Class Clustering for Program Comprehension: A Case Study in Java

Jauhar Ali
College of Engineering
Abu Dhabi University
Abu Dhabi, United Arab Emirates
Email: jauhar.ali {at} adu.ac.ae

Abstract—**Program comprehension is the most time consuming activity during software maintenance. Programmers need support to help them in understanding large software systems. This paper presents an approach to extract useful knowledge from Java byte code, and apply hierarchical 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 distance among classes is determined by considering their organizational structure and various kinds of couplings. A prototype system has been developed and evaluated using a medium and a large sized Java application.**

Keywords- Program comprehension, hieararchical clustering, Java, data mining

I. INTRODUCTION

Software maintenance is considered a very important phase in software lifecycle 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 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 II reviews the work done on using data mining techniques for software comprehension. In Section III, we present our approach of extracting useful knowledge from Java byte code. Section IV explains our prototype tool. Section V presents the results of a case study using the prototype. Finally, conclusions and directions for future work are given in Section VI.

II. 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].

III. PROPOSED APPROACH

We have developed a prototype system in which we apply clustering to find groups of closely related 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 apply clustering, 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.

A. *Data Model and Metrics*

To be able to apply clustering to classes in a Java application, we need to have a metrics for determining distance between any two classes. The classes having small distance are considered as closely related classes compared to those having large distances. The distance is calculated by considering a number of aspects. Each aspect contributes to the overall distance between the classes. The distance determined by considering a particular aspect is in the range of 0 and 1, where a distance close to 0 means closely related and a distance of 1 means not related. All these component distances are weighted and summed up to calculate the overall distance between any two classes. We consider the following component distances

for determining the overall distance between two classes. Components 1 to 4 are based on the organizational structure of classes while the remaining components are based on various kinds of *coupling* among classes [13].

1) Package Distance: If both classes belong to the same Java package they are closely related. Package Distance of class A and B (PkD_{AB}) is 0 if they belong to the same package, otherwise it is 1.

2) Inheritance Distance: When classes have inheritance relationship they are closer to each other, compared to other classes. If one class is a direct subclass of the other, their inheritance distance (IhD_{AB}) is 0; if one is an indirect subclass of the other, IhD_{AB} is 0.5; in all other cases IhD_{AB} is 1.

3) Subling Distance: If both classes are subclassed from the same superclass, their subling distance (SbD_{AB}) is 0, otherwise it is 1.

4) Interface Distance: When two classes implement the same interface(s), they both have impementations for all the methods declared in the interface(s). Such classes are closer to each other as they have somewhat similar behavior. If two classes do not implement the same interface, their interface distance (ItD_{AB}) is 1, otherwise it is calculated as below.

$$ItD_{AB} = 1 - \frac{nmI}{nmA + nmB}$$

Equation 1: Interface distance betweeen class A and B

Where *nmA* and *nmB* are the number of methods in class *A* and *B* respectively, and *nmI* is the number of methods in the interface(s) implemented by both class *A* and *B*.

5) Field Distance: If class *A* has a field (instance variable) whose type is class *B*, there is a coupling between the two classes. Such coupling becomes stronger when we have more of such fields in the two classes. The field distance between classes A and B (FdD_{AB}) is the average of the field distance from A to B ($FdD_{A \to B}$) and from B to A ($FdD_{B \to A}$).

$$FdD_{AB} = \frac{FdD_{A \to B} + FdD_{B \to A}}{2}$$

$$FdD_{A \to B} = \frac{nfAB}{nfA}, FdD_{B \to A} = \frac{nfBA}{nfB}$$

Equation 2: Field distance between classes A and B

Where $FdD_{A \to B}$ represents field coupling from class *A* to class *B*; $nfAB$ is the number of fields in class *A* whose type is class *B*; and nfA is the total number of fields in class *A* whose type is any other class.

6) Parameter Distance: If class *A* has a constructor's or method's parameter whose type is class *B*, there is a coupling from class A to class B. The parameter distance (PmD_{AB}) of two classes is calculated as below.

$$PmD_{AB} = \frac{PmD_{A \to B} + PmD_{B \to A}}{2}$$

$$PmD_{A \to B} = \frac{npAB}{npA}, PmD_{B \to A} = \frac{npBA}{npB}$$

Equation 3: Parameter distance between classes A and B

Where $PmD_{A \to B}$ represents parameter coupling from class *A* to class *B*; $npAB$ is the number of parameters in class *A* whose type is class *B*; and npA is the total number of parameters in class *A* whose type is any other class.

7) Method Distance: If class *A* has a method which returns data of type *B*, there is a coupling from class A to class B. The method distance (MtD_{AB}) of two classes is calculated as below.

$$MtD_{AB} = \frac{MtD_{A \to B} + MtD_{B \to A}}{2}$$

$$MtD_{A \to B} = \frac{nmAB}{nmA}, MtD_{B \to A} = \frac{nmBA}{nmB}$$

Equation 4: Method distance between classes A and B

Where $MtD_{A \to B}$ represents method coupling from class *A* to class *B*; $nmAB$ is the number of methods in class *A* whose return data type is class *B*; and nmA is the total number of methods in class *A* whose return data type is any other class.

Overall Distance: The overall distance between two classes is calculated by summing up all the above seven component distances. Before summing up, each component distance is multiplied by a proper weight that indicates the importance of the component in the overall calculation, as below.

$$D_{AB} = PkD_{AB} \times PkW + IhD_{AB} \times IhW + SbD_{AB} \times SbW + ItD_{AB} \times ItW + FdD_{AB} \times FdW + PmD_{AB} \times PmW + MtD_{AB} \times MtW$$

Equation 5: Overall distance between classes A and B

B. Data Extraction

To calculate the values for different kinds of distances discussed in the previous subsection, we need to extract all relevant data from the classes of the system under study. The data includes the following.

- Names, packages and super classes of all classes
- Interfaces implemented by any class and the methods declared in those interfaces
- Fields, methods' parameters, and methods' return types of all classes

The above data can be extracted from the source code, but we choose to retrieve the data from the Java classes' byte codes using the Java reflection facilities. The main advantage of this approach is that compiled Java systems without source code and having only byte code can also be analyzed by our system.

C. 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 the distance among classes 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 Java classes and is calculated by using Equation 5.

IV. PROTOTYPE

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:

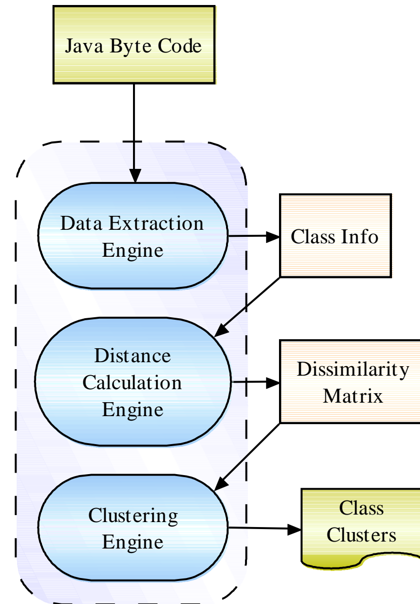

Figure 1: Prototype tool parts and dataflow

Data Extraction Engine (DEE): This part loads the Java classes of the target system from Java class files or JAR file (byte code) and then uses the Java reflection facilities (java.lang.reflect.*) [14] to retrieve the data needed for calculating distances among classes as described in the previous section.

Distance Calculation Engine (DCE): This part creates a dissimilarity matrix [12] using the data retrieved by DEE. A dissimilarity matrix for n classes is represented by an n-by-n table where D_{AB} is the distance or dissimilarity between classes *A* and *B* (**Error! Reference source not found.**). The calculation of D_{AB} is based on Equation 5.

$$\begin{bmatrix} 0 & D_{12} & D_{13} & \dots & D_{1n} \\ D_{21} & 0 & D_{23} & \dots & D_{2n} \\ D_{31} & D_{32} & 0 & \dots & D_{3n} \\ : & : & : & : & : \\ D_{n1} & D_{n2} & D_{n3} & \dots & 0 \end{bmatrix}$$

Figure 2: Dissimilarity Matrix

Clustering Engine (CE): This part takes the dissimilarity matrix produced by DCE as input and applies the Hierarchical Agglomerative Clustering (HAC) algorithm [12] to find clusters of closely related classes. A number of methods are available for hierarchical clustering, such as single link, average link, complete link, and wards link [15]. We have used wards link method because of it giving better results than the others. The resulting class clusters are presented as a

dendrogram graph [16], as well as text. To implement this part, we used Java Numerical Library [17].

V. 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 match intended by the original developers, we used one of our previous moderate-size Java application, called Mudrik [18]. Mudrik has 76 classes organized into 9 packages. Table 1 shows the values of the weights we used for different component distances in Equation 5.

Table 1: Weights for component distances.

Distance Type	Variable	Weight
Package	PkW	3
Inheritance	IhW	5
Subling	SbW	3
Interface	ItW	3
Field	FdW	5
Parameter	PmW	2
Method	MtW	2

The tool has identified clusters as shown in the dendrogram of Figure 3. The clusters closely match the original grouping of classes we intended while developing Mudrik. The labels on the vertical axis are class names and the values on the horizontal axis are inter-clusters distances.

In another evaluation of the tool we tried to find the class clusters for open source 3D Java library called Jun [19]. Jun is a large library having 695 classes and 85 packages. We are familiar with Jun as we used it while developing Mudrik. Figure 4 shows the Jun class clusters discovered by our tool. To be able to read the classes names, the figure needs to be zoomed in. The clusters match closely to the groupings of classes we expected for Jun. The tool took around 7 minutes to process all Jun classes and produce the clusters. For Mudrik, it took less than a minute.

VI. CONCLUSIONS

We proposed an approach to extract useful knowledge from Java 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. A prototype tool has been developed as proof of concept. The tool has been evaluated using a moderate and a large sized Java application. The results are encouraging.

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] Java API documentation. http://docs.oracle.com/javase/7/docs/api/. Retrieved on February 3, 2014.

[15] El-Hamdouchi, A., Willett, P. Comparison of Hierarchic Agglomerative Clustering Methods. The Computer Journal, Vol. 32, No. 3, pp. 220-227, (1989)

[16] Dendrogram. http://en.wikipedia.org/wiki/Dendrogram. Retrieved on February 3, 2014.

[17] Java Numerical Library. http://www.roguewave.com/products/imsl-numerical-libraries/java-library.aspx. Retrieved on January 3, 2014.

[18] 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).

[19] Jun4Java. ftp://ftp.sra.co.jp/pub/lang/java/jun4java/. Retrieved on February 3, 2014

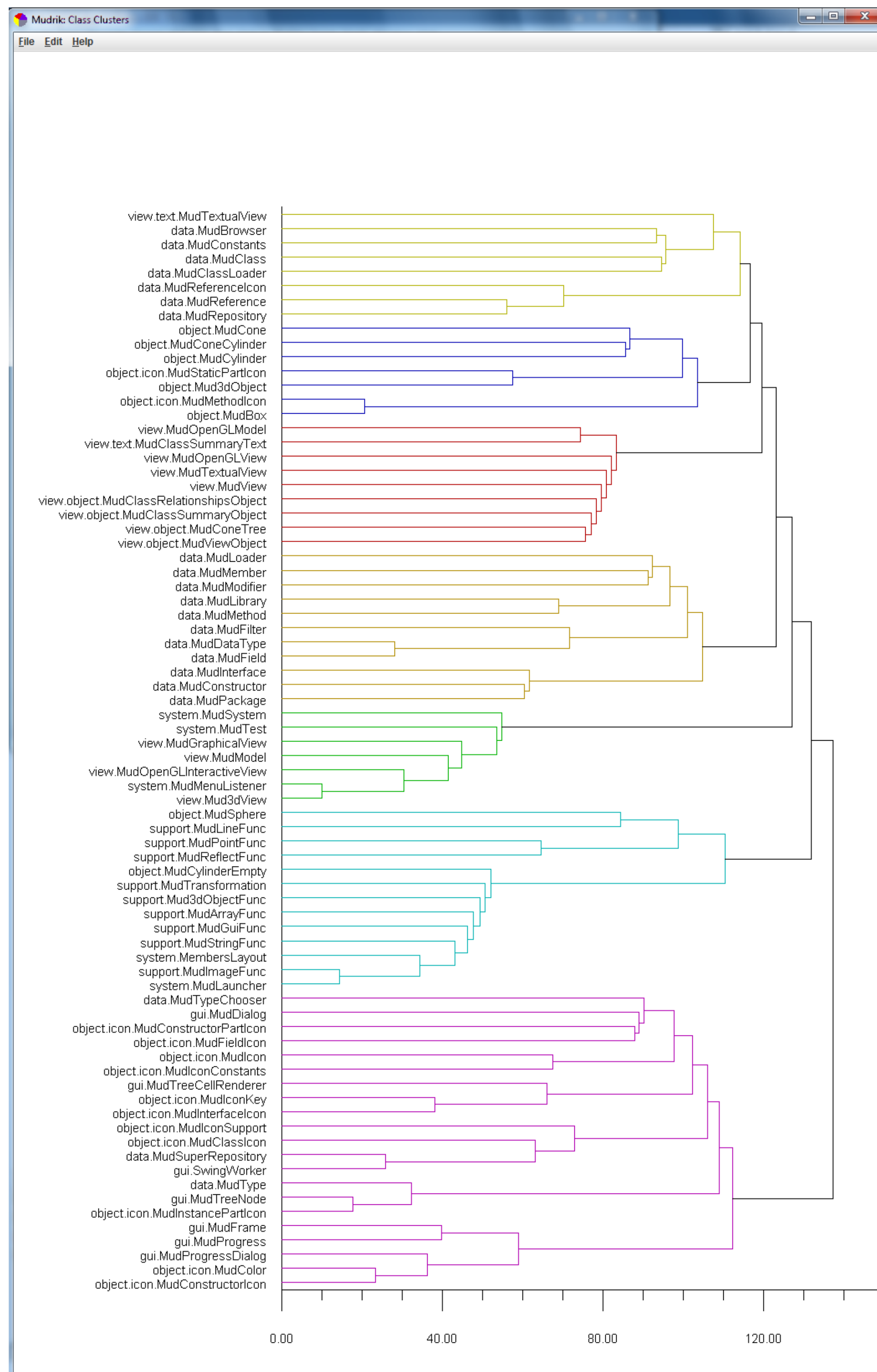

Figure 3: Mudrik class clusters

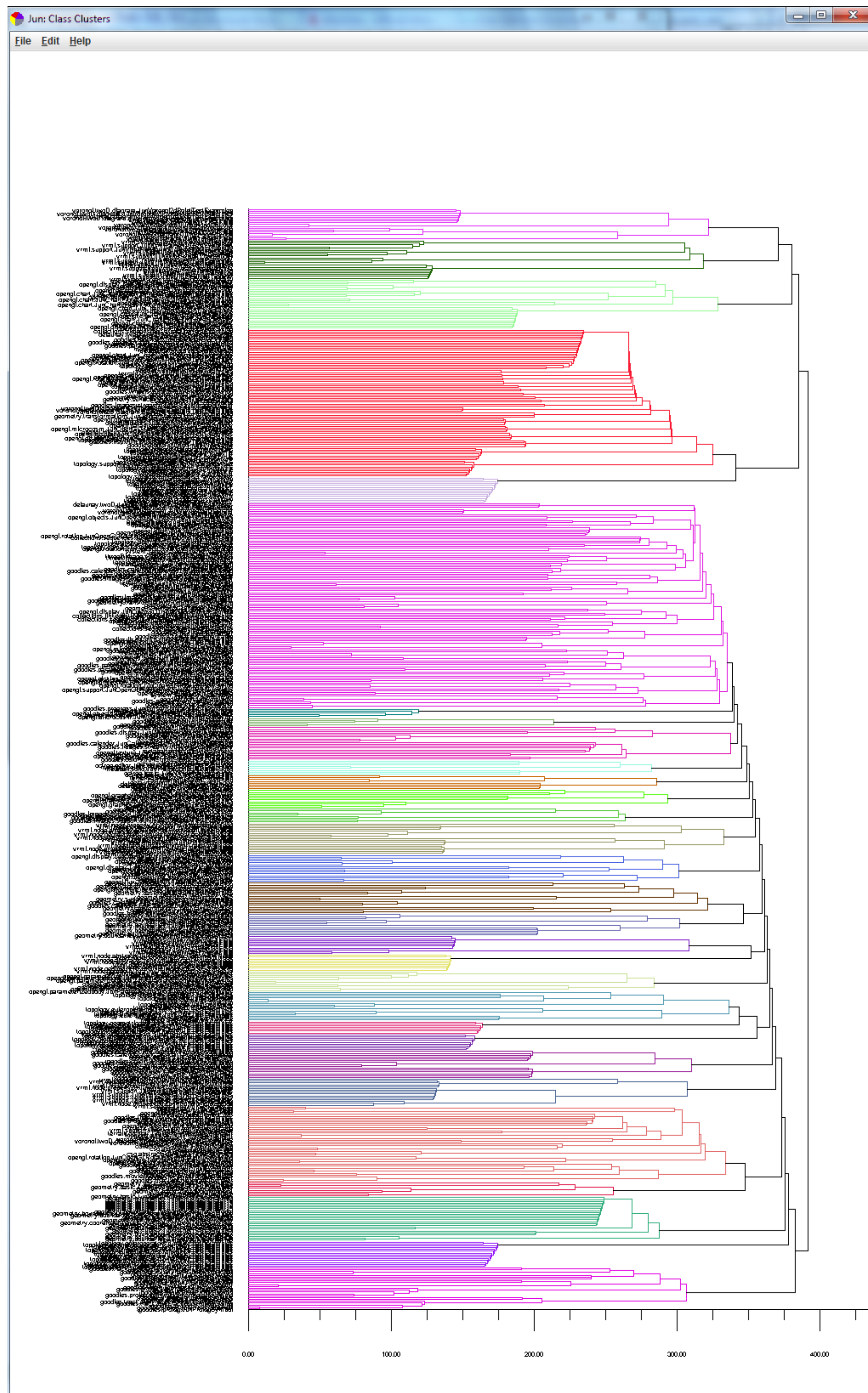

Figure 4: Jun class clusters