JAVA APPLICATIONS COMPREHENSION BY DISCOVERING CLUSTERS OF RELATED CLASSES

Jauhar Ali
College of Engineering and Computer Science
Abu Dhabi University
Abu Dhabi, UAE
jauhar.ali@adu.ac.ae

ABSTRACT

Software maintenance is an important phase of software lifecycle consuming the majority of effort. Program comprehension is the most time consuming activity during software maintenance. Data mining techniques have been used to support program comprehension for large software systems. This paper presents an approach to extract useful knowledge from Java source/byte code, and apply clustering to discover groups of closely related classes. The groups of classes can assist programmers to know the high level structure of large software systems without prior knowledge, and programmers can learn the classes in the same group together. The groups of classes are formed based on different types of coupling between classes, and having common characteristics, such as having the same superclass. A prototype system has been developed and evaluated using a medium sized Java application.

KEYWORDS

Source code mining, program comprehension, clustering, Java, data mining

1. INTRODUCTION

In software lifecycle, *software maintenance* is considered a very important phase typically consuming 50-70% of the total effort allocated to a software system [1], [2]. *Program comprehension* is an important part of this phase, especially when the program is complex and documentation is not up to date. Software maintenance engineers spent 50-90% of their time on program comprehension [3]. *Software reuse* is a common technique which attempts to save time and energy by reducing redundant work. It is one of the goals of object-oriented technology and is the reason for the existence of software libraries [4], [5]. Program comprehension plays a very important role in software reuse as without understanding the functionality of a program, it cannot be reused effectively.

The purpose of this work is to assist programmers in comprehending the structure of a software system. We present an approach to apply clustering on class information extracted from Java source/byte code. The clusters of classes can support programmers to know related classes, thus helping them comprehending the modules and functionality of the system.

The rest of the paper is organized as follows: Section 2 reviews research work on using data mining techniques for software comprehension. In Section 3, we present our approach of extracting useful knowledge from Java source/byte code. Section 4 explains our prototype tool that is used as a “proof of concept” and the results of a case study. Finally, conclusions and directions for future work are presented in Section 5.

2. BACKGROUND

Data mining can produce high level overviews of source code and interrelationships among program components thus facilitating software systems understanding [6], [7]. It is considered a suitable solution in assisting program comprehension, often resulting in remarkable results [8], [9], [10], [11].

Clustering is one of the well-known and well-studied techniques of data mining [12]. It does not require prior knowledge of possible groups to which the objects under study belong, thus making it suitable for discovering groups of related entities in a software system without prior knowledge. Clustering as a means of supporting software comprehension and maintenance has been used for software systems developed in different programming languages and addressing varying levels of abstractions [7], [9], [10], [11].

3. PROPOSED APPROACH

Our aim is to apply clustering to find groups of similar classes in a Java application. These groups of classes can assist a programmer to have an overview of the whole system and to comprehend the classes in the same group together. To achieve this goal, we need to (1) define an appropriate data model and metrics, (2) use proper data-extraction technique, and (3) apply a suitable clustering algorithm. In the following subsections, we explain these steps.

3.1. Data Model and Metrics

To be able to apply clustering to classes in a Java application, we need to know certain attributes for each class in the application. Due to the code structure of Java programs, we believe that the following attributes are important to find the *similarity* or "closeness" between any two classes:

A1) *Name of source file*, as classes are close to each other if they are defined in the same source file.

A2) *Name of package*, as two classes in the same package should be close to each other.

A3) *Name of superclass*, as a common superclass will suggest that the two classes are close to each other.

A4) *Implementing same interface?* as classes implementing the same interface will behave very similarly.

The first three attributes are *nominal* and the fourth one is *boolean*. In addition, we add the following attributes that represent different types of *coupling*, as tight coupling suggests dependencies between classes [13].

A5) *Variable type coupling*: If class *i* has a variable of class *j* type, there is a variable type coupling from class *i* to class *j*. It is calculated by $vc_{ij} = \frac{vn_{ij}}{vn_i}$, where vn_{ij} is the number of variables of type *j* defined in class *i*, and vn_i is the total number of variables defined in class *i* whose type is some other class in the application.

A6) *Parameter type coupling*: If some method in class *i* has a parameter of class *j* type, there is a parameter type coupling from class *i* to class *j*. It is calculated by $pc_{ij} = \frac{pn_{ij}}{pn_i}$, where pn_{ij} is the number of parameters of type *j* defined in methods of class *i*, and pn_i is the total number of parameters defined in the methods of class *i* whose type is some other class in the application.

A7) *Method return type coupling*: If class *i* has a method whose return type is class *j*, there is a method return type coupling from class *i* to class *j*. It is calculated by $mrc_{ij} = \frac{mrn_{ij}}{mrn_i}$, where mrn_{ij} is the number of methods defined in class *i* whose return type is class *j*, and mrn_i is the total number of methods defined in class *i* whose return type is some other class in the application.

A8) *Method call type coupling*: If some method in class *i* calls a method in class *j*, there is a method call type coupling from class *i* to class *j*. It is calculated by $mcc_{ij} = \frac{mcn_{ij}}{mcn_i}$, where mcn_{ij} is the number of method-calls made in class *i* to methods in class *j*, and mcn_i is the total number of method-calls made in class *i* to methods in other classes in the application.

All the attributes that represent couplings (A5 to A8) are numeric and their values will always be in the range of 0 to 1. A value close to zero will indicate loose or no coupling and a value close to 1 will indicate tight coupling.

3.2. Data Extraction

The values for attributes described in the previous subsection need to be extracted from Java application under study. We choose to use SOOT [14], [15] for this purpose. SOOT is a freely available powerful framework and its main focus is code optimization. However, it converts Java code to a very simple Intermediate Representation called Jimple. In Jimple, there are only 15 different types of statements compared to around 200 types of statements at byte code level. It also gives the flexibility to extract data from either Java source code or Java byte code. It is a big advantage, as sometimes, programmers need to understand the structure of an application whose source code is not available. We can easily extract all the required data (described in the previous subsection) from any Java application using the SOOT framework.

3.3. Clustering

For clustering, we use the Hierarchical Agglomerative algorithm [12] because it does not need the number of desired clusters to be specified. This algorithm produces sets of clusters in order of decreasing similarity. The algorithm requires that all non-numerical values be transformed to numerical, so that the distance among entities can be measured and stored in a *dissimilarity matrix* [12]. Each attribute of this matrix is going to be assigned a numerical value. This numerical value is the distance between two records of the table of the database. In our case, each record represents a Java class.

The distance *d(i, j)* between two objects can range between 0 (being the nearest) and 1 (being the value that corresponds to the farthest distance). So, $0 \le d(i, j) \le 1$. The distance is calculated by using the following equation.

$$d(i,j) = \frac{\sum_{f=1}^{p} \delta_{ij}^{(f)} d_{ij}^{(f)}}{\sum_{f=1}^{p} \delta_{ij}^{(f)}}$$

Equation 1: Distance between two objects

where $\delta_{ij}^{(f)} = 0$ if there is no value of attribute *f* for object *i* or object *j*; otherwise, $\delta_{ij}^{(f)} = 1$. The contribution of attribute *f* to the dissimilarity between *i* and *j*, that is, $d_{ij}^{(f)}$, is computed dependent on its type.

- If *f* is numerical : $d_{ij}^{(f)} = \dfrac{|x_{if} - x_{jf}|}{\max_f - \min_f}$.
- If *f* is nominal/binary: $d_{ij}^{(f)} = 0$ if $x_{if} = x_{jf}$; otherwise $d_{ij}^{(f)} = 1$.

In our case, attributes A1 to A4 are nominal and for them we use the above computation as it is. The remaining attributes, which show coupling of different types between classes, are numerical. However, we need to modify the above computation properly for them. The reason is that a low value (close to 0) for coupling from class i to class j, and a low value from class j to class i, when combined together, should be translated to a high value (close to 1) for $d_{ij}^{(f)}$, as low coupling means higher distance. Similarly, high values for the same coupling from both sides, when combined, should be translated to a low value (close to 0) for $d_{ij}^{(f)}$. Also, no attribute will have missing value as all the attribute values are computed. So we modify Equation 1 to the following.

$$d(i,j) = \frac{\sum_{f=1}^{p} d_{ij}^{(f)}}{8}$$

Equation 2: Distance between two classes

In this equation, the denominator 8 represents the number of attributes, and $d_{ij}^{(f)}$ represents the distance between class *i* and *j* with respect to attribute *f*.

For attributes that show coupling, $d_{ij}^{(f)} = \frac{2-(d_{i\to j}^{(f)} + d_{j\to i}^{(f)})}{2}$, where $d_{i\to j}^{(f)}$ is the coupling from class *i* to class *j*. For example, for attribute A5 (*Variable type coupling*), $d_{i\to j}^{(f)} = vc_{ij} = \frac{vn_{ij}}{vn_i}$.

4. PROOF OF CONCEPT

To evaluate our approach we implemented a prototype tool. The process used in the tool is shown in Figure 1. The tool has the following parts:

- Pre-processing Pass 1 (PP1): This part converts the Java source code or byte code into Jimple representation with the help of SOOT [14]. From Jimple code, PP1 retrieves the following data for each class:

    1. Class name
    2. Source file name
    3. Package name
    4. Super-class name
    5. Names of implemented interfaces (as a list)
    6. Types of non-primitive variables (as a list)
    7. Types of non-primitive parameters of all methods (as a list)
    8. Return-types of all non-void and non-primitive methods (as a list)
    9. Names of methods and their enclosing classes, which are called from the current class' methods (as a list)

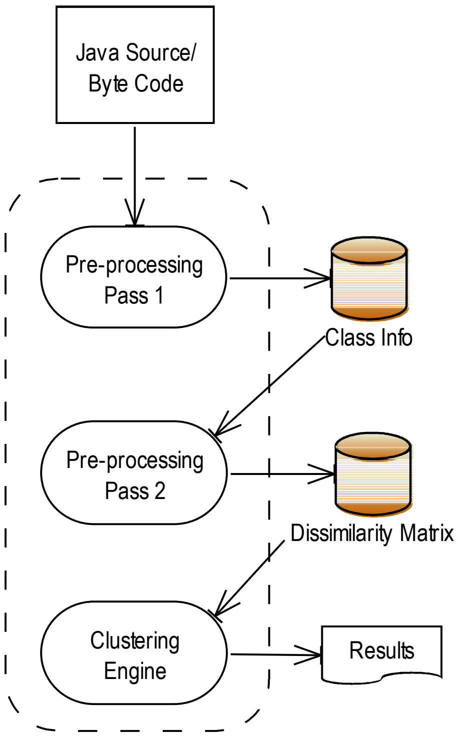

Figure 1: Proof of concept tool

- Pre-processing Pass 2 (PP2): This part creates a *dissimilarity matrix* [12] using the data retrieved in PP1. A dissimilarity matrix for *n* classes is represented by an *n*-by-*n* table where *d(i,j)* is the measured *difference* or *dissimilarity* between classes *i* and *j* (Figure 2). The calculation of *d(i,j)* is based on Equation 2.

- Clustering Engine (CE): This part takes the dissimilarity matrix produced by PP2 as input and applies the Hierarchical Agglomerative Clustering (HAC) algorithm [12] to find clusters of closely related classes. The results are presented as graphs, as well as text.

$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & \vdots & \\ d(n,1) & d(n,2) & \ldots & \ldots & 0 \end{bmatrix}$$

Figure 2: Dissimilarity matrix

4.1. Case Study

Our tool groups Java classes that are closely related to each other. To evaluate whether the groupings of classes found by the tool are close to the groupings of classes intended by the original developers, we used one of our previous medium-size Java application, called Mudrik [16]. The comparison results were very encouraging. Further case studies of relatively bigger applications are needed.

5. CONCLUSIONS AND FUTURE WORK

We proposed an approach to extract useful knowledge from Java source/byte code and apply clustering to group Java classes that are close to each other. The grouping of classes can assist programmers in comprehending large Java systems developed by others. This can save considerable time usually spent by maintenance programmers, especially when documentation of the system is not up to date.

As future work, we plan to do the followings:

- Consider other object-oriented metrics which may give better groupings of classes.
- Use other clustering algorithms which may possibly give better results.
- Evaluate our approach by using more case studies. We intend to apply our approach on some large open source system, such as JBoss [17] and OpenJDK [18].

REFERENCES

[1] Pigoski, T. M.: Practical Software Maintenance: Best Practices for Managing Your Software Investment. John Wiley, (1997).

[2] Sommerville, I.: Software Engineering, 9th ed. Addison Wesley, (2011).

[3] Tjortjis, C., Layzell, P.J.: Expert Maintainers' Strategies and Needs when Understanding Software: A Qualitative Empirical Study. In Proc. IEEE 8th Asia-Pacific Software Engineering Conf. (APSEC 2001), IEEE Comp. Soc. Press, pp. 281-287, (2001).

[4] Frakes, W.B., Kang, K.: Software Reuse Research: Status and Future. IEEE Transactions on Software Engineering, 31(7), pp. 529-536, (2005).

[5] Code reuse, http://en.wikipedia.org/wiki/Code_reuse. Retrieved on December 27, 2012.

[6] Sartipi, K.., Kontogiannis, K., Mavaddat, F.: Architectural Design Recovery Using Data Mining Techniques. In Proc. 2nd European Working Conf. Software Maintenance Reengineering (CSMR 00), IEEE Comp. Soc. Press, pp. 129-140, (2000).

[7] Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y., Gansner, E.R.: Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In Proc. 6th Int'l Workshop Program Understanding (IWPC 98), IEEE Comp. Soc. Press, pp. 45-53, (1998).

[8] Oca, C. M., Carver, D. L.: Identification of Data Cohesive Subsystems Using Data Mining Techniques. In Proc. Int'l Conf. Software Maintenance (ICSM 98), pp. 16-23, (1998).

[9] Kanellopoulos, Y., Dimopulos, T., Tjortjis, C., Makris, C.: Mining Source Code Elements for Comprehending Object-Oriented Systems and Evaluating Their Maintainability. SIGKDD Explorations, Vol. 8, No. 1, pp. 33-40, (2006).

[10] Xiao, C., Tzerpos, V.: Software Clustering on Dynamic Dependencies. In Proc. IEEE 9th European Conf. Software Maintenance Reengineering (CSMR 05), pp. 124-133, (2005).

[11] Zhong, S., Khoshgoftaar, T. M., Seliya, N.: Analyzing Software Measurement Data with Clustering Techniques. IEEE Intellegent Systems, Vol. 19, No. 2, pp. 20-27, (2004).

[12] Han, J., Kamber, M.: Data Mining Concepts and Techniques, 3rd Ed. Mourgan Kaufmann Publishers, (2011).

[13] Chidamber, S. R., Kemerer, C. F.: A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, Vol.20, No.6, pp.476-493, (1994).

[14] SOOT: A Java optimization framework. http://www.sable.mcgill.ca/soot/. Retrieved on December 29, 2012.

[15] Lam, P., Bodden, E., Lhotak, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. CETUS'11, (2011).

[16] Ali J.: Cognitive support through visualization and focus specification for understanding large class libraries. Journal of Visual Languages and Computing, 20, pp. 50-59, (2009).

[17] JBoss, http://www.jboss.org. Retrieved on December 30, 2012.

[18] OpenJDK, http://openjdk.java.net. Retrieved on December 30, 2012.