ISSN 22773061

Application of Artificial Intelligence methods in Finding Program Comprehension Differences in Novice Object Oriented Programmers

Marzieh Ahmadzadeh, Elham Mahmoudabadi
School of Computer Engineering & IT, Shiraz University of Technology, Shiraz, Iran
ahmadzadeh@sutech.ac.ir
School of Computer Engineering & IT, Amirkabir University of Technology, Tehran, Iran
mahmoudabadi@aut.ac.ir

ABSTRACT

Program comprehension is the first step required for software maintenance, which accounts for a considerable number of job opportunities. For this to happen, it seems obvious that improving this ability in the teaching environment is required. The literature shows, however, that not enough solutions for improving program comprehension are offered as much as for programming itself.The aim of this research therefore, is to find a pattern of how different students vary in terms of comprehending a code written in an object-oriented language. For this, we have focused on two concepts including inheritance and polymorphism, gathered data online and analyzed it qualitatively. To find the right subject for all the students to study, a data mining technique i.e., the K-means clustering algorithm, was used. Results showed that a slight difference in programming experience can have a significant impact on program comprehension ability. The methods that were used by participants who succeeded in the experiment were the same as methods used by experts as mentioned in earlier research. *Inheritance* and *polymorphism* did not play an important role in lack of success in the process of program comprehension.

Indexing terms/Keywords

Program Comprehension, Experimental Approach, Data Mining, Object-Oriented Programming.

Academic Discipline And Sub-Disciplines

Computer Science Education; Data Mining.

SUBJECT CLASSIFICATION

Programming, Object-Oriented, Clustering.

TYPE (METHOD/APPROACH)

Experimental Method.

ISSN 22773061

INTRODUCTION

The majority of jobs in the area of software development are allocated to software maintenance [13]. This has major implications for universities to help enhance the abilities of students' program comprehension as potential future employees as the first step toward debugging and maintaining a program [15] as much as they do for programming. To find a way to help students in this area, one needs to find out what is involved in the process of program comprehension and how students differ in this ability. The former has received significant attention[15], while the latter has gaps in research mostly because a new paradigm of programming is being introduced, which requires more research.

Anecdotal evidence shows that students sometimes learn from their peers better than from their lecturers even if their peers' explanations are not precise or comprehensive. The reason for that, perhaps, is that the explicator has just recently passed through the same experience, from *not-knowing* to *grasping* the concept. Perhaps the students do not share the same learning style and a specific lecture works for one but does not work for another and the explicator unconsciously explains the subject matter in a way that matches his/her peer's learning style. Whatever the reason, we can take advantage of this for our teaching purposes. If different methods that different students in the same level take to approach program comprehension (i.e. successful students in program comprehension vs. unsuccessful students) are explored, they can be included in the teaching process, one way or another, or even in developing an adaptive tool to improve students' program comprehension.

In this research we aimed to find these differences through an experimental approach when students need to understand a program that has been written in an object-oriented manner in which concepts such as polymorphism and inheritance have been included. In other words, we needed to explore the differences between the actions that each individual student or groups of students take in this process. Choosing the right participants for the study, however, was not an easy task. For example, we could not take the high performing students of a class as one group and low performing students as another group and compare them since high performance in programming does not necessarily lead to performing better in program comprehension. We have previously observed that two students with the same ability in programming do not necessarily have the same ability in debugging [1]. This means that the skills required for programming are not the same as the skills needed for program comprehension. One solution to this problem was to select the right participants through an experiment. Since this kind of research is qualitative in nature and requires a lot of time and effort for data analysis, we decided to use a clustering algorithm (i.e. K-means – elaborated later) to differentiate between groups of students. After the right subjects were chosen, the main experiment was run and data was analyzed qualitatively.

This paper is organized as follows. In the next section the review of literature and their connection to current research are explained. The designs of both experiments are elaborated in section three and the results are discussed in the fourth section. Finally, a conclusion, summary and suggestions are presented in the final section.

Review of Literature

Program comprehension research has received significant attention ranging from exploring cognitive theories to developing comprehension tools via empirical to experimental approaches during the past few decades [4-6, 8-10, 12, 14].

Shneiderman and Mayers [12] believed that comprehension takes place in a bottom-up manner in which a programmer reads code statements and then groups those codes to form a high level understanding of the code. Brooks [4] however, theorized that a programmer makes a general hypothesis about the given program in the first place and then purifies and verifies it in a hierarchical manner. The verification of hypothesis depends on how familiar the code feature is. Later, Soloway and Ehrlich [14] stated that a top-down understanding takes place when the code is familiar. Letovsky [7] mentions that knowledge of the specific application is an important issue for programmers to form their initial mental models, which will evolve during program examination. This was later confirmed by Shaft and Vessay[10] . Littman, et.al., [8] observed that in order to comprehend a program, programmers either read the code from top to bottom or focused on the part that is relevant to the problem.

From a different perspective, Pennington's [9] experiment revealed that the process of program comprehension is affected by the language that program is written in. For example, programs written in FORTRAN were better understood than the ones written in COBOL in terms of control-flow understanding. It should be noted however, that the participants in her experiments were expert programmers who are different from novices.

While all these works have been carried out in the procedural programming area, the research by Karahasanovic, et. al., [5] examines program comprehension in object-oriented programming. In this research they ran a controlled experiment to understand the strategies that participants use to maintain a program in an object oriented language (i.e. Java). The results showed that their beginner participants applied both the *semantic* and the *as-needed* strategy in the program comprehension process. In their paper, semantic strategy has been defined as a comprehension process by which a programmer gets as much information as is possible from documentation, source codes, etc. about the program while the as-needed strategy pointed to situations where the programmer did not go into the details of the program from the beginning of the process. In terms of object-oriented concepts they observed that participants had problems specifically in the inherited functionalities of the program.

In an effort to compare novice and expert object-oriented programmers' program comprehension, LaToza, et.al., [6] found that programmers used *facts* that were not in any rate complicated to comprehend a program. Differences found between novices and experts in their experiment included the fact that experts looked for the root of the problem, visited only the

necessary methods (not all of them) and applied the modifications faster while novices paid attention to symptoms only, spent time reading the unnecessary code and were not able to explain the facts.

In this current research the focus is on object-oriented program comprehension, specifically considering inheritance and polymorphism concepts. We chose an experimental approach for data gathering with participants being novice students.

Design of Experiments

As explained in the introduction of the paper, we were aware that students with high ability in programming, who are better in comprehending a program, could not be chosen as participants. Therefore, an experiment was required in order to select the right participants for this study. This was done through a first round of experiments. In the second round of experiments we were able to work with a lower number of the students who were eligible for the study.

Both experiments were carried out at Shiraz University of Technology. The participants were juniors who had already finished their first programming course and were in the middle of their second programming course. The language taught for this course was Java and we used object-last strategy in teaching the programming.

The details of both experiments were explained to the participants prior to running the experiments to give them a choice to accept or reject participation in the experiments. In the first round we encouraged all of the juniors to participate in the experiment. Participation was completely voluntary, however, some incentives were considered. Thirty six students signed up for the experiment. At first the participants were given a questionnaire in which they specified whether or not they had programmed prior to starting the course and their level of interest in programming. The level of interest ranged between one (none) and five (high level of interest). They were then given a program specification and an incomplete working version of the code and asked to complete the program. No time limit was set for this experiment and students were allowed to spend as much time as they needed. The experiment was finished when a student finished the coding or resigned from proceeding. We used Camtasia software to record every single action that they took to solve the problem instead of directly observing their actions. We did this to avoid the Hawthorne effect, which means that we did not want students to change their behavior just because somebody was watching them. The program specifications and the given code are found in Appendix A.

Among the 36 students, seven students turned off the recording software. Therefore, we were not able to track their work. Sixteen students did not make any significant effort to make the program work. To keep the validity of the analysis, these were excluded from data analysis. Thus, the remaining thirteen students were considered for data analysis. Six out of thirteen delivered perfectly working code and the other seven completed half. These thirteen students were our main participants in the second round of the experiment. As mentioned before, we were looking for participants who showed some ability in program comprehension. Therefore, according to the codes that were delivered, we were able to select thirteen students.

In the second round of the study, similar to the first round, a program specification and an incomplete version of a code were given to our thirteen participants who were then asked to complete and deliver a working code. The program specification and the incomplete version of the code can be seen in Appendix B. Once again, no time restriction was set. Six out of thirteen students delivered a non-working code in which the code transformation was not consistent with what was asked. These were excluded from the data analysis. Therefore, the data received from seven students during two experiments was the final data for analysis. The performance of our seven participants is elaborated in the analysis section.

At the end of the experiments three types of data were inserted into a database. A first table contained program information consisting of experiment number, student identification, start time, end time and the level of completeness of the delivered code. The last field ranged from zero to two. Zero meant that the delivered code was complete and worked perfectly while students who were assigned one, had made some effort but the delivered code did not work perfectly. Codes that did not work at all, whether no significant code was added to the original code or an erroneous code was delivered, were assigned number two. The second table contained students' information consisting of student identification, midterm mark, final mark, level of interest, and familiarity with programming prior to starting the course. The third table contained three fields including student identification, experiment number and activity code. Activity code refers to the action taken by a student. This meant that for one student in an experiment tens or even hundreds of rows were filled. Rows were added according to time of occurrence, which meant the action inserted in row *n+1* happened after the action inserted in row *n*

The idea of introducing activity codes comes from Sharp, et. al. [11], where human behavior is studied qualitatively. In fact, we decided to study human behavior in this research because we were interested in seeing how different students approached solving such problems. Therefore, defining and using activity codes was an obvious application of this research. To define activity codes we first used domain knowledge for what was expected to be accomplished. For example, in order to define a *setter* method one needs to type *public*, the name of the method, and then define the input argument, etc. Each of these actions was assigned an individual code. We then looked at a few programs randomly to check for any missed activity. If any had been left out, it would have been assigned a code. Of course, our list was not a complete list of actions that one could take to program instead, it was a limited actions list that was needed to complete our specific assignment. The designed list of activities, with corresponding codes, can be seen in Appendix C. Consecutive numbers were chosen for each activity except for the *wait* activity that was assigned code 1000 for two reasons. First, we needed a number that was straightforward and required less computation. For example, if somebody waited for 2 and another for 3 minutes, their corresponding activity was 1002 and 1003 respectively. Second, we were not

able to predict how many activities we would have, thus, we needed to choose a number which was big enough to create enough space for future activities to be inserted.

We borrowed our exploratory approach from other researchers [6, 17] who chose two groups of programmers, experts vs. novices, to investigate their debugging processes. In this research we categorized two groups of novice students into ones who were successful in debugging the given program and ones who were not. We were then able to find the intra and inter similarities and differences between the two groups.

Analysis

At the end of data gathering we had 4278 records of data in our third table ready to be analyzed. This data belonged to the seven students from both experiments, including 2232 records from the first experiment and 2046 records from the second experiment. To get an understanding of what is involved in comprehending a program, we needed to look at the data that was received from our experiments. This was not an easy task since the amount of data was too big to be processed manually. However, a relatively simple way to analyze the data was to cluster students into similar groups and then investigate the behavior of the groups. To group them we used a clustering technique that is elaborated upon in the next subsection.

Clustering Algorithms

Data mining is an effort to extract knowledge from massive numbers of available data. One of the techniques introduced in data mining is clustering. The job of clustering is to separate the available data into groups that have the most similarity. A number of clustering algorithms have been proposed in literature, from which we used the most suitable one.

Clustering techniques can be divided into two separate groups. The first group creates a hierarchy of clusters in which each cluster is a subset of another cluster, while the second group of data clustering techniques creates non-overlapping groups, which have a maximum inter-group and minimum intera-group distance. The latter techniques were exactly what we were looking for, while the former had no indication. Among the most popular and flexible algorithms is the *k-means* algorithm which has acceptable performance and accuracy as reported in Ahmadzadeh, et. al. [2], for our application and does not have a large number of clusters. Using expert domain we chose to have three clusters since we had only seven students and the students' performed over a range of good, moderate or weak).

The *k-means* algorithm [16] starts with random *k* data, which indicates the number of clusters. Therefore, the value of *k* in our experiment was equal to three. Normally this *k* data is the first *k* data in data sets, which are called *centroid*. The algorithm then calculates the distance between centroid and other data and assigns each data to a group that has a minimum distance to its centroid. For the clusters that have already been created, an optimum centroid is computed and the data is regrouped. These steps repeat until the centroids no longer change.

We used Weka software [3] to do the job of clustering for us. As mentioned above, we chose to have three clusters, data from third table plus duration (end time – start time), and completeness of the delivered code as a data set for clustering. We will call the level of completeness of the code *code status* from this point forward for the simplicity of explanation. It should be noted that participants with code status equal to 2 were excluded from the analysis. The results achieved from this clustering are discussed next.

Results

Since we did not expect the results of clustering for both experiments to match 100%, we chose students who were placed in the same clusters in two experiments and compared their problem-solving patterns. Therefore, in section 2.1 we explained how these students were chosen and in section 2.2 reviewed their code.

First analysis phase

For simplicity of explanation we identified our students with numbers 11, 12, 14, 19, 23, 25 and 34. We clustered the data into two stages to ensure that the clusters were reliable. The first clustering was done in the first experiment and the second clustering was done in the second experiment and so on. Table 1 shows the results of the first clustering. Prior familiarity with programming, their level of interest and their marks out of 40 have also been added to this table.

All of the members of the first cluster finished their codes completely, had previous familiarity and high interest in programming. This group had spent minimal time in programming, on average, compared to the other groups. For the second group the story was different, as expected. None of the members of the second group had previous familiarity with programming but were not similar in the other two fields. The maximum time spent in programming, on average, belonged to this group. The only member of the third group who did not finish the code, had no prior familiarity in programming and less interest than the other participants. No significant differences in exam marks were seen between participants. In fact, all of the students belonged to the groups of students who received good marks. This showed us that their inability in completing the code was not due to lack of programming ability but, instead, in program comprehension. The number of activities that belonged to clusters 1 to 3 were 862, 1085 and 285 respectively, which accounted for 39%, 49% and 13% of all records accordingly. This shows that more records (i.e. more effort) to solve the problem belonged, on average, to cluster 2.

ISSN 22773061

Table 1: Results of the first clustering

	Student ID	Duration	Code Status	Familiarity	Interest	Mark /40
C. 1 39%	25	60	0	1	5	37.5
	23	80	0	1	5	35.5
	14	90	0	1	5	30.5
C. 2 49%	19	155	0	0	4	29.5
	12	135	1	0	4	30.25
	11	135	1	0	5	33
C. 3 13%	34	135	1	0	3	36

Through this clustering we found students who had conducted similar activities in one assignment. To validate our finding, we replicated the experiment by clustering the same information achieved (i.e. code activities, duration, code status) from the second programming task plus prior familiarity and interest. Again we asked for three clusters and the results, as shown in Table 2, indicate that the grouping of students changed in this programming task. Clusters (1 to 3) allocated 1199, 503 and 344 activities respectively, which meants that, on average, cluster 3 had conducted less and cluster 2 conducted more activities in order to solve the problem.

One interesting result achieved, when comparing both tables, shows that students numbered 11 and 12 were placed in the same cluster in each of the experiments. Neither finished the programs given to them in either experiment. The same applied to students numbered 23 and 25 who were in the same cluster in both experiments. These two students were able to complete the given tasks in both experiments. Therefore, we can consider students numbered 23 and 25 as representative of students who comprehend the program well against students numbered 11 and 12 who did not possess the same ability. It seems reasonable to find the pattern of problem -solving for the first group of students (numbered 11 and 12) and carry out the same effort for the other two students and see how different/similar the found patterns are. In the next section these comparisons will be explained.

Table 2: Results of the first clustering

	Student ID	Duration	Code Status
C. 1 59%	34	75	0
	23	125	0
	19	85	0
	25	45	0
C. 2 25%	14	130	1
C. 3 17%	12	55	1
	11	75	1

Analysis of pattern

In this section we explain inter similarity and differences between the first group (i.e. numbered 23 and 25) and the second group (i.e. numbered 11 and 12).

Both of the members of the first group created a project in Eclipse, copied all the given classes, ran the code and switched to the class that contained the *main* method without even having a single look at other classes. They started tracing the code from the *main* method up to the line where error occurred. At this point they had definitely found the reason behind the error because they switched to the other class to define a constructor in order to correct the error. In this process there were slight differences between the two members. Number 23 accomplished the process of reading from input and creating an object out of the input without any problems but number 25 did not do it in as straight forward a fashion. Next, they had a quick glance at other classes and methods and came back to the main class where they had to output the data, write the related code and execute it to see the result. In this process they needed to call methods from other classes. Each time they called a method they switched to the class that contained the method, corrected the method and added another when it was needed. During this process, they ran the program to see the results several times. This action was performed by number 25 more than number 23. After being satisfied with the results they both tried to make the code neater by deleting blank lines, correcting the indentation and presenting the output neater. They ran the program for the last time and then submitted the code.

There were no significant differences between these two members except that number 25 found out the problems mostly by repeatedly running the code and following the variables values by inserting a print statement. Number 23, on the other hand, had a lot of wait-time which meant he/she was busy reading the code and thinking about the logic.

Both members of the second group ran the code and started to read a class that did not contain the *main* method but each of them selected a different class to read (i.e. number 11 chose class *employee* while number 12 started with class *temporary*). This group did not pay attention to the generated error (i.e. they did not wait enough to convince us that they were reading the error but, instead, immediately clicked on the code area and the classes that were mentioned before). They both traced the code line by line from the point they had started. They both knew that they had to define a constructor but they did it in the wrong class. This confirmed the idea that they did not pay enough attention to the error. After several trial and error attempts, they finally placed the constructor in the right place. To continue, they both had some problems that were obvious from a lot of meaningless switching from one class to another and moving the cursor up and down. They were similar in these unsuccessful actions, however, number 12 made more useless effort, which did not lead to a proper result. Number 11, after many useless actions, including looking at method signature, finally got the point and corrected the error while number 12 gave up. They both wanted to output the data but were still not oriented enough to do it in a straight forward manner. Rather, they called several unnecessary and unrelated methods. Number 11 gave up at this stage but number 12 succeeded in doing part of the job. There were some actions in between that we did not understand because they were not related to the actual problem and the bugs. The members then repeatedly wrote a code, deleted it, ran it and finally submitted the uncompleted code.

The difference between these two members was the order in which they solved the problem: number 11 defined the first constructor, inputted data and then wrote the code to output the data; number 12 defined both constructors and then the related code for outputting the data.

What can be said in summary about these two groups is that the first group did not pay attention to any class or method that was not related to the problem, did the programming more smoothly than the other group, and ran the program when they thought the correction was done. They seemed to know what the next step should be. This does not however mean that they had a comprehensive understanding of the code because there were many parts in the program that they did not refer to. The second group read almost all of the code but did not focus on the problem itself. They used a trial and error strategy and conducted many unnecessary actions.

These results were achieved from observing the first program. With this impression, when we glanced at their second program we could confirm that the same pattern happened in that program.

Discussion

What was found in the current research, to some extent matches the results achieved by Karahasanovic, et.al. [5]. If the first group - who had a better program comprehension ability - is considered, one of the subjects used an *as-needed* strategy solely to solve the problem while the other used both an *as-needed* and a *semantic* strategy.

Our result is slightly different from LaToza, et.al. [6], in terms of our participants. Although all of our participants were novices we still observed the same as that observed in experts in LaToza's research for our first group of participants. Despite the fact that the first group had prior programming experience, they were not expert programmers. On the other hand, for the second group the results matched perfectly with their research since our participants did not have any prior programming experience. We predict that if we run the same experiment next semester, the second group will show the same behaviour as was observed for the first group in the current experiment. This shows that program comprehension, similar to programming, is a skill that is achieved by doing. Therefore, it might be a good idea if we include some debugging assignments in our teaching, i.e. students get to debug their peers' programs or a program given by the lecturer. The process of debugging, in a way that helps program comprehension, can also be taught explicitly. It should be noted that in our experiment the first group succeeded in the process of program comprehension but with different speeds. The participant who completed the process faster used a strategy such as printing the variable value or commenting some part of the code, while the other just read the code to get the meaning out of that. These are all issues that can be explicitly included in teaching.

In this research we did not find any evidence that shows that students lacked the ability of program comprehension due to having problems in object-oriented concepts such as polymorphism or inheritance. Rather, the main problem involves the lack of ability in finding the root of the problem. The difference between our two groups of novices, one with some prior experience and one without, was mostly on how they approached comprehending the program.

Overall, our results suggest that as with programming, program comprehension needs attention in the teaching process because the differences between our two groups in program comprehension were the same as differences between experts and novices mentioned in previous research. However, these two groups did not have significant differences in terms of experience.

Acknowledgement

This research was funded by Shiraz University of Technology grant number 90-EE-1.

References

[1] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An Analysis of Patterns of Debugging Among Novice Computer Science Students," in ITiCSE '05: ACM SIGCSE Annual Conference on Innovation and Technology in Computer Science Education. . Lisbon, Portugal, 2005.

[2] M. Ahmadzadeh and E. Mahmoudabadi, "A Feasibility Study on How Clustering Algorithm Helps in Program Comprehension Research," presented at submitted for 20th IEEE International Conference on Program Comprehension (under review), Passau, Germany, 2012.

[3] R. R. Bouckaert, E. Frank, M. A. Hall, G. Holmes, B. Pfahringer, P. Reutemann, and I. H.Witten, "WEKA—Experiences with a Java Open-Source Project," Journal of Machine Learning Research, vol. 11, pp. 2533-2541, 2010.

[4] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," International Journal of Machine Studies, vol. 18, pp. 543-554, 1983.

[5] A. Karahasanovic, A. Levine, and R. Thomas, "Comprehension strategies and difficulties in maintaining object-oriented systems: an exploratory study," Journal of Systems and Software, vol. 80, pp. 1541–1559, 2007.

[6] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program Comprehension as Fact Finding," in Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. Dubrovnik, Croatia: ACM, 2007, pp. 361-370.

[7] S. Letovsky, " Cognitive processes in program comprehension," Empirical Studies of Programmers, pp. 58–79, 1986.

[8] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance " Empirical Studies of Programmers, pp. 80–98, 1986.

[9] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Progreams," Cognitive Psychology, vol. 19, pp. 295-341, 1987.

[10] T. Shaft and I. Vessey, " The Relevance of Application Domain Knowledge: the Case of Computer Program Comprehension," Information Systems Research vol. 6, pp. 286–299, 1995.

[11] H. Sharp, Y. Rogers, and J. Preece, Interaction Design: Beyond Human Computer Interaction, 2nd ed: Wiley, John & Sons, Incorporated, 2007.

[12] B. Shneiderman and R. Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," International Journal of Computer and Informtion Sciences, vol. 8, pp. 219-238, 1979.

[13] H. M. Sneed, "Offering Software Maintenance as an Offshore Service," presented at 24th IEEE International Conference on Software Maintenance Beijing, China 2008.

[14] E. Soloway and K. Ehrlich, "Emprical Studies of Programming Knowledge," IEEE Transaction on Software Engineering, vol. 10, pp. 595-609, 1984.

[15] M.-A. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," presented at 13th International Workshop on Program Comprehension, St. Louis, MO, USA, 2005.

[16] P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining: Addison-Wesley, 2006.

[17] I. Vessey, "Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols," IEEE Transactions on Systems, Man and Cybernetics, vol. 16, pp. 621 - 637, 1986.

ISSN 22773061

Author's Biography

Marzieh Ahmadzadeh is an Assistant Professor of Computer Science and head of school of Computer Engineering and IT at Shiraz University of Technology. She received her PhD in Computer Science and MSc in Information Technology from the University of Nottingham, UK and her BSc in Software Engineering from Isfahan University, Iran. She teaches a variety of postgraduate and undergraduate courses and her research interest includes Data Mining, Business Intelligence, Data Security, Computer Science Education and Human Computer Interaction.

Elham Mahmoudabadi has a MSc in Computer Network received from Amirkabir University of Technology and a BSc in Information Technology Engineering from Shiraz University of Technology. Her research interest includes Programming, Data Mining and Computer Network.

Appendix A

We have an organization with two types of staff, permanent and temporary. Permanent staffs receive their salary monthly as was said in their contract. Temporary staffs however receive their wages according to the hours that they have worked and the rate per hour. In this program you have several staffs that you see their information in a file called '*input*'. You are about to read information from this file and print a pay slip for every given input. A source code, which is not complete, is given and your job is to complete it according to specification..

The output format is as follow:

Employee: Name is …… (rate is …….. and hours are…….)

Pay: …….

Employee: Name is …… (salary is ……)

Pay: …….

```
public class Permanent extends Employee{
        double salary;
        public Permanent(String name,double salary){
                super(name);
        }
        public double getSalary(){
                return salary;
        }
        public double pay(){
                return salary;
        }
}
```

```
import java.io.File;
import java.io.IOException;
```

```
import java.util.Scanner;
import java.util.Vector;
public class MainClass{
	public static void main(String[] args){
		String name, rate;
		Scanner  sc_file = null;
		Vector <Object> vec = new Vector <Object> ();
		try{
			sc_file = new Scanner(new File("input.txt"));
			while (sc_file.hasNext()){
				for(int i = 0; i < 3; i++){
					name = sc_file.next();
					rate = sc_file.next();
					Double r = Double.parseDouble(rate);
					Employee emp = new temporary(name, r);
					vec.addElement(emp);
				}
			}
	}
		catch (IOException e){
			System.out.println("No such file exist.\n"+ e);
			}finally{
				sc_file.close();
				}
			}
}
--------
public class temporary extends Employee{
	private double rate;
	private double hours;
	private double pay;
	public temporary (String name, double rate){
		super(name);
		setRate(rate);
		}
	public void setRate(double rate){
		this.rate = rate;
	}
	public double getRate(){
		return rate;
	}
	public double getHour(){
```

```
                return hours;
        }
        public double pay(){
            return pay;
          }
        }
-------
abstract public class Employee{
                abstract public double pay();
    private String name;
        public Employee(String name){
                setName(name);
        }
        public String getName(){
                return new String(name);
        }
        private void setName(String name){
                this.name = new String(name);
        }
        public String toString(){
                return "name is " + name;
        }
}
```

Appendix B

We have a company that offers products and services. Two types of customers are available, one who buys products only and another who not only buys products but also uses the available services. When it comes to discount, the company has two different approaches. Both the customers receive 15% on their purchase and second type of customers receive 20% discount on their received services. Your job is to manipulate the customer entries and to print the final payment of each customer. The output format should look like this:

Special Customers:

Name: …….. ID: ……. Payment: …….

Regular Customers:

Name: …….. ID: ……. Payment: …….

```
 public class SpecialCustomer extends Customer{
        double Payment;
        public SpecialCustomer(String _name, int _ID, double _PurchaseAmount){
                super(_name, _ID);
                super.setPurchaseAmount(_PurchaseAmount);
        }
        public void setServiceAmount(double _Samut){
                Samut = _Samut;
        }
        public double Payment(){
```

```
                return Payment;
        }
}
--------
public class RegularCustomer extends Customer{
        double Payment;
        public RegularCustomer(String _name, int _ID){
                super(_name, _ID);
        }
        public double getRegularCustomer(){
                return Pamut;
        }
}
--------
public class Customer {

        double Pamut, Samut;
        String name;
        int ID;
        public Customer(String _name, int _ID) {
                name = _name;
                ID = _ID;
        }
        public void setPurchaseAmount(double _Pamut){
                Pamut = _Pamut;
        }
        public double getPurchaseAmount(){
                return Pamut;
        }
        public String getName(){
                return name;
        }
        public int getID(){
                return ID;
        }
}
--------
import java.util.Vector;
public class Test {
        static Vector <String[]> vec = new Vector <String[]> ();
        static Vector <Object> My_vec = new Vector <Object> ();
        public static void main(String[] args){
```

```
Test ts = new Test();
Customer cm ;
String name;
double PurchaseAmount, ServiceAmount ;
        for(int i = 0; i < 7; i++){
            name = ts.vec.elementAt(i)[0];
          PurchaseAmount
    =Double.parseDouble(ts.vec.elementAt(i)[2]);
            if(i < 3){
                ServiceAmount=
     Double.parseDouble(ts.vec.elementAt(i)[3]);
                cm = new SpecialCustomer(name, PurchaseAmount,
ServiceAmount);
            }
            else
                cm= new RegularCustomer(name, ID, PurchaseAmount);
                ts.My_vec.add(cm);
        }
}
public static void getInformation(){
//Customer1
String[] spec = new String[4];
spec[0] = "G.A.";
spec[1] = "1000";
spec[2] = "75.0";
spec[3] = "28.0";
vec.insertElementAt(spec, 0);
//Customer2
spec = new String[4];
spec[0] = "W.M.";
spec[1] = "1001";
spec[2] = "85.0";
spec[3] = "30.0";
vec.insertElementAt(spec, 1);
//Customer3
spec = new String[4];
spec[0] = "T.S.";
spec[1] = "1002";
spec[2] = "65.0";
spec[3] = "18.0";
vec.insertElementAt(spec, 2);
//Customer4
spec = new String[4];
```

```
spec[0] = "D.A.";
spec[1] = "1003";
spec[2] = "125.0";
vec.insertElementAt(spec, 3);
//Customer5
spec = new String[4];
spec[0] = "J.F.";
spec[1] = "1004";
spec[2] = "150.0";
vec.insertElementAt(spec, 4);
//Customer6
spec = new String[4];
spec[0] = "J.J.";
spec[1] = "1005";
spec[2] = "300.0";
vec.insertElementAt(spec, 5);
//Customer7
spec = new String[4];
spec[0] = "S.T.";
spec[1] = "1006";
spec[2] = "200.0";
vec.insertElementAt(spec, 6);
}
}
```

Appendix C

Code	Activity Description	Code	Activity Description
1	Mouse movement without any purpose	44	Declaring a *double* constructor argument
2	purposeful Mouse movement	45	*Super* keyword
3	Delete a code	46	*this* keyword
4	Reading error and returning to source code	47	Defining an array
5	Correcting an error	48	*try* keyword
6	Save	49	*catch* keyword
7	Executing the program	50	*instanceOf* keyword
8	New Line	51	Inserting an *if*
9	*else* keyword	52	Casting *String* to *double*
10	Comment	53	Casting to *Object*
11	Removing a comment	54	Three consecutive switches from one class to another
12	*void* keyword	55	*break* keyword
13	*static* keyword	56	*continue* keyword
14	Inserting an *int* return type	57	Surfing the net for help

15	Inserting a *String* return type	58	Initializing an *int* variable
16	Inserting an *double* return type	59	Initializing a *String* variable
17	Inserting *public* access modifier (field)	60	Initializing an *double* variable
18	Inserting *private* access modifier (field)	61	Defining an *object*
19	Inserting *public* access modifier (method)	62	*System.out* (real)
20	Inserting *private* access modifier (method)	63	*System.out* (for debug purpose)
21	Inserting *public* access modifier (setter method)	64	*return* keyword
22	Inserting *private* access modifier (setter method)	65	Inserting a *for-loop*
23	Inserting *public* access modifier (getter method)	66	Checking output in console
24	Inserting *private* access modifier (getter method)	67	Receive help from Editor's list of methods
25	Inserting *public* access modifier (constructor)	68	Declaring an *object* reference variable
26	Inserting *private* access modifier (constructor)	69	Declaring a *Vector* reference variable
27	Declaring an *int* field	70	Inserting data to a vector
28	Declaring a *String* field	71	Invoking a method
29	Declaring a *double* field	72	Declaring a *boolean* variable
30	Declaring an *int* variable	73	*abstract* keyword
31	Declaring a *String* variable	74	Inserting *protected* access modifier (field)
32	Declaring a *double* variable	75	Declaring a *Scanner*
33	Declaring an *int* method argument	76	Inserting a *while-loop*
34	Declaring a *String* method argument	77	Instantiating a *vector*
35	Declaring a *double* method argument	78	*Switch* keyword
36	Declaring an *int* setter method argument	79	*case* keyword
37	Declaring a *String* setter method argument	80	*Default* keyword
38	Declaring a *double* setter method argument	81	Inserting a breakpoint
39	Declaring an *int* getter method argument	82	Opening debug page
40	Declaring a *String* getter method argument	83	Removing breakpoint
41	Declaring a *double* getter method argument	84	Move to source code
42	Declaring an *int* constructor argument	85	Declaring an *Object* constructor argument
43	Declaring a *String* constructor argument	1000	Wait