Automark++ a Case Tool to Automatically Mark Student Java Programs

Jubair Al-Ja'afer and Khair Eddin Sabri
King Abdullah II School for Information Technology, The University of Jordan, Jordan

Abstract: *The quality assessment of a computer program is a critical process for ensuring its effectiveness. In this paper, an easy to apply tool, AUTOMARK++, is introduced to automatically evaluate the Java programs. The marking of a program under evaluation is based on its style. AUTOMARK++ is based on Redish and Smyth tool called AUTOMARK [12]. Two modifications were made to the AUTOMARK: First, new factors have been introduced to give the new tool flexibility in evaluating object-oriented languages such as Java. Second, the new tool automatically generates a model template for program evaluation instead of writing a specific model for each program under evaluation. AUTOMARK++ has been tested on simple and complex programs and the obtained results showed that the tool is considerably useful.*

Keywords: *Software engineering, style metric, software quality, Java programming language.*

Received November 2, 2003; accepted January 22, 2004

1. Introduction

Software quality assessment tools are important especially in two areas, education and industry. In education, these tools can help instructors evaluate students' programs, and give them feedback on the strengths and weaknesses of their programs. In addition, students may use these tools to test their programs prior to submission to the instructor.

Another area of application is industry, where both programmers and managers can utilize such tools. The programmer can use an assessment tool to evaluate the quality of his/ her programs. Whereas the manager can use this tool to maintain the quality controls and uniform standards for a project team [7].

Many quality assessment tools have been developed. For example, Redish and Smyth developed a tool called AUTOMARK to evaluate student style-based Pascal programs [12]. Also, Berry and Meekings have developed another tool to assess student programs written in C language depending on style [3, 6]. Jones used the concept of testing to automate the evaluation of student programs [9]. Also, Jackson and Usher developed a tool called ASSYST to automate student programs depending on their correctness, efficiency, complexity and style [8]. Jumaa developed a tool to evaluate structural languages such as Pascal, FORTRAN, C, and Basic based on Halstead, McCabe, Style, and Lipow and Thayler models [10].

2. The AUTOMARK++ Tool

The AUTOMARK tool is proposed by Redish and Smyth to evaluate student programs based on style [12]. This tool requires a model program created by the instructor to evaluate the student programs against it. The tool proved to be suitable for intermediate courses. As for advanced courses with big projects, it is impractical for the instructor to write a model program for each assignment. Also in the industry, it is difficult to write a model program in order to assess an industrial program.

Two modifications have been introduced to the AUTOMARK model:

1. Some factors are added in order to evaluate object-oriented languages such as the depth of inheritance tree, total number of children, cohesion between methods, coupling between classes, total number of inherited methods, total number of methods and attributes in a class, and the total number of attributes used inside methods.
2. The values of factors used in the evaluation can be computed automatically by the AUTOMARK++ tool based on the statistical information obtained from a random set of programs. Hence, there is no need to write a model template for every program under evaluation.

The Following are the factors used in the AUTOMARK++ tool to assess program quality:

- *Factor1:* Depth of Inheritance Tree (DIT), DIT is defined as the maximum number of steps from the class node to the root of the inheritance tree. Well-engineered object-oriented software systems are those structured as forests of classes, rather than one very large inheritance lattice. The deeper a class within the hierarchy is, the greater the number of methods is likely to inherit making it more complex to predict its behavior, and more difficult to test and

maintain. Hence, software with classes of very large DIT tends to be of a lower quality and is consequently awarded fewer marks [2, 4, 11].

- *Factor 2*: Total Number of Children (TNC), children are defined as the number of immediate descendants of a class in the hierarchy. It is an indicator of potential influence of a class on the design of the system. The greater the number of children is, the greater the likelihood of improper abstraction of the parent. Classes with large number of children require more testing of the methods in that class. Software with a large TNC tends to be of lower quality [2, 4]. Therefore, the less number of children in a program, the higher mark it will get.
- *Factor 3*: Total Number of Methods (TNM), TNM is a useful indication of the class size. If the number of methods per class grows up significantly, the class objects tend to have many more functions. This implies that the class is more difficult to understand, reuse, test and maintain, and the system design is less modular. Software with significantly larger TNM is likely to be of lower quality and deserves fewer marks [2, 11].
- *Factor 4*: Total Number of Attributes (TNA), TNA is also a useful indication of a class size. If TNA grows too large, the class needs to provide much more information to other classes or within the same class. Therefore, it is likely to be more difficult to test and maintain. Also, it will reduce the reusability of the class. Software with significantly large TNA is likely to be of lower quality. If the program is not well designed, and the class becomes larger with more attributes, then it deserves lower mark [2, 11].
- *Factor 5*: Total Number of Inherited Methods (TNIM), using inheritance makes a program simpler and reduces defect density. So, it is expected that as the number of inherited methods increases, the quality of program increases and, therefore, it deserves higher mark [11].
- *Factor 6*: Total Size of Methods (TSM).
- *Factor 7*: Total Number of Attributes used inside Methods (TNAM).

The above two factors are used to measure the size and complexity of a method. A large size method which uses a large number of attributes is complex, difficult to understand, test, and maintain. When a method is simple, it is usually of a high quality and deserves high mark [1].

- *Factor 8*: Total Coupling of Classes (TCC), a large number of coupling increases complexity, reduces encapsulation and potential reuse, and limits understandability and maintainability [2, 4]. Coupling between classes is essential for every program. However, increasing coupling is unfavorable as it means that the program is not designed properly, and needs to be reviewed and redesigned. If the coupling is overused, the program will deserve a lower mark.
- *Factor 9*: Cohesion Between Methods (CBM), CBM measures the similarity of methods in a class which is calculated by computing the number of method pairs that accesses the same attributes. Low cohesion increases complexity, thereby increases the likelihood of errors during the development process. Whereas high cohesion indicates good class subdivision and implies simplicity, high reusability, and deserves high mark [2, 4].
- *Factor 10*: Total Number of Unique Operators.
- *Factor 11*: Total Number of Unique Operands.
- *Factor 12*: Total number of Operators.
- *Factor 13*: Total number of Operands.

The above four factors are the basis of Halstead theory [5]; the best known for measuring software complexity. It proposed the first analytical laws for computer software supported by several empirical studies. Halstead measure is also considered as one of the most widely accepted measures in industry and academia. Software science defines additional metrics such as: Program vocabulary, program length, program level, program size, and program and data difficulty. As these factors increase, the complexity of the program increases, and the mark decreases [5, 10].

- *Factor 14*: Total Number of Decision Nodes (TNDN), TNDN is a count of the number of test cases needed for testing a program. A program with low number of decision nodes decreases testing effort and increases understandability and, therefore, it gets higher mark. Decision nodes are essential for every program. Although loops and other decision nodes are needed in most programs, the programmer should limit their use [5, 10].
- *Factor 15*: Total Number of Assignment Statements (TNAS).
- *Factor 16*: Total Number of Function Calls (TNFC).
- *Factor 17*: Total Number of Comment Statements (TNCS).

The above three factors were used by Lipow and Thayler model to measure the complexity of a program. A large number of assignment statements or function calls and a low number of comment statements would increase the complexity of a program, and make it difficult to understand, test and maintain. Using assignment statements and function calls is important in every program, but its overuse may increase the complexity of the program and decrease its quality as well as its mark. Also, a program with a low number of comment statements would be difficult to understand and maintain, and therefore it gets low mark [10].

As a result of this research, there is no need to write a "model" program template for every program under evaluation. The tool uses some statistical information, collected from a set of programs having different

goals, complexity and quality, to generate a model program automatically. The model program is then compared with the program under evaluation. This model represents an acceptable frequency range of each evaluation factor by extracting from it the Weight Factor (WF) and the Tolerance Factor (TF) depending on the size of the evaluated program.

The size of the program being evaluated is essential for generating a "model" program. The above mentioned 17 factors are classified into four categories; each category uses an appropriate size.

- *Size 1:* Since the DIT, TNC, TNM, TNA, and TCC depend on the number of classes in a program, the size of the program is computed as its total number of classes.
- *Size 2:* As the three factors (TNIM, TSM, and TNAM) are related to the methods, it is expected that they depend on the total number of methods in the program.
- *Size 3:* Although CBM depends on the number of methods per class, it has been experimentally shown that it correlates better with the square of the number of methods per class.
- *Size 4:* The other factors depend on the total number of tokens in the program which can be computed as the sum of the total number of operators and operands.

2.1. The Algorithm of AUTOMARK++ Tool

Input: Program to be evaluated.
Output: Evaluated program.

- *Step 1*: Find the size of the program (PS) to be evaluated.
  Size 1 = Total number of Classes.
  Size 2 = Total number of methods.
  Size 3 = $(\text{Size2})^2$ / Size1.
  Size 4 = Total No. of Operators + Total No. of Operands.
- *Step 2*: For each of the 17 evaluation factors, follow the steps (3-7).
- *Step 3*: Calculate the average factor as follows:
  AF = FW PS*
  where:
  AF: Average of Factors.
  WF: Factor Weight "experimentally calculated".
  PS: Program size computed in step 1; each factor uses an appropriate size.
  Factors (1, 2, 3, 4, 8) use size 1
  Factors (5, 6, 7) use size 2
  Factor (9) uses size 3
  Factors (10-17) use size 4
- *Step 4*: Calculate the upper and lower boundaries of factors using the following formula:
  *UPPER BOUNDRAY = AF + AF * FT*
  *LOWER BOUNDRAY = AF - AF * FT*
  where:
  FT: Factor Tolerance "experimentally calculated".
- *Step 5*: Count the frequency of the evaluation factor in program *(F)*.
- *Step 6*: Calculate a numerical mark *(MARK)* for the above factor using linear interpolation formula. The *LOWER BOUNDRAY* is assigned a score of 60 and the *UPPER BOUNDRAY* is assigned a score of 80 or the opposite depending on the factor.

$$MARK = 80 - \frac{80-60}{UB-LB}(F-LB)$$

for the Factors: 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, and 16.

or:

$$MARK = 80 - \frac{80-60}{UB-LB}(UB-F)$$

for the Factors (5, 9, and 17).
where:
UB: UPPER BOUNDRAY (Step: 4)
LB: LOWER BOUNDARY (Step: 4)
F: Frequency of the evaluation Factor (Step 5)

- *Step 7: TOTAL MARK = MARK * W*
  where:
  MARK: The mark computed from step 6
  W: Non-negative weight assigned to the mark for each factor. The weight value depends on the importance of the factor.
- *Step 8:* $FINAL\ MARK = \frac{\sum_{i=1}^{n} TOTALMARK}{TW}$
  where:
  FINAL MARK: Final mark of the evaluated program [0: 100]
  n: The number of factors. [n = 17]
  TW: Total Weight which can be calculated as follow:

$$TW = \sum_{i=1}^{n} W(Fi)$$

where:
n: The total number of factors, n = 17
W(Fi): The Weight of Factor i

It is assumed that the "model" program is adequately but not perfectly engineered. Factors in an evaluated program, located in the range of the upper and lower boundaries, may have a mark between 60% and 80%.

3. Experimental Results and Analysis

The AUTOMARK++ tool has been tested on two different types of programs: Simple programs which

usually contain one class and complex programs which may have many classes. The tested programs are collected from different courses given in the KASIT School at the University of Jordan. More than one hundred simple programs are taken from students in introductory courses and approximately forty complex programs are taken from students in senior courses.

The diagram of Figure 1 is obtained after evaluating approximately forty programs taken from one section of an introductory course in Java programming language. As shown in the diagram, a normal distribution with a mean of 76% is obtained from the marked programs. However, object-oriented factors such as DIT, TNC, TNIM, and TCC are not useful for simple programs.

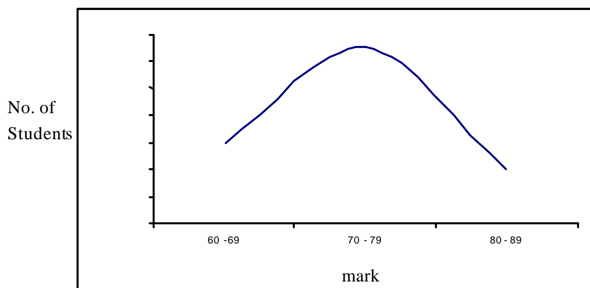

Figure 1. Distribution of student marks generated by the AUTOMARK++ tool.

In Appendix A, Figures (1, 2, 3), show three simple programs. The output of their evaluation produced by the AUTOMARK++ is shown in Tables (1, 2, 3). The score of Program 1 is (62%), Program 2 (64%) and Program 3 (82%).

The low mark of Program 1 is due to the weakness of factors such as: TNA, TSM, TNAM, TNDN, and TNCS. The tool indicates that there is a large number of attributes defined in the class which increases its complexity. Also, there is a problem with the high number of decision points which makes the program more complex and more difficult to test and maintain. Therefore, this number needs to be decreased. The tool also indicates that there is an excessive usage of the attributes inside the methods of the evaluated program and the total size of methods is large. This means that the methods do too much work which increases their complexity and, therefore, need to be decomposed for simplification. Additionally, the number of comment statements should be increased to make the program easier to understand.

Regarding Program 2, the tool shows that the number of assignment statements and the number of decision points are higher than the numbers expected from the model program. This makes the evaluated program more complex and relatively difficult to test. Also, as in Program 1, there is a problem with the number of comment statements and attributes. In addition, the tool indicates that the cohesion between the methods is not strong enough and, therefore, class decomposition or restructuring is recommended.

Program 3 scored a high mark (82%), and all scores were within or better than the model program template generated by the tool. This means that the evaluated program is well engineered. Based on the evaluation of student programs, it is clear that the marks are varied and the distribution curve is normal. Evaluated programs may be well engineered with no need for further modifications, or they suffer some weaknesses and recommendations to improve their quality are given by the tool.

AUTOMARK++ is also tested on larger and more complex programs. Several large programs, taken from senior courses, are evaluated by AUTOMARK++. In all cases, AUTOMARK++ confirms its capability of identifying the deficiencies of each program. To show how AUTOMARK++ can detect the deficiencies of large programs, three example programs, having the same functions and different designs, were evaluated as shown in Appendix B. Each program gives information about employee's first and last name as well as ID number. Also, it provides information about the hourly rate of both the temporary and permanent hourly employees. All permanent employees have a benefit deduction attribute. Permanent piece-worked employees have information regarding the product quality and the cost per piece. Also, there are permanent employees who have a fixed salary including those who receive a commission sale.

The design of Program 1 is shown in Appendix B, Figure 1 and the evaluation results in Table 1. The results show that there is a large number of methods defined in one class and the cohesion between them is small. Also, there is a large number of attributes and operators defined. This means that the class does an excessive work which makes the program more complex and more difficult to reuse. Therefore, the tool recommends that the class should be subdivided for simplification. Another deficiency area of the evaluated program is the absence of inherited methods. As the number of inheritance methods increases, the defect density and its fixing effort decrease accordingly. This is why using inheritance in the program is also recommended.

The design of Program 2 is shown in Appendix B Figure 2, and the evaluation results in Table 2. The results show that this design is better than the previous one and its mark is higher. This is due to the decomposition of the class and the use of inheritance. However, there are still many deficiencies in the program such as the high number of methods in the classes and the low cohesion between them. Therefore, more class decomposition is need.

In Appendix B, Figure 3 shows a well-designed program and Table 3 shows its evaluation results. It is obvious that this program does not need any modification

4. Conclusions

The following conclusions have been reached:

1. AUTOMARK++ can be used not only to evaluate the programs but also to enhance them. This can be achieved by eliminating the deficiencies of the evaluated program as suggested by the tool and, therefore, increasing the quality of its design.
2. Any program that gets a high mark should be well-engineered, which means that there is a compromise between all the factors. Also, all the factors should be in an acceptable range or better to get a high mark.
3. The distribution of the marks of student programs taken from the same course fits the normal distribution curve. Therefore it can be used in universities to mark student programs.

5. Summary

In this research, a CASE tool, AUTOMARK++, is developed to evaluate object-oriented languages by introducing new factors. These factors are used to assess the design of a program. Normally, a well-designed program gets a high mark. This can be achieved by compromising the use of these factors. For example, the class size should not be too large and should not have too much functionality. This can be obtained by computing the TNM and TNA. However, some classes may have only a few but huge methods, while others may have many but very simple methods. So the former classes may be more complex than the latter ones. Therefore, two factors are introduced to measure the size and complexity of the methods (TSM and TNAM). The larger the method size is, the less mark the program will get. In order to decrease the size of a class, it should be decomposed into subclasses. But the decomposition should be done correctly so that it can get a high mark in the coupling and cohesion factors. Also, the TNIM is applied to assess the use of inheritance. However, frequent use of inheritance increases the TNC or the DIT which results in lowering the mark.

Well-known factors in structural languages are also used by the tool to compute the complexity of a program. Such factors are the basic elements of Halstead which include the Total Number of Operators, Total Number of Operands, Total Number of Unique Operators and the Total Number of Unique Operands. In addition, other factors are considered by the tool for computing the program complexity such as TNDN and the factors taken from Lipow & Thayler model: TNAS and TNFC. Also, the TNCS is used to measure the readability of the program.

In general, the AUTOMARK++ tool is used to evaluate the style of the whole program (system level). However, there are some cases where the quality of the whole program is high, but it may contain a complex method which can't be detected by the tool.

A suggestion for future development is to evaluate programs at the class and method levels in addition to the system level. The tool may be further developed to evaluate different object-orient languages such as C++, C# and Smalltalk. Additional work can be generated by introducing other new factors to the tool.

Appendix A

Simple Java programs and their evaluation using AUTOMARK++

Program 1

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class program1 extends JApplet implements ActionListener{
JButton gradeButton;
JTextField gradeField;
JLabel gradeLabel;
Color color;
int value;
char grade;
Container container = getContentPane();
public void init() {
        container.setLayout ( new FlowLayout() );
        gradeLabel = new JLabel (" enter the numeric
        grade:");
        container.add (gradeLabel);
        gradeField = new JTextField (10);
        gradeField.setEditable (true);
        container.add (gradeField);
        gradeButton = new JButton ("Click here!");
        gradeButton.addActionListener (this);
        container.add (gradeButton);
}

public void actionPerformed (ActionEvent actionEvent) {
        value = Integer.parseInt (gradeField.getText());
        if (value > 100 || value < 0)
                showStatus ("Wrong mark");
        if (value >= 90 && value <= 100)
        {
                grade = 'A';
                showStatus ("This student got a A..");
                color = Color.red;
                container.setBackground (color);
        }
        if (value >= 80 && value <= 89)
        {
                grade = 'B';
                showStatus ("This student got a B..");
                color = Color.blue;
                container.setBackground (color);
        }
        if (value >= 70 && value <= 79)
        {
                grade = 'C';
                showStatus ("This student got a C..");
                color = Color.green;
                container.setBackground (color);
```

```
        }
        if (value >= 60 && value <= 69)
        {
                grade = 'D';
                showStatus ("This student got a D..");
                color = Color.yellow;
                container.setBackground (color);
        }
        if (value<60)
        {
                grade = 'F' ;
                showStatus ("This student got F so he didnt
                pass the exam..");
                color = Color.lightGray ;
                container.setBackground (color);
        }
}
}
```

Figure 1. Program 1.

Table 1. The output after evaluating Program 1

Factors	Model Program		Evaluated Program	
	Lower Bound	Upper Bound	Score	Mark (%)
Depth of inheritance tree	0	0	0	100
Total number of children	0	0	0	100
Total number of methods	4	7	2	90
Total number of attributes	2	3	7	0
Total number of inherited Methods	0	0	0	80
Total size of methods (LOC)	24	27	31	40
Total number of attributes used inside methods	3	4	9	0
Total number of coupling	0	1	0	100
Cohesion between methods	0	1	1	80
Total number of operators	198	209	188	90
Total number of unique operators	39	46	38	80
Total number of operand	107	114	106	80
Total number of unique operands	57	64	52	90
Total number of decision nodes	9	10	11	40
Total number of assignment statements	15	16	15	80
Total number of function call	22	25	23	80
Total number of comment statements	5	6	0	0
Final Mark is 62 %				

Comments

- Since the total number decision nodes is large, decrease it.
- Since there are no comment statements, write them.
- Since the number of attributes used inside methods is large, simplify the methods.
- Since the size of the methods is large, simply the methods.
- Since the number of attributes is large, decrease it.

Program 2

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
class Program2 extends JApplet implements ActionListener
{
   Color bgcolor;
  double grade;
   String letter;
 JTextField gfield;
 Container container;
public void init()
{
        container = getContentPane();
        container.setLayout (new FlowLayout());
        gfield = new JTextField (10);
        gfield.addActionListener (this);
        container.add (gfield);
}
public void actionPerformed (ActionEvent action)
{
        grade = Double.parseDouble (gfield.getText());
        letter = letterg (grade);
        bgcolor = bg (bgcolor);
        container.setBackground (bgcolor);
        showStatus (letter);
}
public String letterg (double grade)
{
        String var;
        var = "";
        if (grade >= 90 && grade <= 100)
                var = "A";
        if (grade >= 80 && grade < 90)
                var = "B";
        if (grade >= 70 && grade < 80)
                var = "C";
        if (grade >= 50 && grade < 70)
                var = "D";
        if (grade < 50 && grade >= 0)
                var = "F";
        return var;
}
Color bg (Color color)
{
        int rand;
        rand = 1 + (int) (Math.random() * 5);
        switch (rand)
        {
        case 1:   color = Color.orange;
                  break;
        case 2:   color = Color.yellow;
                  break;
        case 3:   color = Color.blue;
                  break;
        case 4:   color = Color.green;
                  break;
        case 5:   color = Color.black;
                  break;
        }
        return color;
}
}
```

Figure 2. Program 2.

Table 2. The output after evaluating Program 2.

Factors	Model Program		Evaluated Program	
	Lower Bound	Upper Bound	Score	Mark (%)
Depth of inheritance tree	0	0	0	100
Total number of children	0	1	0	100
Total number of methods	4	7	4	80
Total number of attributes	2	3	5	20
Total number of inherited methods	0	1	0	60
Total size of methods (LOC)	48	55	31	100
Total number of attributes used inside methods	6	9	8	70
Total number of coupling	0	1	0	100
Cohesion between methods	3	4	2	40
Total number of operators	195	206	184	100
Total number of unique operators	39	46	42	80
Total number of operand	105	112	105	80
Total number of unique operands	56	63	49	100
Total number of decision nodes	9	10	15	0
Total number of assignment statements	14	15	17	20
Total number of function call	21	24	15	100
Total number of comment statements	5	6	0	0
Final Mark is 64%				

Comments

- Since the number of assignment statement is high, decrease it.
- Since the number of decision nodes is high, decrease it.
- Since there is no comment statements, write them.
- Since the cohesion between the methods is low, divide the class.
- Since the number of attributes is high, decrease it.

Program 3

```
/* This program reads a mark from a user and then converts
   it to a letter from A to F. A message appears in the status
   bar and the color of the background changes. If the
   entered mark is more than 100 or less that 0, an error
   message appears. */
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class program1 extends JApplet implements ActionListener
{
   Container container = getContentPane ();
   JTextField textfield ;   // TextField to read the mark
   int mark ;      // Contains the entered mark.
   // The initialisation of the user interface
public void init () {
         container.setLayout (new FlowLayout());
         JLabel label = new JLabel ("Enter the mark..");
         container.add (label);
         textfield = new JTextField (10);
         textfield.addActionListener(this);
         container.add (textfield);
}
// Automatically invoked when the use press Enter.
public void actionPerformed (ActionEvent e ) {
         mark = Integer.parseInt (textfield.getText ()) ;
 // Read the mark from the textfield
         display() ;
}
// Display of results depends on read mark.
void display() {
         String msg = "" ;
         if (mark  > 100 || mark < 0 )
         {
               showStatus("Mark is out of range");
               container.setBackground(Color.blue);
         }
         else if (mark >= 90){
               msg = "A" ;
               container.setBackground(Color.green);
         }
         else if (mark >= 80) {
                msg = "B" ;
                container.setBackground(Color.cyan);
         }
         else  if (mark >= 70){
                msg = "C" ;
                container.setBackground(Color.black);
         }
         else  if (mark >= 60) {
                msg = "D" ;
                container.setBackground(Color.red);
         }
         else {
                msg = "F" ;
                container.setBackground(Color.gray);
         }
         showStatus(" Your Grade is  " + msg);  // print the
          message in status bar
      }
}
```

Figure 3. Program 3

Table 3. The output after evaluating Program 3.

Factors	Model Program		Evaluated Program	
	Lower Bound	Upper Bound	Score	Mark (%)
Depth of inheritance tree	0	0	0	100
Total number of children	0	0	0	100
Total number of methods	4	7	3	80
Total number of attributes	2	3	3	60
Total number of inherited methods	0	1	0	60
Total size of methods (LOC)	36	41	22	100
Total number of Attributes used inside methods	4	7	6	70
Total number of coupling	0	1	0	100
Cohesion between methods	1	2	3	100
Total number of operators	161	170	162	80
Total number of unique operators	31	38	33	70
Total number of operand	87	92	124	80
Total number of unique operands	46	53	47	80
Total number of decision nodes	7	8	6	100
Total number of assignment statements	12	13	10	100
Total number of function call	17	20	18	80
Total number of comment statements	4	5	7	100
Final Mark is 82 %				

Comment

This program is well-engineered. No modification is recommended.

Appendix B

Different designs and their evaluation using AUTOMARK++.

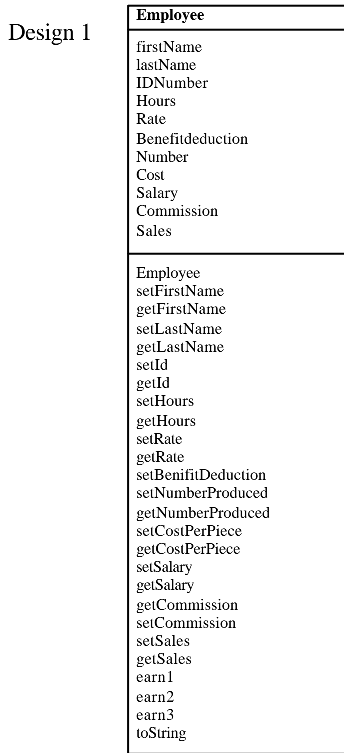

Figure 1. The design of Program 1.

Table 1. The output after evaluating Program 1.

Factor	Model Program		Evaluated Program	
	Lower Bound	Upper Bound	Score	Mark (%)
Depth of inheritance tree	0	0	0	100
Total number of children	0	0	0	100
Total number of methods	4	7	26	0
Total number of attributes	2	3	11	0
Total number of inherited methods	8	13	0	30
Total size of methods (LOC)	321	362	28	100
Total number of attributes used inside methods	44	67	37	80
Total number of coupling	0	1	0	100
Cohesion between methods	243	300	44	10
Total number of operators	222	233	239	50
Total number of unique operators	44	53	42	80
Total number of operands	119	128	89	100
Total number of unique operands	63	72	26	100
Total number of decision nodes	10	11	0	100
Total number of assignment statements	17	18	14	100
Total number of function call	24	27	0	100
Total number of comment statements	5	6	23	100
Final Mark is 66%				

Comments

- Since the total number of methods defined in the classes is large, decompose some of classes.
- Since the total number of attributes defined in the classes is large, decompose some of classes.
- Since the total number of inherited methods is small, use more inheritance
- Since there is a lack of cohesion between the methods, subdivide the class.
- Since the total number of operator is large, simplify the program.

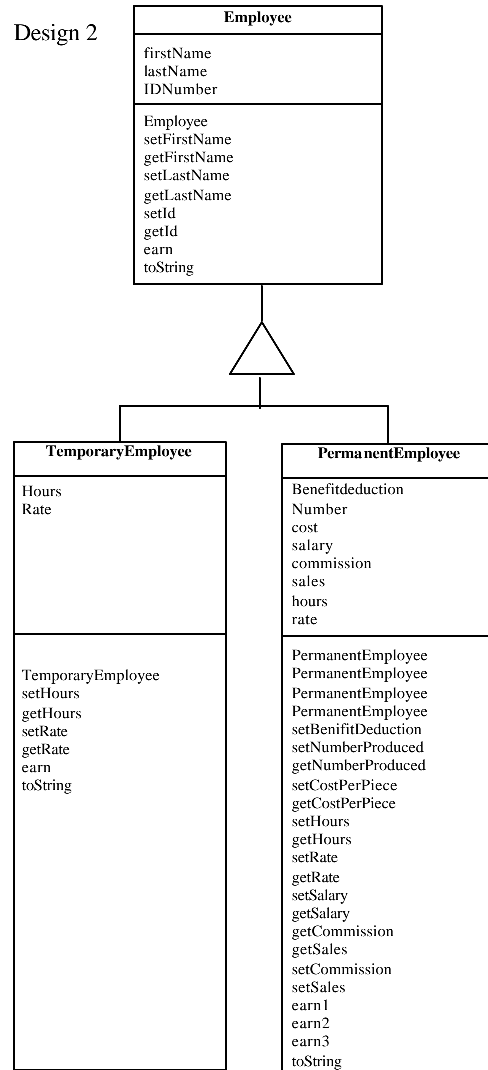

Figure 2. The design of Program 2.

Table 2. The output after evaluating Program 2.

Factor	Model Program		Evaluated Program	
	Lower Bound	Upper Bound	Score	Mark (%)
Depth of inheritance tree	0	1	1	60
Total number of children	1	2	2	60
Total number of methods	14	23	39	30
Total number of attributes	8	13	13	60
Total number of inherited methods	12	19	18	80
Total size of methods (LOC)	482	543	54	100
Total number of attributes used inside methods	67	102	56	80
Total number of coupling	3	4	0	100
Cohesion between methods	115	142	87	40
Total number of operators	420	443	432	70
Total number of unique operators	84	101	49	100
Total number of operands	227	242	190	100
Total number of unique operands	121	138	32	100
Total number of decision nodes	21	22	0	100
Total number of assign statements	33	34	30	100
Total number of function call	47	54	2	100
Total number of comment statements	12	13	22	100
Final Mark is 76%				

Comments

- Since the total number of methods defined in the classes is large, decompose some of these classes.
- Since there is a lack of cohesion between the methods, subdivide the class.

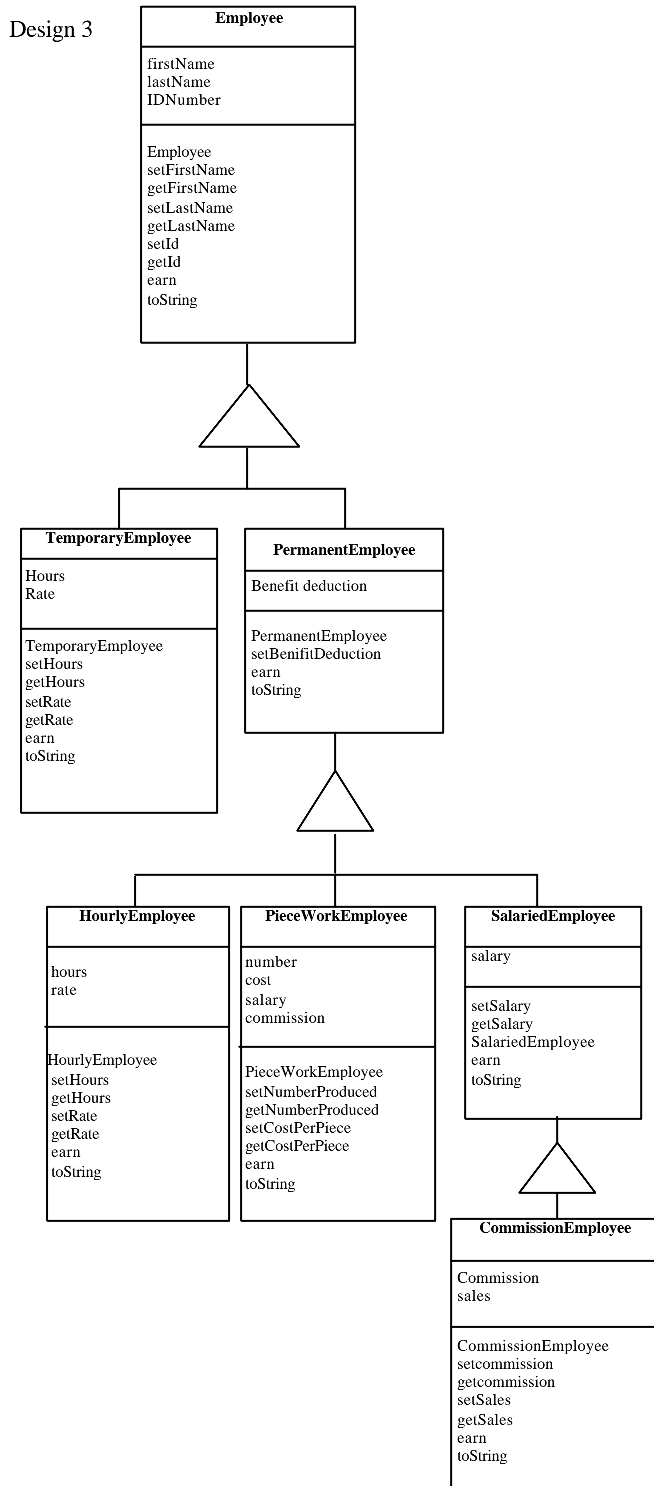

Figure 3. The design of Program 3.

Table 3. The output after evaluating Program 3.

Factor	Model Program		Evaluated Program	
	Lower Bound	Upper Bound	Score	Mark (%)
Depth of inheritance tree	2	3	3	60
Total number of children	3	6	6	60
Total number of methods	34	53	45	70
Total number of attributes	21	32	13	90
Total number of inherited methods	15	22	65	100
Total size of methods(LOC)	557	626	56	100
Total number of attributes used inside methods	78	117	50	90
Total number of coupling	7	10	0	100
Cohesion between methods	65	80	67	7
Total number of operators	490	517	510	70
Total number of unique operators	98	117	50	100
Total number of operands	264	283	215	100
Total number of unique operands	141	160	37	100
Total number of decision nodes	25	26	0	100
Total number of assignment statements	38	39	26	100
Total number of function call	55	62	6	100
Total number of comment statements	14	15	28	100
Final Mark is 86%				

Comments

This program is well-engineered. No modification is recommended.

References

[1] Abreu B. and Carapuca R., "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *Journal of Systems and Software*, vol. 26, no. 1, 1994.

[2] Alhadithi J. and Taka A., "Application of Object-Oriented Software Quality Metrics to Measure the TPS System," *in Proceedings of the ACIT Conference*, Qater, 2002.

[3] Berry R. and Meekings B., "A Style Analysis of C Programs," *Communication of the ACM*, vol. 28, no.1, pp. 80-88, 1985.

[4] Chidamber S. and Kemerer C., "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering,* vol. 20, pp. 476-493, 1994.

[5] Conte S. D., Dunsmore H. E., and Shen V. Y., *Software Engineering Metrics and Models*, The Benjamin/ Cummings Publishing Company Inc., 1986.

[6] Harrison W. and Cook C., "A Note on the Berry-Meekings Style Metric," *Communication of the ACM*, vol. 29, pp. 123-125, 1986.

[7] Hung S., Kwok L., and Chan R., "Automatic Programming Assessment Metrics," *Computers and Education*, vol. 20, no.2, pp.183-190, 1993.

[8] Jackson D. and Usher M., "Grading Student Programs Using ASSYST," *in Proceedings of the 28th ACM SIGCSE Technical Symposium on Computer Science Education*, San Jose, California, USA, pp. 335-339, 1997.

[9] Jones E., "Grading Student Programs: A Software Testing Approach," *ACM Journal of Computing in Small Colleges*, vol. 16, no.2, pp.187-194, 2001.
[10] Jumaa D. "A Computer Model for Evaluation of Programs," *MSc Thesis*, University of Engineering and Science, 1992.
[11] Pressman R. and Ince D., *Software Engineering: A Practitioner's Approach: European Adaptation*, Schaum, 2000.
[12] Redish K. and Smyth W., "Program Style Analysis: A Natural By-product of Program Compilation," Communication *of the ACM*, vol. 29, no. 2, pp.126-133, 1986.

Jubair Al-Ja'afer is a professor of computer science at the King Abdullah II School for Information Technology, the University of Jordan. His main interests are software engineering, biocomputing, wisdom and ontology. He obtained his BSc in physics from the University of Baghdad 1968, BSc, MSc, and PhD in computer science from the United Kingdom.

Khair Eddin Sabri is currently working as a lecturer in the Computer Science Department at the University of Jordan. He obtained his BSc degree in computer science from the Applied Science University, Jordan in June 2001, and MSc degree in computer science from the University of Jordan in January 2004. His main interest is software engineering, especially software metrics and reverse software engineering. He has also published some papers in the area of data hiding.