Clustering data retrieved from Java source code to support software maintenance: A case study

Dimitris Rousidis and Christos Tjortjis
School of Informatics, University of Manchester,
PO Box 88, Manchester, M60 1QD, UK
D.Rousidis@postgrad.umist.ac.uk, tjortjis@manchester.ac.uk

Abstract

Data mining is a technology recently used in support of software maintenance in various contexts. Our works focuses on achieving a high level understanding of Java systems without prior familiarity with these. Our thesis is that system structure and interrelationships, as well as similarities among program components can be derived by applying cluster analysis on data extracted from source code.
This paper proposes a methodology suitable for Java code analysis. It comprises of a Java code analyser which examines programs and constructs tables representing code syntax, and a clustering engine which operates on such tables and identifies relationships among code elements.
We evaluate the methodology on a medium sized system, present initial results and discuss directions for further work.

1. Introduction

Program comprehension is an important part of software maintenance, especially when program structure is complex and documentation is unavailable or outdated [7]. Data mining can produce high level overviews of source code and interrelationships among program components thus facilitating legacy systems understanding [1], [5], [6], [8], [10].

The aim of this work is to establish whether data mining techniques can support maintenance of Java software systems and the extent to which understanding such systems can be semi-automated by use of relevant tools. The main objective is to define a methodology which supports pattern identification and program element similarity assessment.

Code mining in that sense requires the definition of a data model which can be used to transform Java source code into database tables suitable for mining. Clustering is a natural choice among data mining techniques when it comes to identifying and measuring similarities among program entities. It is also a technique which does not require any prior domain knowledge, making it more suitable for maintainers with limited or no knowledge of the program which is analysed.

This paper presents a methodology for clustering information extracted from Java source code aiming at capturing program structure and achieving better program understanding. A tool was implemented to assess this methodology. It uses a data model defining entities and attributes which can be extracted from code and then groups these entities based on similarity measurements. These groups indicate interrelated entities. This can be verified by the findings of a case study presented here.

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 methodology for extracting useful knowledge form Java source code and present a "proof of concept" tool we used for experimentation. The case study is discussed in section 4. Section 5 concludes and proposes ideas for further work.

2. Background

Data mining can discover non-trivial and previously unknown relationships among records or attributes in large databases [3]. This highlights the capacity of data mining to obtain useful knowledge about the structure of large systems. It has three features that make it useful for program comprehension and related maintenance tasks [6]:

a) It can be applied to large volumes of data. This implies that it has the potential to analyse large systems with complex structure.
b) It can be used to expose previously unknown non-trivial patterns and associations between items in databases. Therefore, it can be used to reveal hidden relationships among program components.

c) It can extract information regardless of any previous domain knowledge. This feature is ideal for maintaining software with poor knowledge about its functionality or implementation details.

Data mining has been previously used for clustering over a Module Dependency Graph (MDG) [5] and for identification of subsystems based on associations (ISA methodology) [6]. Both approaches provide a system abstraction up to the program level. The former creates a hierarchical view of system architecture into subsystems, based on the components and the relationships between components that can be detected in source code. This information is captured through an MDG, which is then analysed in order to extract the required structural view. The latter approach produces a decomposition of a system into data cohesive subsystems by detecting associations between programs sharing the same files.

Sartipi et al. used data mining for architectural design recovery [8]. Their method is based on associations among system components and uses system modularity measurement as an indication of design quality. This approach models software systems as attributed relational graphs with system entities as nodes and data-control-dependencies as edges. Application of association rules mining decomposes such graphs into domains of entities based on the association property.

Work on clustering as a means to supporting software maintenance and understanding has been also conducted in a number of contexts addressing varying levels of abstraction and a variety of programming languages ranging from COBOL, to C or even C++ [1], [4], [9], [10].

3. Proposed Methodology

Our approach aims at defining entities and respective attributes in Java code which can then be fed into a clustering engine in order to produce groupings of such entities according to their similarity. The methodology consists of two main parts the input model and the clustering algorithm.

The input model takes into account five basic Java code elements: files, packages, classes, methods, and parameters. These elements form the entities to be stored in respective tables. Each entity has a number of associated attributes. A brief description of the attributes in the input model is given in Table 3.1.

Table 3.1: Input model description

Table name	Attributes	Description
Files	*fileID*	Unique file ID
	fileName	File name and directory path
Packages	*PackageID*	Package unique ID
	PackageName	Package name
	ImportedPackage	Packages imported from API
	FileID	File ID the package belongs to
Classes	*ClassID*	Unique class ID
	ClassName	Class name
	Inherits	Yes/no
	InheritsFrom	Superclass name
	Implements	Interfaces names
	ImplemetsTo	Name of class that follows the *implements* clause
	FileID	File ID the class belongs to
Methods	*MethodsID*	Unique method ID
	MethodName	Method name
	HasArguments	Yes/no
	ArgumentsNum	Number of arguments passed by the method
	ReturnedValue	Value type returned by the method
	Modifier	Modifiers a method can be declared with
	Other	Any additional method features
	FileID	File ID the method belongs to
Parameters	*ParameterID*	Unique parameter ID
	ParameterName	Parameter name
	ParameterType	Parameter type
	ParameterUse	Parameter used by reference / value
	FileID	File ID the parameter belongs to

3.1 Clustering Algorithm

We employ the *Hierarchical Agglomerative Clustering* (HAC) algorithm as this technique gives more intuitive results and has been used extensively in similar contexts [3], [9]. This algorithm requires pre-processed data and produces sets of clusters in order of decreasing similarity. As the tables contain entities with both nominal and numerical values all values need to be transformed to numerical so that the distance among entities can be measured and stored in a *similarity matrix* [2]. 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.

This distance (d(i, j))can range between 0 being the nearest and 1 being the value that corresponds to the farthest distance. So, $0 \leq d(i, j) \leq 1$. The distance is calculated by applying the following formula on each

record of the tables that are of great significance in the database.

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

Formula 3.1: Distance Calculation

where:

d(i, j): is the distance between the two records i and j,
n: is the number of attributes of each record.

The two functions that are dependent on *f*, are X and Y, and
$X_{i,j}$: is a function that can obtain just 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 is also dependent on the type of the attribute of each record:

i) if the attribute is of binary type (for instance Boolean), or whichever nominal type, 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:

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

Formula 3.2: The $Y_{i,j}$ function calculation

where:

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

$\max(q_{m,f})$ is the maximum numerical value of the attribute of the column of the record, an

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

By applying the formula 3.1 it is feasible to translate the records of the tables within the database into material that is appropriate for the data mining clustering.
Clusters are merged recursively using the *single linkage* technique, i.e. two clusters are merged if the distance between an element in one cluster and an element in the other is minimum. When merging two clusters or entities the union of their attributes is the attribute list of the new, merged cluster.

3.2 "Proof of Concept" Tool

We implemented a prototype tool in order to evaluate our methodology. It consists of two main parts:

- The pre-processing engine which parses the source code and stores data extracted according to the input model in a database.
- The clustering engine which allows for record and attribute selection and produces clusters of program elements using the *HAC* algorithm.

Results are presented via a Graphical User interface. The whole process is shown in Fig. 3.1.

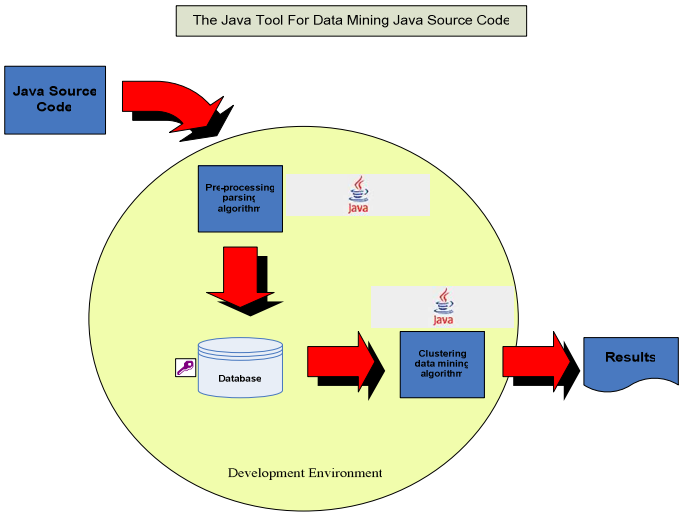

Fig. 3.1: Proof of concept tool and process

4. Case study

Our methodology groups Java program components according to their similarity. Such groupings can be evaluated by comparison to the original developers' perceptions on components' "natural groupings". As a case study we used a small fragment of a medium application called "KIT (Keep In Touch)". The part of the application provided has 15 methods. Its programmer grouped these methods according to their conceptual similarity into 3 groups (DB Control, Setters-Getters and GUI) as depicted in Table 4.1.

Table 4.1: Programmer's groups

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

We then input the applications code into the "proof of concept" tool and retrieved four groups as depicted in Table 4.2.

Table 4.2: Groups proposed by methodology.

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

Comparison of the two tables shows that 13 out of 15 methods (86.67% precision) were placed in the correct group and only one method (activitySupport) was misplaced. One method (prospectActivity) was not grouped with any others because of the high number of its arguments.

5. Conclusions and Future Work

In this paper we proposed a methodology for clustering data extracted from Java source code in order to better understand similarities among program elements in support of software maintenance. This methodology consists of an input model and a clustering algorithm. It correctly recognises data about packages, classes, methods and parameters.

A tool was developed to assess this fully automated approach. Initial experimental results from a case study were encouraging. The tool successfully reveals similarities among Java code elements.

Precision could further improve by adding more attributes to the entities. For example a *packageID*, a *ClassID*, and a *MethodID* could be included at the *Class*, *Method* and *Parameter* entities. Also argument type and name can be used as extra attributes.

Other possibilities worth of exploration involve extending the data model with more grammatical elements like *objects* and *array*s, control statements (*if…else*, *while*, *do…until*, *switch*), and *exceptions*.

The clustering engine can also be fine tuned or parameterised by use of alternatives algorithms and linkage methods. Assigning weighs to important attributes could also improve performance but this may require specialised domain knowledge. Finally, further evaluation on larger and more complex programs is needed to assess how the methodology 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] M.H. Dunham, *Data Mining, Introductory and advanced topics*, Prentice Hall, 2002.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing your Software Investment*, Wiley Computer Publishing, 1996.

[8] 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.

[9] 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.

[10] V. Tzerpos and R.C. Holt,"ACDC : An Algorithm for Comprehension-Driven Clustering", Proc. Working Conf. on Reverse Engineering 2000, (WCRE00) IEEE Comp. Soc. Press, 2000, pp. 258-267.