Understanding Java Source Code by Using Data Mining Clustering

Dimitris Rousidis and Christos Tjortjis

School of Informatics, University of Manchester, PO Box 88, Manchester, M60 1QD, UK
Email: D.Rousidis@postgrad.umist.ac.uk, tjortjis@manchester.ac.uk

Abstract. This paper presents ongoing work on using data mining clustering to facilitate software maintenance, program comprehension and software systems knowledge discovery. We propose a method for grouping Java code elements together, according to their similarity. The method aims at providing practical insights and guidance to maintainers through the specifics of a system, assuming they have little familiarity with it. Our method employs a pre-processing algorithm to identify the most significant syntactical and grammatical elements of Java programs and then applies hierarchical agglomerative clustering to produce correlations among extracted data. The proposed method successfully reveals similarities between classes and other code elements thus facilitating software maintenance and Java program comprehension as shown by the experimental results presented here. The paper concludes with directions for further work.

1. Introduction

A well documented problem faced by software maintainers when understanding a software system is the lack of familiarity with it, combined with the lack of accurate documentation [14]. Several techniques and methods have been proposed in order to facilitate this time consuming activity [3], [4], [7], [10], [12].

The work presented in this paper is part of a wider research effort investigating the applicability and suitability of using data mining to facilitate program comprehension and maintenance [9], [15], [17], [19]. This effort aims at developing a methodology for semi-automated program comprehension incorporating data mining. A fundamental underlying assumption is that the maintainer may have little or no knowledge of the program which is analysed.

Data mining comprises of a number of different techniques with a varying degree of suitability for software maintenance purposes [1], [11], [13], [16], [20]. Two are the most prominent issues: defining an appropriate data model for pre-processing as code is not a "tabular" format, and then selecting a suitable data mining technique. We have already proposed methods addressing these issues for both COBOL and C/C++ code [9], [19].

This work is based on Java, a well established programming language, which however has not been studied extensively in this context. We aim at providing software maintainers with a method that aids them to identify similarities and patterns among Java code elements, thus assisting program comprehension. Knowledge of relations among system parts is very important to effective maintenance.

The objectives of this work are:

i) to define a data model for Java, suitable to create target data for mining. Data should be extracted from Java source code by selecting the most essential elements for data mining purposes. We propose here a pre-processing algorithm that *'cleanses'* Java code by omitting any elements of non interest as ineligible data, extracts data from code using the input data model and populates a database.

ii) to convert data stored in the database into substantive clustering material by applying a transformation algorithm, and to cluster the converted data thus producing non-trivial and valid patterns.

The rest of this paper is organised as follows: Section 2 reviews research on data mining used for software maintenance and comprehension. In section 3 we propose a method for extracting useful knowledge form Java source code. Two case studies are discussed in section 4. Finally, section 5 concludes and proposes ideas for further work.

2. Background

Software maintenance is the most difficult stage in software lifecycle, often performed with limited understanding of the overall structure of a system because of commercial pressures [14]. Fast, unplanned modifications give rise to increased code complexity and deteriorated modularity, thus resulting in 50%-90% of the maintainers' time to be spent on *program comprehension* [18].

Data mining and its ability to deal with vast amounts of data, has been considered a suitable solution in assisting software maintenance often resulting in notable results [1], [11], [13], [16], [20]. We have also used data mining in the past, to get insights into systems design and structure [9], [17], [19].

The following paragraph briefly reviews some solutions in the area of data mining for software maintenance and compares these to our approach.

2.1 Using Clustering to Produce High-Level Organizations of Source Code.

This solution proposes a model for automatically partitioning system components into clusters in order to minimise inter-connectivity (i.e. connections between components of two distinct clusters) while maximising intra-connectivity (i.e. connections between components of the same cluster) by using a collection of existing clustering algorithms [11].

The proposed architecture is divided in four parts:

a. extract module-level dependencies from the source code and populate a database.

b. query the database, filter the query results and produce a textual representation of the Module Dependency Graph (MDG).

c. apply various clustering algorithms to the MDG.

d. visualise results.

This approach applies clustering to module dependencies and its main objective is to facilitate program comprehension with the underlying assumption is that a well-designed system is organised into cohesive clusters that are loosely interconnected.

2.2 A Software Evaluation Model Using Component Association Views

This solution proposes a model for architectural design evaluation of a system based on the association among system components [16]. It allows system modularity measurement, as an indication of design quality. The following three system association views are generated:

i) Control passing: It represents the correlation among system components based on function invocation.

ii) Data exchange: It epitomises the correlation among system components based on aggregate data types that are either passed as parameters between two functions or are referenced by a function.

iii) Data sharing: It signifies the correlation among system components based on functions sharing their global variables.

In this approach the software system is modelled as an attributed relational graph with system entities as nodes and data-control-dependencies as edges. At this point, application of mining association rules helps decomposing the graph into domains of entities based on the association property. The next step is to populate a database of these domains. This approach is based on the concept of the association between the components of a system. There are however other characteristics that play crucial role in grouping system components, such as the number of member data or functions in a class. These can be discovered by using other data mining techniques such as clustering.

3. Description of the Proposed Framework

The proposed framework was developed for pre-processing Java source code at the program level and consists of the following parts:

i) The input data model and the respective data type used to create and populate a database.

ii) The pre-processing model which transforms tables into a format suitable for clustering.

iii) The data clustering method which is applied to the pre-processed data.

The remaining of this section outlines the main characteristics of the input model, the pre-processing and cluster analysis methodologies.

3.1 Input Model

We selected four Java code elements to be used for clustering: packages, classes, methods, and parameters. We consciously have excluded less relevant code parts and unnecessary details. These elements are the entities stored in four respective tables. After we pre-process code to strip it from comments and white space, we analyse it in a top-down fashion (starting from *files/packages,* then *classes,* then *methods* and finally *parameters*). Each entity in this input data model has attributes which are stored in respective tables.

3.2 Pre-Processing Methodology

Pre-processing is a phase when Java code files are analysed and data are extracted and stored in corresponding tables, according to the input model discussed in 3.1. The process model for this is illustrated in Figure 2.

First we process packages and extract information about their name, and any packages imported from the Java API. Then we collect information on *Classes* such as their name, whether they inherit and which class from, and whether the class *implements* other classes and the names of the implemented classes.

Methods are parsed next. We extract their names, whether they have any arguments, their number and type, the returned values and the method's modifiers. Finally, parameters' attributes such as their name, and their type are gathered.

3.3 Clustering Methodology

We use an Agglomerative *Hierarchical Algorithm Clustering* (HAC) on the data extracted from Java code. This is because this technique gives more intuitive results and has also more widely used in the past in similar context [8], [17]. This algorithm creates sets of clusters in order of increasing distance. It requires that data should be pre-processed. As the tables contain entities with both nominal and numerical values we need to transform all values to numerical so that the distance among entities can be measured and stored in a matrix [6]. The details of this technique are analysed in Section 3.3.1. Finally, we use the *Single Linkage* Technique for merging two clusters.

3.3.1 Data transformation. The tables consist of raw, unprocessed data that are not suitable for clustering. These must be transformed in order to be of an acceptable

format for the algorithm. The records need to be transformed to a format appropriate for calculating their distance. Therefore we apply the following procedure:

We create a basic distance matrix where each attribute is assigned a numerical value. This value, (d(i, j)), is the distance between records I and j in the table. This distance can range between 0 and 1 (1 corresponding to the greatest distance) and is calculated by applying the following formula to each record:

$$d(i, j) = \frac{\sum_{f=1}^{n} X_{i,j} Y_{i,j}}{\sum_{f=1}^{n} X_{i,j}} \qquad \text{(Formula 3.1)}$$

where:

d(i, j): is the distance between the two records i and j,

n: is the number of attributes of each record.

X and Y are two functions dependent on *f*, and

$X_{i,j}$: is a function that can have only two values, 0 and 1.

- $X_{i,j}$ is 0 when an attribute, namely $q_{i,f}$ or $q_{j,f}$, of one of the two records is missing, otherwise,
- $X_{i,j}$ is 1.

$Y_{i,j}$: is a function that also depends on the type of attributes on each record:

i) if the attribute is binary or nominal, then if:

- $q_{i,f} = q_{j,f}$ then $Y_{i,j}$ equals 0, otherwise
- $Y_{i,j}$ equals 1.

ii) if the attribute is numerical then the function f is calculated based on the following formula:

$$Y_{i,j} = \frac{|q_{i,f} - q_{j,f}|}{(\max(q_{m,f}) - \min(q_{m,f}))} \qquad \text{(Formula 3.2)}$$

where:

$|q_{i,f} - q_{j,f}|$ is the absolute value of the subtraction between $q_{i,f}$ and $q_{j,f}$,

$\max(q_{m,f})$ is the maximum attribute value, and

$\min(q_{m,f})$ is the minimum attribute value.

Results after applying Formula 3.1 are shown in Figure 1. It is a distance matrix that stores a collection of proximities that are available for all pairs of n objects.

$$\begin{pmatrix} 0 & & & & & \\ d(2,1) & 0 & & & & \\ d(3,1) & d(3,2) & 0 & & & \\ . & . & . & 0 & & \\ . & . & . & & & \\ . & . & . & & & \\ d(n,1) & d(n,2) & \dots & \dots & 0 & \end{pmatrix}$$

Fig. 1: The Distance Matrix

The next example illustrates how the above formulas can be applied to the records of a database. Let us assume that the *Methods* Table in the database consists of 4 records with 8 attributes (MethodID, Method Name, Has Arguments, Number of Arguments, Return Type, Modifier, Other, ClassID) as in Table 1.

Table 1: *Methods* Table

MethodID	Method Name	Has Arguments	Number of Arguments
M0003	tick	no	0
M0011	swap	yes	2
M0017	singleton	no	0
M0033	createIcon	yes	1
Return Type	**Modifier**	**Other**	**ClassID**
void	public	static	C0002
void	public		C0002
void	private	static	C0004
Icon	protected	static	C0012

By applying the Formula 3.1 we derive a distance matrix that corresponds to a format similar to that of the matrix in Figure 1. The *methods* distance matrix is shown in Figure 2.

	1	2	3	4
1	0			
2	0.571	0		
3	0.429	0.857	0	
4	0.786	0.786	0.786	0

Fig. 2: Distance Matrix for the *Methods* Table

For instance, the distance between the second record (method *Swap*) and the third record (method *Singleton*) is 0.857, with 0 being the most similar and 1 the least similar. We calculate this distance using Formula 3.1. For example, when comparing attribute values for *ReturnType*, the fifth column in Table 1, we see that both methods

have *void* return type. Therefore, the function $Y_{i,j}$ receives the value 0, since the two objects are equal and the attribute's element is a nominal relation.

The same procedure is followed for remaining attributes, except the (*NumOfArguments*), where Formula 3.2 is used, as the attribute is numerical. In this occasion we take into consideration the minimum and maximum value for attribute *NumOfArguments*, zero and two respectively.

We apply the formulas to these two records (the second and the third row), and calculate their distance to be:

$$d(2,3)=\frac{\sum_{f=1}^{n} X_{i,j} Y_{i,j}}{\sum_{f=1}^{n} X_{i,j}}=\frac{\sum_{f=1}^{7} X_{2,3} Y_{2,3}}{\sum_{f=1}^{7} X_{2,3}}=\frac{1*1+1*1+1*1+1*0+1*1+1*1+1*1}{1+1+1+1+1+1+1}=\frac{6}{7}=0.857$$

where:

the first product represents the distance of the two clusters for the first column that is checked (the *MethodName* column, since the *MethodID* column is been excluded), the second product represents the distance for the second column and so on.

3.3.2 Merge into a new cluster with the smallest distance. Whenever the smallest distance between two clusters is found then these two clusters are merged into a new one, and the number of the rows is decremented by one.

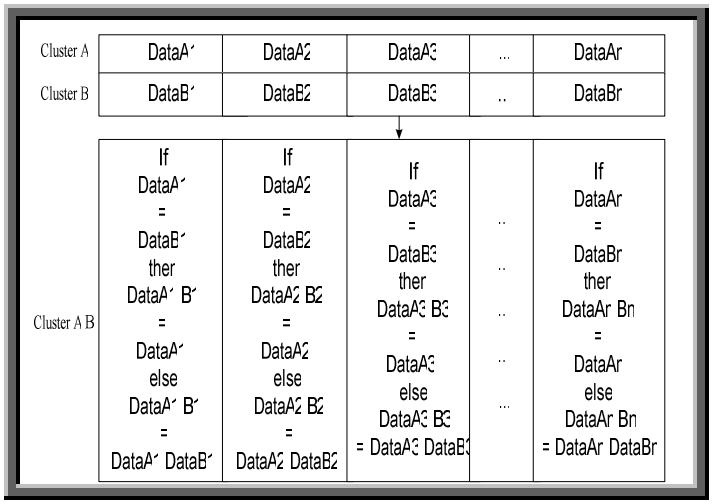

Fig. 3: Merging the two most similar clusters

The methodology that is followed for this part of the clustering procedure was based on the comparison of the clusters:

i) If the two merging clusters have the same data, numerical or nominal, then the new cluster has exactly the same information. For instance if the data at both the initial clusters is *"sameData"*, then the new merged column also receives the *"sameData"* information.

ii) If the clusters have different data then these two different pieces of data are joined into a new one, which will have both pieces of data separated by a comma. For instance if the value (nominal or numerical) on the first cluster is *"value1"*and on the second cluster is *"value2"* then the merged cluster receives in the same column the value *"value1,value2"*.

This pattern is shown in Fig. 3.

After applying an *Agglomerative Algorithm* results and a textual representation of correlations and patterns among Java code elements are produced.

4. Result Evaluation

Our method aims at grouping Java program components according to their similarity. These grouping can be evaluated for precision by comparing them with the original developers' perceptions on such "natural groupings". Thus we cross-reference actual results as produced by our method, with expected ones as proposed by the developers of Java systems. We discuss here a case study on a program called "Keep in Touch (KIT)", where the programmer was requested to group classes according to their perceived conceptual similarity among these classes. Keep In Touch (KIT) is a medium application. A very small fragment of this application with 15 methods was used for experimentation.

4.1 Case Study

Methods were grouped by the programmer according to their perceived similarity into 3 groups (DB Control, Setters-Getters and GUI) as presented in Table 2. Results after applying our method are shown in Table 3.

Table 2: Case study: Expected results

Group 1 (DB Control)	Group 2 (Setters-Getters)	Group3 (GUI)
prospectActivity	activitySupport	activityForm
getPractivityRow	clearAllFields	showDialog
getPPractivityPK	updateAllFields	setState
getActivity	Create	process
getDescription	Commit	fire

Table 3: Case study: Actual results.

Group 1 (DB Control)	Group 2 (Setters-Getters)	Group3 (GUI)
getPractivityRow getPPractivityPK getActivity getDescription	clearAllFields updateAllFields create commit	activityForm showDialog setState process fire *activitySupport*
Group 4 (Misc)		
prospectActivity		

Comparison of the two Tables 2 and 3 shows that 13 out of 15 methods (86.67% precision) were placed in the correct group and just one method (activitySupport) was actually misplaced. The prospectActivity method was not grouped with any methods because of the high number of its arguments. Such result was expected as the number of the method's arguments is of great importance for the distance measurement (as it is appears in Formula 3.2). The main drawback of this formula is that it takes twice into account the property of a method having or not having an argument, once with the *HasArguments* attribute, and second time with the *NumOfArguments* attribute. We run further tests on the inclusion or not of these attributes and turned out that, despite the increase in distance between two records when there is a big difference in their arguments number, using these increases overall precision.

All in all, the main reason for achieving precision slightly less than 100% can be attributed to the exclusion of significant attributes. Results could be improved if a top-level attribute would be added at the tables. The *packageID* should be included at the *Class* Table, the *ClassID* at the *Method* Table and the *MethodID* at the *Parameter* Table. Finally, including more attributes which address Java code details is an option worth considering. For instance argument type along with its name can be used as extra attributes.

5. Conclusions and Future Work

This section presents conclusions drawn by evaluating the proposed method and comparing it to similar ones. Directions and ideas for future work are also discussed.

5.1 Conclusions

The proposed method successfully reveals similarities between classes and other code elements. In addition, whenever a frequently used method or class is identified, these can be added at the Java vocabulary and then used without having to reconstruct

them. This pattern recognition can reduce the programmer's effort when understanding and maintaining Java systems. In addition, possible repetitious or redundant methods and classes can be identified, and thus some reduction of the code's volume can be achieved.

Another benefit of the method is the freedom it provides for controlling records in the database. Tables are accessible for filtering and selection of specific information through queries.

The method successfully takes into account and recognises data about *packages, classes, methods* and *parameters*. On the other hand, not all of the grammar was considered. There are many keywords, along with their accompanied data that were excluded from our analysis; this is work we plan to carry out in the future. The following section gives some more directions for further work.

5.2 Future Work

We have so far explored only a limited subset of possibilities with regards to the input model and the data mining technique to be used on Java code. Obviously the input model depends on the technique. The model could be extended to include more elements like *Objects* and *Array*s, control statements (*if…else, while, do…until, switch,* and so on), *Exceptions* and even other keywords.

We used a hierarchical agglomerative clustering with single linkage. Alternatives include a number of different clustering algorithms which can be parameterised for maximum performance. Additional data mining techniques like classification, association rules could also be used for experimentation. Moreover, metrics can be used and the methods proposed by Basili et al [2] or Chidamber and Kemerer [5] can be adopted, although data mining seems to be more suitable for our method.

Another possible improvement could be to include weighs for attributes [5] to reflect their relative importance. The algorithm also can be optimised. For instance the merging algorithm introduced in section 3.3.1 could improve by use of a more sophisticated version which involves having more than one "smallest distance" pair of clusters in the distance matrix. Instead of merging just one record at a time we could be merging more than one pairs with the same smallest distance.

Finally, further evaluation on larger and more complex programs is needed to assess how the method scales to deal with real industrial scale systems.

References

1. N. Anquetil and T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization method", *Proc. 6th Working Conf. Reverse Engineering* (WCRE 99), IEEE Comp. Soc. Press, Oct. 1999, pp. 235-255.
2. V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", IEEE Trans. Software Engineering, 22(10), 1996, pp. 751-761

3. E. Burd, M. Munro, "Evaluating the Use of Dominance Trees for C and COBOL", Proc. Int'l Conf. Software Maintenance (ICSM 99), IEEE Comp. Soc. Press, 1999, pp. 401-410.
4. G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "Decomposing legacy systems into objects: an eclectic approach", Information and Software Technology, Vol. 43, 2001, pp 401-412.
5. S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", IEEE Trans. Software Engineering, 20(6), 1994, pp. 476-493.
6. M.H. Dunham, *Data Mining, Introductory and advanced topics*, Prentice Hall, 2002.
7. A.R. Fasolino and G. Visaggio, "Improving Software Comprehension through an Automated Dependency Tracer", *Proc. 7th Int'l Workshop Program Understanding* (IWPC 99), IEEE Comp. Soc. Press, 1999.
8. U. Fayyad, G. Piatetsky-Shapiro, and R. Uthurusamy, "From Data Mining to Knowledge Discovery: An Overview", in *Advances In Knowledge Discovery and Data Mining*, AAAI Press/The MIT Press, 1996.
9. Y. Kanellopoulos and C. Tjortjis, "Data Mining Source Code to Facilitate Program Comprehension: Experiments on Clustering Data Retrieved from C++ Programs", *Proc. IEEE 12th Int'l Workshop Program Comprehension* (IWPC 2004), IEEE Comp. Soc. Press, 2004, pp. 214-223.
10. P. Linos, Z. Chen, S. Berrier, and B. O'Rourke, "A Tool for Understanding Multi-Language Program Dependencies", *Proc. IEEE 11th Int'l Workshop Program Comprehension* (IWPC 03), IEEE Comp. Soc. Press, 2003, pp. 64-72.
11. S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen and E.R. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code", *Proc. 6th Int'l Workshop Program Understanding* (IWPC 98), IEEE Comp. Soc. Press, 1998, pp. 45-53.
12. A. von Mayrhauser and A.M. Vans, 'Program Understanding Behavior During Adaptation of Large Scale Software', *Proc. 6th Int'l Workshop Program Comprehension* (IWPC 98), IEEE Comp. Soc. Press, 1998, pp.164-172.
13. C.M. de Oca and D.L Carver, "Identification of Data Cohesive Subsystems Using Data Mining Techniques", *Proc. Int'l Conf. Software Maintenance* (ICSM 98), IEEE Comp. Soc. Press, 1998, pp.16-23.
14. T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, Wiley Computer Publishing, 1996.
15. D. Rousidis and C. Tjortjis, "Clustering data retrieved from Java source code to support software maintenance: A case study", *Proc. IEEE 9th European Conf. Software Maintenance and Reengineering* (CSMR 2005), IEEE Comp. Soc. Press, 2005, pp. 276-279.
16. K. Sartipi, K. Kontogiannis and F. Mavaddat, "Architectural Design Recovery Using Data Mining Techniques", *Proc. 2nd European Working Conf. Software Maintenance Reengineering* (CSMR 00), IEEE Comp. Soc. Press, 2000, pp. 129-140.
17. C. Tjortjis, N. Gold, P.J. Layzell and K. Bennett, "From System Comprehension to Program Comprehensions", *Proc. IEEE 26th Int'l Computer Software Applications Conf.* (COMPSAC 02), IEEE Comp. Soc. Press, 2002, pp. 427-432.
18. C. Tjortjis and P.J. Layzell, "Expert Maintainers' Strategies and Needs when Understanding Software: A Qualitative Empirical Study", *Proc. IEEE 8th Asia-Pacific Software Engineering Conf.* (APSEC 01), IEEE Comp. Soc. Press, 2001, pp. 281-287.
19. C. Tjortjis, L. Sinos and P.J. Layzell, "Facilitating Program Comprehension by Mining Association Rules from Source Code", *Proc. IEEE 11th Int'l Workshop Program Comprehension* (IWPC 03), IEEE Comp. Soc. Press, 2003, pp. 125-132.
20. V. Tzerpos and R.C. Holt, "ACDC: An Algorithm for Comprehension-Driven Clustering", *Proc. Working Conf. on Reverse Engineering 200*0, (WCRE00) IEEE Comp. Soc. Press, 2000, pp. 258-267.