answer set for the query is 0 and therefore, we could not calculate the percentage. From four retrieval techniques mentioned

in Section 3, retrieved Java files are categorized as follows:

1. matching of class name and all methods signature
2. matching of class name and all methods signature (generalization)
3. matching of class name and all methods signature (reduction)
4. matching of class name and all methods signature (nameOnly)
5. matching of class name only
6. matching of method signature only

Each retrieved document is being given different points if it is classified under the above categories. The values are then summed over all the categories to obtain a ranking for the retrieved files (in terms of their relevance to the initial query). Documents with the highest value of points will be ranked as the most relevant program to the program template defined by a user.

Comparing this process to a software repository such as `Tucows.com` where users are presented with software to be downloaded (in different categories), this system presents Java source code to help users (especially programmers and system analysts) in reusing existing software libraries. Similarly, in `SourceForge.net`, open-source applications are catalogued based on their particular categories, such as Programming Language and Operating System, etc. The search process utilised in `SourgeForge` makes use of keywords, and is based on general descriptions given to each of the stored packages. In other hand, both `SourceForge` and `Tucows` do not allow any source code retrieval based on users query. We see our approach as an obvious extension of the search process supported by such public domain software repositories – and our current test set is based on a subset of source code obtained from `SourceForge`.

5. Conclusion and Future Work

With the emerging interest in making source code available, and the significant emphasis being placed on this by many software architects, DLs that support the searching for source code have become necessary. The importance of the reuse of software artifacts has been discussed as a key motivator for the implementation of such source-code digital libraries. A system for supporting this is identified, to support users in locating Java source code. Relevant information is extracted from a program template provided by the user and this is compared against index files generated from software packages stored in the repository.

Currently, as a continuity to the research, we are extracting other information from the repository which includes class hierarchies and associations between different classes within a software. This is being achieved by automatically developing a Unified Modelling Language (UML) diagram (using reverse engineering tools), and by automatic extraction of design patterns from the source code. With this, users will have more choices in creating their query, and specifying their search criteria. Relevant programs will then be presented to the user ranked according to their relevance of both exact/inexact keyword search, and relevance based on program structure. We also intend to make better use of software categories in existing source code archives, and include these with the approaches mentioned above.

References

[1] W. Chong and A. Goh. Fies:financial information extraction system. *Information Services and Use*, 17(4):215–223, 1997.

[2] G. Chowdury, N. Kemp, M. Lawson, and M. F. Lynch. Automatic extraction of citations from the text of english-language patents - an example of template mining. *Information Science*, 22(6):423–436, 1996.

[3] J. Cowie and W. Lehnert. Information extraction. *Commun. ACM*, 39(1):80–91, 1996.

[4] Y. Ding, G. Chowdhury, and S. Foo. Template mining for the extraction of citation from digital documents. *Library Trends*, 48(1):181–207, 1999.

[5] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, 1987.

[6] W. B. Frakes and R. Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc., 1992.

[7] R. Grishman. Information extraction: techniques and challenges. In M. T. Pazienza, editor, *Information Extrcation*, Springer-Verlag Lecture Notes in AI, pages 10–27, Rome, 1997.

[8] P. Jones and D. Chris. A 'select and generate' approach to automatic abstracting. In T.McEnery and C.D.Paice, editors, *Proceedings of the BCS 14th Information Retrieval Colloqium*. Springer-Verlag, 1992.

[9] S. Lytinen and A. Gershman. Atrans:automatic processing of money transfer messages. In *Proceedings of Fifth National Conference on Artificial Intelligence*, pages 1089–1093, Los Altos, CA, 1986. Morgan Kaufmann.

[10] M. A. Peggy, J. H. Philip, K. H. Alison, M. S. Linda, B. N. Irene, and P. W. Steven. Automatic extraction of facts from press releases to generate news stories. In *Proceedings of the 3rd Conference on Applied Natural Language Processing*, pages 170–177, Trento, Italy, 1992.

[11] P. Ruben and F. Peter. Classifying software reuse. *IEEE Software*, 4(1):616, 1987.

[12] P. Santanul and P. Atul. A framework for source code search using program patterns. *IEEE Transaction on Software Engineering*, 20(6):463–475, 1994.

[13] S. Ugurel, R. Krovetz, and C. L. Giles. What's the code?: automatic classification of source code archives. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638. ACM Press, 2002.