METRICS FOR OBJECT ORIENTED DESIGN (MOOD) TO ASSESS JAVA PROGRAMS

Prof. JUBAIR J. AL-JA'AFER
University of Jordan
jubair@ju.edu.jo

KHAIR EDDIN M. SABRI
University of Jordan
sabrikm@ju.edu.jo

Abstract: MOOD metrics are well known metrics used to measure some characteristics of the Object-Oriented programs. In this research, a system, based on the MOOD, has been developed to evaluate and grade Java programs. The interval of each MOOD metrics has been adapted, based on experimental results, to be fit in the evaluation of Java Programs. Also, a weight factor has been introduced to reflect the importance of each characteristic. The system has been tested with many different programs that vary in their complexities and functionalities. Also, the metrics have been tested in the evaluation of student programs in the University of Jordan and grading them. For all cases, the system shows good results.

Keywords: MOOD metrics, software quality, Java programming language, experimental software engineering

1. Introduction:

Software quality is critical to the development of software systems, especially large scale ones. "High" quality software would reduce the cost of software maintenance, and it enhances the potential software reuse. To assess software quality more quantitatively and objectively, software metrics appear to be a powerful and effective technology [1].

Software measurements have become increasingly essential in software engineering. Many of the best software developers measure characteristics of the software to make sure that requirements are consistent and complete, the design is of high quality, and the code is ready to be tested. Effective project managers measure attributes of process and product to be able to tell when the software should be ready for delivery and/or the budget has been exceeded. Target customers measure aspects of the final product to determine if its quality meets their requirements. Maintenance staff must be able to assess the current product to see what should be upgraded and improved.

In this research, the MOOD metrics are used to assess Java programs. The MOOD metrics consist of the following software quality indicators: Attribute Hiding Factor (AHF), Method Hiding Factor (MHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Coupling Factor (COF), and Polymorphism Factor (POF) [2].

2. Related Work

MOOD metrics to measure object-oriented programs have been used by many software developers. Abreu *et al*. [3] presented some advances towards the quantitative evaluation of design attributes of object-oriented software systems. An experiment for the collection and analysis of MOOD metrics was described and several suppositions regarding the design were evaluated. A considerable number of class taxonomies written in the C++ language were used as a sample. A tool to collect those metrics was built and used for that purpose. Statistical analysis was performed to evaluate the collected data.

Abreu and Melo [4] described the results of a conducted study in which the impact of object-oriented design on software quality characteristics was experimentally evaluated. MOOD metrics were adopted to measure the use of OO design mechanisms. Data collected on the development of eight small-sized information management systems, based on identical requirements, were used. Data obtained in this experiment show how OO design mechanisms such as inheritance, polymorphism, information hiding and coupling can influence quality characteristics such as reliability and maintainability.

Abreu *et al.* [5] introduced a MOOD-to-Eiffel binding. Some code fragments were presented to illustrate the concepts and to clarify the measurement process. A sample of Eiffel libraries was used to collect these metrics. Statistical analysis was performed on the sample and some hypotheses were drawn and discussed. Some preliminary heuristics that can be used during the design process are then derived.

Harrison *et al.* [6] described the results of an investigation of MOOD metrics. Each one of the six MOOD metrics was discussed from a measurement theory point of view, taking into consideration the recognized object-oriented features intended to measure: encapsulation, inheritance coupling and polymorphism. Empirical data, collected from three different application domains were analyzed using the MOOD metrics to support this theoretical validation. Results showed that the

metrics could be used to provide an overall assessment of a software system, which may be helpful in managing software development projects.

There are also many tools which have been developed in order to assess programs such as: Redish and Smyth [7] developed a tool called AUTOMARK to evaluate student style-based FORTRAN programs. Al-Ja'afer and Sabri [8] developed a system called AUTOMARK++ to assess object-oriented programs based on their style. Berry and Meekings [9] developed a tool to assess student programs written in C language depending on programming style indicators. Jackson and Usher [10] developed a software tool called ASSYST designed to assess programs based on their correctness, efficiency, complexity and style. Also, Jumaa [11] developed a tool to evaluate structural languages such as Pascal, FORTRAN, C and Basic based on Halstead, McCabe, AUTOMARK and Lipow and Thayler models.

3 The MOOD Metrics

The Metrics for Object-Oriented Design (MOOD) set includes the following metrics [2]

a) Attribute Hiding Factor (AHF)
b) Method Hiding Factor (MHF)
c) Method Inheritance Factor (MIF)
d) Attribute Inheritance Factor (AIF)
e) Coupling Factor (COF)
f) Polymorphism Factor (POF)

Each MOOD metric is associated with basic structural mechanisms of the object-oriented paradigm such as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphism (POF) and association (COF). As a consequence, these metrics are expressed as percentages ranging from 0% (no use) to 100% (maximum use) [11].

Encapsulation and Information Hiding

AHF and MHF are measures of the use of the information-hiding concept supported by the encapsulation mechanism [3]. Information hiding should be used to:

1. Cope with complexity by looking at complex components such as black boxes.
2. Reduce "side-effects" provoked by implementation refinement.
3. Support a top-down approach.
4. Test and integrate systems incrementally.

Metric 1: Method Hiding Factor (MHF)

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Where:

$M_h(Ci)$: hidden Methods in class C_i
$M_d(Ci) = M_v(C_i) + M_h(C_i)$: Methods defined in C_i
$M_v(Ci)$: visible Methods in class C_i
TC: Total number of Classes.

The number of visible methods is a measure of the class functionality. Increasing the overall functionality will reduce MHF. In order to implement this functionality, a top-down approach must be adopted, where the abstract interface (visible methods) should only be the tip of the iceberg. In other words, the implementation of the class interface should be a stepwise decomposition process, where more and more details are added. This decomposition will use hidden methods, thus obtaining the above-mentioned information-hiding benefits and favoring an MHF increase. This apparent contradiction is reconciled by considering MHF to have values within an interval. A very low MHF value would indicate an insufficiently abstracted implementation. Conversely, a high MHF value would indicate very little functionality. The best thing for MHF is to be within an interval [2,3,4].

Metric 2: Attributes Hiding Factors (AHF)

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Where:

$A_h(Ci)$: hidden Attributes in class C_i
$A_d(Ci) = A_v(C_i) + A_h(C_i)$: Attributes defined in C_i
$A_v(Ci)$: visible Attributes in class C_i
TC: Total number of Classes.

AHF is a measure of the use of the information hiding concept supported by the encapsulation mechanism. Information hiding allows coping with complexity by turning complex components into black boxes. AHF should be used as much as possible. Ideally all attributes would be hidden, thus being only accessed by the corresponding class methods. Very low values of AHF should trigger the designers' attention. In general, as AHF increases, the complexity of the program decreases [2,3,4].

Inheritance Factors

Both MIF and AIF are measures of inheritance. This is a mechanism for expressing similarity among classes that allows for the portrayal of generalization and specialization relations, and a simplification of the definition of inheriting classes by means of reuse [2,3,4,5].

Metric 3: Method Inheritance Factor (MIF)

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Where:
$M_a(C_i)$: $M_d(C_i) + M_i(C_i)$
M_i: inherited Methods
M_d: defined Methods
TC: Total number of Classes.

At first sight, we might be tempted to think that inheritance should be used extensively. However, the composition of several inheritance relations builds a directed acyclic graph (inheritance hierarchy tree), whose depth and width make understandability and testability fade away quickly [1,2,3,4].

Metric 4: Attributes Inheritance Factor (AIF)

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Where:
A_i: inherited Attributes
$A_a(C_i)$: $A_d(C_i) + A_i(C_i)$
A_d: defined Attributes
TC: Total number of Classes.

Metric 5: Coupling Factor (COF)

The COF metric is a measure of coupling between classes.

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client\ (C_i, C_j)\right]}{TC^2 - TC}$$

Where:

$$is\_client(C_c, C_s) = \begin{cases} 1 \text{ if } (C_c \Rightarrow C_s) \wedge (C_c \neq C_s) \\ 0 \text{ otherwise} \end{cases}$$

TC: Total number of Classes.

The client-supplier relation, represented by $C_c \Rightarrow C_s$, means that C_c (client class) contains at least one non-inheritance reference to a feature (method or attribute) of class C_s (supplier class). The COF numerator represents the actual number of coupling not imputable to inheritance [2,3,4,6].

It is desirable that classes communicate with as few other classes as possible, and that they exchange as little information as possible. Coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability. Also, coupling in software systems has a strong negative impact on software quality, and therefore should be kept to the minimum during the design phase. However, for a given application, classes must cooperate to deliver some kind of functionality. Therefore, COF is expected to be lower bounded [2,3,4].

Metric 6: Polymorphism Factor (POF)

Polymorphism means having the ability to take several forms. In OO systems, polymorphism allows the implementation of a given operation to be dependent on the object that "contains" the operation [2,3,4,6].

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i) \times DC(C_i)]}$$

Where:
$M_o(C_i)$: overriding Methods in class C_i
$M_n(C_i)$: new Methods in class C_i
$DC(C_i)$: number of Descendants of Class *Ci* (derived classes)
TC: Total number of Classes

The POF numerator represents the actual number of possible different polymorphic situations. A given message sent to class C_i can be bound, statistically or dynamically, to a named method implementation which may have as many shapes as the number of times this same method is overridden (in C_i descendants) [2,3,4].

Polymorphism arises from inheritance. Binding (usually at run time) a common message call to one of several classes (in the same hierarchy) is supposed to reduce complexity and to allow refinement of the class hierarchy without side effects. On the other hand, to debug such a hierarchy, by tracing the control flow, this same polymorphism would make the job harder. Therefore, polymorphism ought to be bounded within a certain range [2,3,4].

4. The Evaluating System:

The grade given to the evaluated program depends on Table 1. This table is produced from reported study [2,3,4,5,6] and the results of experiments.

Table 1: The range of MOOD factors

Factor	Minimum	Maximum	Minimum Tolerance	Maximum Tolerance
MHF	12.7 %	21.8%	9.5 %	36.9%
AHF	75.2 %	100 %	67.7%	100%
MIF	66.4 %	78.5 %	60.9%	84.4%
AIF	52.7 %	66.3 %	37.4%	75.7%
COF	0 %	11.2 %	0%	24.3%
POF	2.7 %	9.6 %	1.7%	15.1%

It should be noted that the minimum value of COF reported in the literature is 4.0%. This is due to the fact that coupling is required in every program to deliver some

functionality. However, this value was changed to 0 in our System for two reasons:
First, lowering the minimum value of COF to 0 has no effect on the quality of the program.
Second, the value of 0 will enable us to evaluate part of the program which may have no functionality.

The final grade is computed as follow:

Input: program to be evaluated.

Output: evaluation results.

Step 1: For all the indicators do the following:

IF the indicator value is found within the specified range, THEN *GRADE = 100%*

ELSE IF the indicator value is located in the tolerance range, THEN *60%≤GRADE<100%* by using interpolation.

ELSE, the indicator outside the tolerance range, then *GRADE<60%* by using the interpolation formula and the system should trigger the programmer's attention to a problem.

Step 2: $TOTAL\ GRADE = \sum_{i=1}^{n} GRADE[i] * W[i]$

Where:

GRADE[i]: the grade for the indicator i

W[i]: weight value assigned to the indicator i to reflect its importance

n: the number of factors, n = 6.

Step 3: $FINAL\ GRADE = \frac{TOTAL\ GRADE}{TW}$

Where:

FINAL GRADE: The final grade of the evaluated program [0:100]

TW: The Total Weight which can be computed as:

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

Where:

n: the total number of indicators, n = 6

W (Fi): The weight for indicator i

The grade of each indicator ranges from 0% to 100%. Each indicator is assigned a weight to reflect its importance. The classification of the six indicators is shown in Table 2. This table is produced from reported study [2,3,4,5,6]

Table 2: The importance of MOOD factors

Factor	Importance	Weight
Attribute Hiding Factor (AHF)	Low	1
Method Hiding Factor (MHF)	Medium	2
Method Inheritance Factor (MIF)	Medium	2
Attribute Inheritance Factor (AIF)	Low	1
Coupling Factor (COF)	High	3
Polymorphism Factor (POF)	Medium	2

5. Results and Analysis:

The quality of many programs that vary in their complexity and design have been assessed by the system. In each case the system successfully identified the weaknesses in the program being assessed. Appendix A shows three different designs of programs with their assessment results.

The programs have the same function: storing and accessing information about employees in an organization. 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 on sales.

When design 1 is assessed, the obtained grade is low due to the following deficiencies in the design:

1. The total number of inherited methods compared to the total number of methods is very low. 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 the program also increases and, therefore, a low grade is given.
2. It is expected that 12.7% to 21.8% of methods are hidden in the program, however, none has been found. The implementation of the class interface should be a stepwise decomposition process, to which more and more details are added. This decomposition uses hidden methods, thus obtaining the above mentioned information-hiding benefits and favoring an MHF increase. As a very low MHF indicates an insufficiently abstracted implementation, a low grade is given to this factor.
3. Polymorithism which reduces complexity is too low.

To increase the grade through the enhancement of program's quality, inheritance should be used and the class needs to be decomposed for simplification.
The grade of design 2 is higher (i.e. its quality is better) than the previous one, and this is due to the use of inheritance which simplified the classes. Although the grade is relatively high, there are still some existing deficiencies such as:

1. MHF should be increased, the number of inherited methods is still low,
2. Polymorithism is high and using it frequently makes the program more difficult to test and maintain and, therefore, should be decreased. Although using inheritance increases the grade of

the program, a different design needs to be constructed to further increase the quality of the program.

The third design gets the highest grade, and all factors are within the expected range except MHF. This factor is low due to the simple functions (read and write) of the program. Therefore, no utility (private) methods are needed.

Weight values are used to adjust the grades. There are some factors such as COF is more important than MIF, MHF or POF which in turn are more important than AHF or AIF.

6. Conclusion

1. The MOOD metrics are most suitable metrics to assess object oriented programs, and proved successfully used to assess Java Program
2. The System is not only useful for assessing programs, but also a tool to find the deficiency in each program under assessment.
3. The use of weights are very adequate to adjust the marking scheme of the assessed programs.
4. The System can easily used to assess programs at process level.
5. The system can only be used to assess large Java programs.

References:

[1] R. Pressman, *Software Engineering: a Practitioner's Approach: European Adaptation*, 5th edition, McGraw-Hill, UK, 2000.

[2] F. Abreu and R. Carapuça, Object-Oriented Software Engineering: Measuring and Controlling the Development Process, *Proceedings of the 4th International Conference on Software Quality,* McLean, VA, USA, 1994.

[3] F. Abreu, M. Goulão and R. Esteves, Toward the Design Quality Evaluation of Object-Oriented Software Systems, *Proceedings of the 5th International Conference on Software Quality,* Austin, Texas, USA, 1995.

[4] F. Abreu and W. Melo, Evaluating the Impact of Object-Oriented Design on Software Quality, *Proceeding of the 3rd International Software Metrics Symposium (METRICS'96),* IEEE, Berlin, Germany, pp. 90-99, 1996.

[5] F. Abreu, S. Esteves and M. Goulao, The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics, *Proceedings of TOOLS'96*, Santa Barbara, CA, USA, 1996.

[6] R. Harrison, S. Counsell and R. Nithi, An evaluation of the MOOD set of object-oriented software metrics*, IEEE Transaction on Software Engineering,* 24(6), 1998, pp. 491-496.

[7] K. Redish, and W. Smyth, Program Style Analysis: A Natural By-Product of Program Compilation, *Communications of the ACM,* 29(2), 1986, pp. 126-133.

[8] J. Al-Ja'afer, and K. Sabri, AUTOMARK++ an CASE tool to automatically mark student Java Programs., *International Arab Journal of Information Technology*, to appear.

[9] R. Berry, and B. Meekings (1985), A Style Analysis of C Programs, *Communications of the ACM*, 28(1), 1985, pp. 80-88.

[10] D. Jackson and M. Usher, Grading Student Programs Using ASSYST. *Proceeding 28 the ACM SIGCSE Tech. Symposium on Computer Science Education*, San Jose, California, USA, pp. 335-339, 1997.

[11] D. Jumaa, A Computer Model for Evaluation of Programs, Master Thesis, University of Engineering and Science, Iraq, 1992.

[12] A. Baroni, Formal Definition of Object-Oriented Design Metrics, Master Thesis, Universidade Nova de Lisboa, Portugal, 2002.

Appendix A

Different Designs and Their Evaluation Using The Developed System

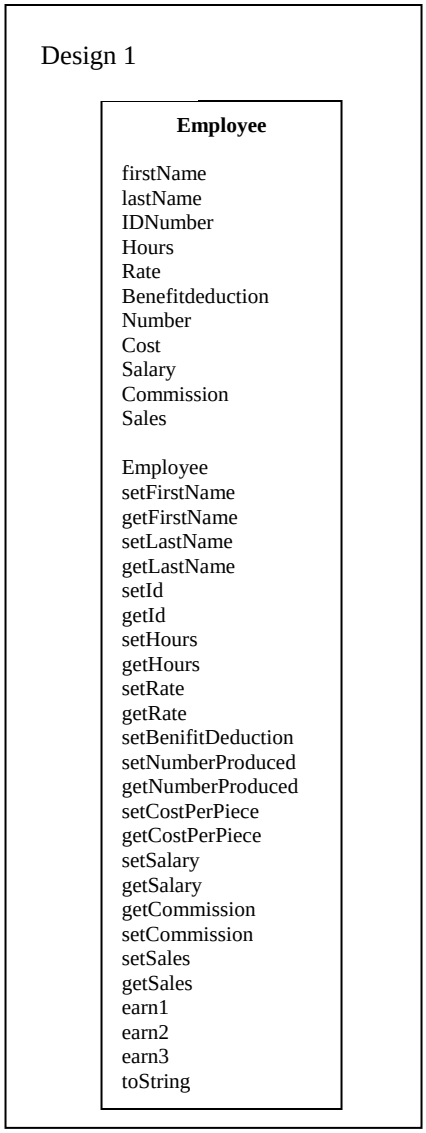


Figure 1: The design of program 1

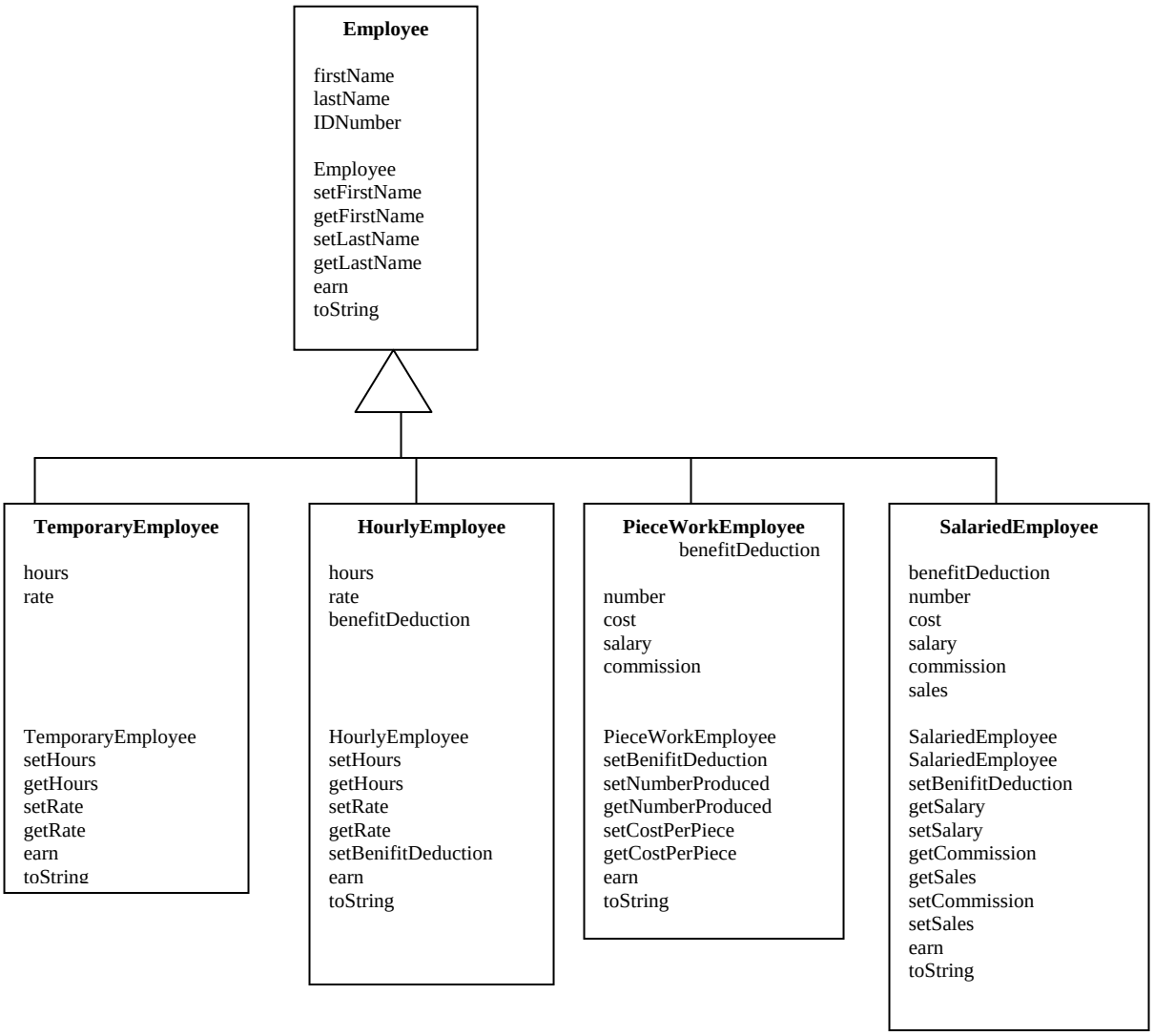
Evaluating Results

Factor	Model Program		Evaluated Program		Factor Weight
	Lower Bound	Upper Bound	Score	Grade (%)	
Method Hiding Factor (MIF)	12.7	21.8	0.0	0	1
Attribute Hiding Factor (AHF)	76.0	100.0	100	100	2
Method Inheritance Factor (MIF)	66.4	78.5	0.0	0	2
Attribute Inheritance Factor (AIF)	52.7	66.3	0.0	0	1
Coupling Factor (COF)	0.0	11.2	0.0	100	3
Polymorithism Factor (POF)	2.7	9.6	0.0	0	2
Final Grade is 45 %					

Comments:

- Hide more methods.
- Increase the total number of inherited methods.
- Increase the total number of inherited attributes.
- Increase the use of polymorithism

Design 2



Evaluating Results

Factor	Model Program		Evaluated Program		Factor Weight
	Lower Bound	Upper Bound	Score	Grade (%)	
Method Hiding Factor (MHF)	12.7	21.8	2.3	0	1
Attribute Hiding Factor (AHF)	76.0	100.0	100.0	100	2
Method Inheritance Factor (MIF)	66.4	78.5	45.57	10	2
Attribute Inheritance Factor (AIF)	52.7	66.3	44.44	80	1
Coupling Factor (COF)	4.0	11.2	0.0	100	3
Polymorithism Factor (POF)	2.7	9.6	22.22	30	2
Final Grade is 58 %					

Comments:

- Hide more methods.
- Increase the total number of inherited methods
- Decrease the use of polymorithism.

Design 3

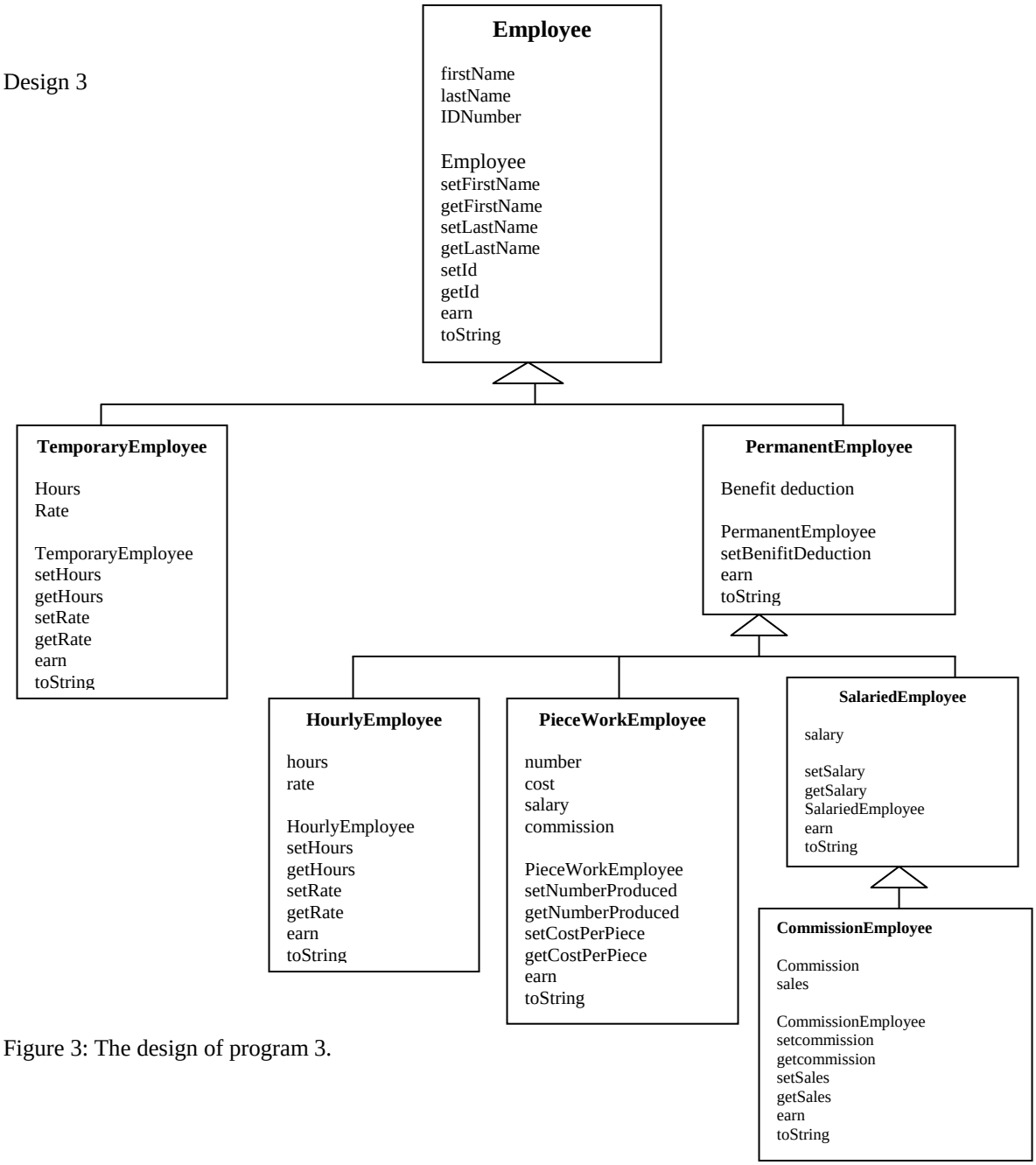


Figure 3: The design of program 3.

Evaluating Results

Factor	Model Program		Evaluated Program		Factor Weight
	Lower Bound	Upper Bound	Score	Grade (%)	
Method Hiding Factor (MHF)	12.7	21.8	4.4	10	1
Attribute Hiding Factor (AHF)	76.0	100.0	100.0	100	2
Method Inheritance Factor (MIF)	66.4	78.5	59.0	60	2
Attribute Inheritance Factor (AIF)	52.7	66.3	63.88	100	1
Coupling Factor (COF)	4.0	11.2	0	100	3
Polymorithism Factor (POF)	2.7	9.6	15.49	60	2
Final Grade is 77 %					

Comments:

- Hide more methods