Effectiveness of Traditional Software Metrics for Object-Oriented Systems

David P. Tegarden
Department of Management Information Systems, College of Business Administration
University of Denver, Denver, CO 80208 USA

Steven D. Sheetz
David E. Monarchi
Information Systems Area, College of Business and Administration
University of Colorado, Boulder, CO 80309-0419 USA

Abstract

An acceptable measure of software quality must quantify software complexity. Traditional software metrics such as lines of code, software science, cyclomatic complexity are investigated as possible indicators of complexity of object-oriented systems. This research reports the effects of polymorphism and inheritance on the complexity of object-oriented systems as measured by the traditional metrics. The results of this research indicate that traditional metrics are applicable to the measurement of the complexity of object-oriented systems.

1: Introduction

There is great interest in software metrics due to their potential for use in procedures to control costs of system development and maintenance activities. The ability to quantify the complexity of designs and software is a necessary condition for the creation of acceptable quality standards and refinement of estimating techniques. Traditional software metrics such as lines of code, cyclomatic complexity and those defined by software science are used in industry as indicators of quality. As object-oriented (OO) technologies emerge to support major applications, issues of quality assurance will become more prevalent throughout the OO system development process.

Increased awareness of quality issues will lead to questions associated with the effectiveness of applying traditional software metrics to OO systems. According to Moreau [11, 12, 13], traditional metrics are inappropriate for OO systems for several reasons. First, the assumptions relating program size and programmer productivity in structured systems do not apply directly to OO systems. Second, the traditional metrics do not address the structural aspects of OO systems. Third, the computation of the system's complexity as the sum of the complexity of the components is not apprpriate for OO systems. He also states [Moreau 12] that:

> ... the only place that existing, traditional, intraobject software metrics might be effective

is within a particular method within an object ...

However, to our knowledge no empirical evidence exists to support these statements. And Moreau himself [14] uses the traditional metrics to compare two implementations of a graphics editor, a traditional implementation in C and an object-oriented implementation in C++. Furthermore, we believe that there are valid reasons for evaluating the use of traditional metrics for OO systems. These reasons include:

- The traditional metrics exist.
- There is empirical evidence to support their use for structured systems.
- They are well understood by practitioners and researchers.

This paper addresses two questions about measuring OO systems using the traditional metrics :

1) Can existing metrics developed for structured systems be used as effective measures of OO systems?

2) Can certain unique aspects of OO systems be measured by traditional metrics?

We explore these questions through the use of an example application of the traditional metrics to different OO systems written in C++.

The next section of the paper reviews software complexity, the traditional metrics, and the salient differences between structured and OO systems. The third section describes the issues associated with the application of traditional metrics to OO systems. The fourth section is the application of the traditional metrics to the example OO system. The final section contains preliminary conclusions and future research directions based on the results obtained from the previous sections .

2: Background

Software complexity is an area of software engineering concerned with the measurement of factors that affect the cost of developing and maintaining software. Software complexity has been defined by Curtis [6] as

> ... a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software.

The term "system" refers not only to other software systems, but also to machines and people. Software complexity measures fall into two general categories: computational and psychological. Computational complexity refers to the efficiency of an algorithm and the use of machine resources. Psychological complexity refers to factors that affect the ability of a programmer to create, modify, and comprehend software and the end user to effectively use the software.

Card [3] and Conte, et al., [5] identify three specific types of psychological complexity

that affect programmer ability to comprehend software: problem complexity, system design complexity, and procedural complexity.

Problem complexity is a function of the problem domain. Simply stated, it is assumed that complex problem spaces are more difficult for a programmer to comprehend than simple problem spaces. This type of complexity is impossible to control, and thus, it is generally ignored in software engineering.

System design complexity addresses the complexity associated with mapping a problem space into a given representation. Card [3] identifies two different types of system design complexity: structural and data. Structural complexity addresses the concept of coupling. Coupling measures the interdependence of modules of source code, e.g., C functions calling other C functions. It is assumed that the higher the coupling between modules, the more difficult it is for a programmer to comprehend a given module. Data complexity addresses the concept of cohesion. Cohesion measures the intradependence of modules. In this case, it is assumed that the higher the measure of cohesiveness, the easier it is for a programmer to comprehend a given module [16].

Procedural complexity is associated with the complexity of the logical structure of a program. This approach to complexity measurement assumes that the length of the program [number of tokens or the lines of code] or the number of logical constructs [sequences, decisions, or loops] that a program contains determines the complexity of the program [3].

To keep this work at a reasonable scope, only the application of traditional metrics to the procedural complexity of OO systems is addressed.

2.1: Traditional metrics

Traditional metrics have been applied to the measurement of software complexity of structured systems since 1976 [9]. To answer the questions above and assess the effectiveness of the traditional metrics for OO systems, two additional questions are of interest. First, how are the traditional metrics calculated for structured systems? Second, how do the differences between OO systems and structured systems affect complexity as measured by the traditional metrics?

There is a variety of research published on the assessment of software complexity. Over 500 interdisciplinary references exist for this area [17]. The research consists primarily of software science [7]; extensions to software science, such as function points [1] and system design complexity [3]; and cyclomatic complexity [9]. In addition to these metrics, the count of source lines of code (SLOC) is often included as a raw indication of size in many studies. This work focuses on SLOC, software science, and cyclomatic complexity.

2.1.1 Source lines of code: Counting the SLOC is one of the earliest and easiest approaches to measuring complexity. It is also the most criticized approach. Problems with SLOC as a measure for structured systems still exist for OO systems. Additional questions are raised by the effects of the unique aspects of OO systems.

2.1.2 Software science: Software science is the only complete theory to attempt to explain the program development process [15]. Though the theory suffers from some significant criticisms and has not been adopted widely in industry, its evaluation and application comprise a large part of the above references. The theory defines the complexity of a program as a function of the number of operators and operands and the total number of occurrences of the operators and

operands in a program. A variety of metrics are then calculated from these counts. The following summarizes the counts and metrics from the theory that are used in this research [7]:

n_1 = Number of unique operators
n_2 = Number of unique operands
N_1 = Number of references to operators
N_2 = Number of references to operands
$n = n_1 + n_2$ (Vocabulary of the algorithm)
$N = N_1 + N_2$ (Length of the algorithm)
$V = N \log_2 n$ (Volume of the algorithm)

In general, the metrics for vocabulary, length, and volume have been supported empirically [15].

2.1.3 Cyclomatic complexity: Cyclomatic complexity is a measure of module control flow complexity based on graph theory [9]. This metric is popular with practitioners and researchers because it is simple to calculate.

Cyclomatic complexity of a module uses control structures to create a control flow matrix which in turn is used to generate a connected graph. The graph represents the control paths through the module. The complexity of the graph is the complexity of the module [9]. The cyclomatic complexity calculation as calculated from the graph for the module is:

$$v(G) = e - n + p$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components. Connected components are the nodes in the module that can be reached from outside the graph or that can transfer control outside the graph. This corresponds to the number of entry and exit points for the module. For a single entry/single exit module, p is two [9].

A technique for calculation of the metric from source code without using a graph for single entry/single exit modules is

$$v(G) = \textit{Number of Decisions} + 1.$$

The metric also accounts for nested hierarchical structure complexity through the summation of connected component complexity. Furthermore, a testing methodology associated with the unique paths through the module exists. Modules with cyclomatic complexity less than 10 have been shown to contain significantly fewer errors than those with greater values. [10]

2.2: Structured systems versus object-oriented systems

OO systems differ from structured systems in that OO systems encapsulate structure [data] and behavior [operations]. OO systems integrate both the structural and behavioral aspects of a program, while structured systems force an artificial separation of the structure from behavior. According to Brodie and Ridjanovic [2],

> The separate treatment of structure and behavior complicates design, specification, modification, and semantic integrity analysis.

Encapsulation, inherent in OO approaches, allows programmers to focus on the current object description and to avoid creating unwanted side effects in other objects.

Polymorphism and inheritance are two aspects unique to OO systems that reduce software complexity as measured by the traditional metrics. Polymorphism and inheritance are described below.

2.2.1 Polymorphism: Polymorphism means having the ability to take several forms. For OO systems, polymorphism allows the implementation of a given operation to be dependent on the object that contains the operation. For example, there can be different operations to pay employees based on the employee's object type, e.g., part-time, hourly, or salaried. Each type of employee object can have its own customized compute-pay operation. When a new type of employee is created, e.g., student, the programmer simply creates a new type of employee object and a new implementation of compute-pay for the new type of employee. When an instance of student receives the message to compute-pay, it uses the operation defined in the new object to perform the calculation. The compute-pay operations of the other types of employee are not affected by the payment operations required for the student. In contrast, structured systems often have all pay operations contained in one program. The program must be capable of differentiating between the different types of employees and applying the appropriate operation. Modifications to add a new type of employee typically require existing structured code to be changed.

We expect that the complexity of OO software will be reduced through use of polymorphism for two reasons. First, the programmer does not have to comprehend, or even be aware of, existing operations to add a new operation. For example, the programmer does not have to consider how the compute-pay operation is defined for part-time, hourly, or salaried employees to create a compute-pay operation for student employees. Second, the programmer does not have to consider methods that are not part of the object when naming the operation. This allows the programmer to preserve the semantics of the operation within the object and to provide common interfaces to types of objects that are similar, e.g., types of employees.

2.2.2 Inheritance: There is one prototypical characteristic of OO systems; the existence of an inheritance mechanism. (For a discussion of some types of inheritance mechanisms associated with OO systems see [8]). Inheritance is a reuse mechanism that allows programmers to define objects incrementally by reusing previously defined objects as the basis for new objects. For example, when defining a new type of employee, e.g., student, the new employee type can inherit the characteristics common to all employees, e.g., name, address, etc., from a generic type of employee. In this approach, the programmer needs only to be concerned with the difference between student employees and generic employees. Structured systems do not have an inheritance mechanism as part of their formal specification.

We expect that inheritance will decrease the software complexity of OO systems. This is primarily due to inheritance enabling the programmer to reuse previously defined objects, including their variables and operations, as part of the definition of new objects. This will reduce the number of operations and operands required.

3: Issues with traditional metrics and object-oriented systems

The main issues associated with traditional metrics and OO systems are determining how to

compute the metrics and how the unique aspects of OO systems are measured. The questions associated with each of the metrics are detailed below.

3.1: Source lines of code

There are four problems associated with the use of SLOC as a measure of complexity in structured systems. First, there is no agreement on what is a line of code. Is a single multi-line statement a single line of code? What about a single line that contains multiple statements? Is a comment or variable declaration line counted as a line of code? Second, SLOC does not count the different levels of complexity of different lines. It assumes that the statement X = 2 is just as complex as the statement X = N! / (n! * (N - n)!). Third, SLOC assumes that two programs with the same number of lines of code are equivalent in complexity. Fourth, the measure is available only after the code has been written. These problems also exist for the use of SLOC to measure the complexity of OO systems.

The effects of inheritance and polymorphism raise additional questions about the utility of SLOC to measure complexity in an OO system. For example, should the SLOC of an inherited instance method be counted as part of every object in the hierarchy that inherits it, or just the objects that use the method? Or should they be counted only once at the level of definition? Additionally, should the SLOC of class and instance methods be counted differently?

When polymorphism is used, the question is whether SLOC for the method is the sum of the SLOC in each definition of the method together or whether each definition should be viewed as a new method that should be measured independently of other definitions?

On a larger scale, is the SLOC for an object the sum of the SLOC of the methods defined in the object or does it also include the variable declarations unique to the object? What part of the large class libraries available for many OO programming environments should be counted as part of an application? Should SLOC of parts of the class library used through messages be counted or just those that are redefined or inherited? Are these classes "utilities" that should not be counted at all?

3.2: Software science

Software science (SS) assumes that two separate programs with the same number of unique operators and operands and the same total number of occurrences of the operators and operands are equivalent in terms of software complexity.

The main questions for OO systems associated with SS metrics concern the rules used to determine what is a countable operator or operand, and what reference is appropriate for each use of a method. We assume that methods correspond to operators and that variables correspond to operands in OO systems. We also assume that the object portion of messages serves only to define the method to be referenced. Therefore, an object is neither an operator or an operand.

Problems similar to those associated with counting SLOC exist. Should an inherited class/instance method count in each object that inherits it, the objects that use it, or only in the object that defines it? What about an inherited variable? Is each polymorphic definition of a method an independent operator or an increment to the count of that operator at its highest level in the object hierarchy? Message references to specific methods may be available only at run time based on the data contained in variables that occupy the object portion of the message. If so, what does this mean for the applicability of these static measures to OO systems?

3.3: Cyclomatic complexity

Calculation of cyclomatic complexity for each method in an object system corresponds to the use of the metric on each module of a structured system. The sum of the complexity measures of subordinate modules in a structured design hierarchy covers all potential contributions to complexity. This is not the case for an object system hierarchy. Cyclomatic complexity cannot be calculated by summing the cyclomatic complexities of the methods in subordinate objects because not all contributions to the complexity of a method can be derived from subordinates. Some methods that are used by the method are inherited from superordinate classes; some methods are used from other parts of the object hierarchy; and some methods are used from the object. To sum the constituent values into a system level metric, consistent with the calculation for structured systems, requires a way to generate a hierarchy similar to a structured design hierarchy (possibly of the system's dynamic activity). This is an area of future research.

Methods will have only one entry point but may have multiple exits. This will increase values for cyclomatic complexity versus similar methods using single exit conventions. This will result in similar OO systems potentially having different levels of complexity.

Each method defined in the system is unique for the purpose of calculating cyclomatic complexity. The code may be performing the same semantic function in each object, but it is different code.

The use of cyclomatic complexity to control individual methods is appropriate to the static evaluation of object software. Low cyclomatic complexity of methods may indicate the code of an OO system is not complex from the view of this metric. Alternatively, it may show that some of the decisions normally measured in a structured module are deferred through message passing to other objects. If this is true, we expect to find that OO systems contain few methods with high values for cyclomatic complexity.

4: Object-oriented system example

To exhibit the use of the different metrics, a sample system is examined. The OO system example is a hypothetical General Ledger Account system. The accounts are stored in a linked list. The contents of the linked list can be exhibited as the chart of accounts.

To demonstrate how the metrics can be applied to this OO system, we calculate SLOC, software science counts and metrics, and cyclomatic complexity for each object and the complete system. These calculations are performed on four versions of the example system. To create these versions, polymorphism and inheritance are manipulated in the example. The versions resulting from the manipulations are:

- Inclusion of neither polymorphism or inheritance.
- Inclusion of only polymorphism.
- Inclusion of only inheritance.
- Inclusion of both polymorphism and inheritance.

An object hierarchy without inheritance, an object hierarchy with inheritance, and a list of the

objects, methods and variables for each version of the example are shown in figures 1 through 6. PC-METRIC from SET Laboratories was used to calculate the metrics. The results of these calculations are contained in tables 1 and 2. The remainder of this section describes the four versions.

Preservation of the encapsulation principle is the controlling factor of the approach used for creating the versions. When inheritance is not used, operands and operations are duplicated in each instantiable object. This is consistent with the approach recommended by Coad and Yourdon for implementing an OO design in a language that does not support inheritance [4]. Figures 1 and 2 show the difference in the object hierarchy of the OO system when inheritance was not used and when it was. When polymorphism was not used, unique operator names were defined for all behaviors. When polymorphism is used, operator names were overloaded for semantically equivalent operations.

4.1: Neither polymorphism or inheritance

Figure 1 contains the object hierarchy for this version of the example. Figure 3 contains a list of the classes and their associated variables and methods for this version. The results at the system level without polymorphism and without inheritance can be found in table 1, column 1. This column contains the highest values in the table for all metrics; it is the most complex version of the example system. This indicates that ignoring the use of polymorphism and inheritance in OO systems leads to more complex systems.

Tables 2a through 2d, column 1, show the results of not using polymorphism or inheritance at the individual object level. For example, in table 2c, column 1 demonstrates that not using these two aspects of OO systems substantially increases the complexity of the AccountList object as viewed by the traditional metrics. This column contains the highest values in the table for all metrics; it is the most complex version of the AccountList object.

4.2: With polymorphism and without inheritance

Figure 1 contains the object hierarchy for this version of the code. Figure 4 contains a list of the classes and their associated variables and methods for this version. The differences in the code are that all the inheritable methods were incorporated into the instanstiable objects. This results in significant code redundancies, but it preserves the encapsulation principle. An alternative approach is to create shared functions, but we believe this would violate the encapsulation principle. A comparison of column 1 and column 2 from table 1shows the use of polymorphism decreases the complexity at the system level as compared to the first version.

Tables 2a through 2d, column 2, show the results of using polymorphism but not inheritance at the individual object level. For example, in table 2c, column 2 the use of polymorphism significantly decreases the complexity of the AccountList object as compared to column 1.

4.3: Without polymorphism and with inheritance

Figure 2 contains the object hierarchy for this version of the example. Figure 5 contains a list of the classes and their associated variables and methods for this version. The difference in the code is that the common methods are abstracted into the super objects, e.g., the GetKey method

in the Account object was abstracted from the Asset, Expense, Liability, OwnersEquity and Revenue objects. This results in significant code reductions. Table 1, column 3, shows the use of inheritance decreases the complexity of the system, e.g., it decreased the cyclomatic complexity (V(G) Sum) from 95 (column 1), when neither polymorphism or inheritance is used, to 57 when only inheritance was used (column 3).

Tables 2a through 2d, column 3, show the results of using inheritance and polymorphism at the individual object level. For example, in table 2c, column 3, the use of inheritance decreases the complexity of the AccountList object more than that for polymorphism shown in column 2. This is a greater reduction from the version of the system that uses neither polymorphism or inheritance shown in column 1.

4.4: With polymorphism and with inheritance

Figure 2 contains the object hierarchy for this version of the example. Figure 6 contains a list of the classes and their associated variables and methods for this version. As in the previous version, the difference in the code is that the common methods are abstracted into the super objects. This results in significant code reductions. In this version, polymorphism was included which reduces the number of operator names. As table 1, column 4 shows, the combination of using both inheritance and polymorphism produced the minimum complexity of the system, e.g., it decreased the Software Science Volume (V) metric from 9,934, when neither are used, to 5,184 when both are used.

Tables 2a through 2d, column 4, show the results of using inheritance and polymorphism at the individual object level. For example, in table 2c, column 4, the use of polymorphism and inheritance decreased the complexity of the AccountList object more than that for using either individually.

5: Conclusion

The ability to quantify the complexity of software is necessary for the measurement of software quality. This research has addressed the effectiveness of applying the traditional software metrics to the measurement of OO systems.

This research indicates that the use of the traditional metrics may be appropriate for the measurement of the complexity of OO systems. Even though the order of magnitude of the traditional metrics may be suspect, the directionality seems to be correct. The use of inheritance and/or polymorphism should decrease the complexity of an OO system. This is captured by the traditional metrics. However, we believe additional metrics are required to fully measure all aspects of OO systems. These metrics should include the complexity of messages being passed and the differentiation of generalization and aggregation structures.

Future research directions include the definition of additional software metrics that address the measurement of other unique aspects of OO systems not measured by the traditional metrics. For example, the effects of inheritance conflicts and the resolution of these conflicts on OO system complexity will be studied.

Acknowledgement

We would like to thank the anonymous referees and Kenneth A. Kozar, of the University of Colorado - Boulder, for their helpful comments on earlier drafts of this paper.

References

1. A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 639-647, Nov. 1983.
2. M. L. Brodie and D. Ridjanovic, "On the design and specification of database transactions, " *On conceptual Modelling: Perspectives from Artificial Intelligence, databases, and Programming Languages*, Springer Verlag, New York, 1984.
3. D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Englewood Cliffs, NJ:Prentice Hall, 1990.
4. P. Coad and E. Yourdon, *Object-Oriented Design*, Englewood Cliffs, NJ:Prentice Hall, 1991.
5. S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Menlo Park, CA: Benjamin/Cummings, 1986.
6. Bill Curtis, "The Measurement of Software Quality and Complexity," *Software Metrics: An Analysis and Evaluation*, Cambridge,MA:The MIT Press, 1981.
7. M. H. Halstead, *Elements of Software Science*, New York: Elsevier, North-Holland, 1977.
8. M. Lenzerini, D. Nardi, and M. Simi, *Inheritance Hierarchies in Knowledge Representationand Programming Languages*, West Sussex, England: John Wiley and Sons, 1991.
9. T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308-320, Dec. 1976.
10. T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Commun. ACM*, vol. 32, pp. 1415-1425, Dec. 1989.
11. D. R. Moreau, *A Programming Environment Evaluation Methodology for Object-Oriented Systems*, Ph.D. Dissertation, University of Southwestern Louisiana, Sep. 1987 .
12. D. R. Moreau and W. D. Dominick, "Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics," *Journal of Systems and Software*, vol. 10, pp. 23-28, 1989.
13. D. R. Moreau and W. D. Dominick, "A programming environment evaluation methodology for object-oriented systems: part I - the methodology," *Journal of Object-Oriented Programming*, vol. 3, pp. 38-52, May/Jun. 1990.
14. D. R. Moreau and W. D. Dominick, "A prgramming environment evaluation methodology for object-oriented systems: part II - test case application," *Journal of Object-Oriented Programming*, vol. 3, pp. 23-32, Sep./Oct. 1990.
15. V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: a critical analysis of the theory and its empirical support," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 155-165, Mar. 1983.
16. W. P. Stevens, G. J. Meyers, and L. L. Constantine, "Structured Design", *Tutorial: Software Design Strategies*, 2nd Ed., New York, NY:IEEE Computer Society, 1981.
17. L. J. Waguespack Jr. and S. Badlani, "Software complexity assessment: an introduction and annotated bibliography," *ACM SIGSOFT Software Engineering Notes*, vol. 12, pp. 52-71, Oct. 1987.

Figure 1: Account System Object Hierarchy using No Polymorphism and No Inheritance.

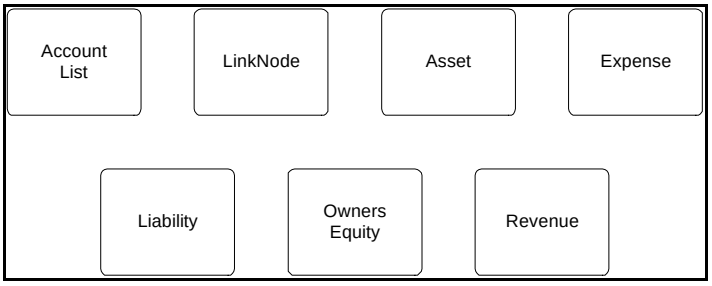

Figure 2:Account System Object Hierarchyusing Polymorphism and Inheritance.

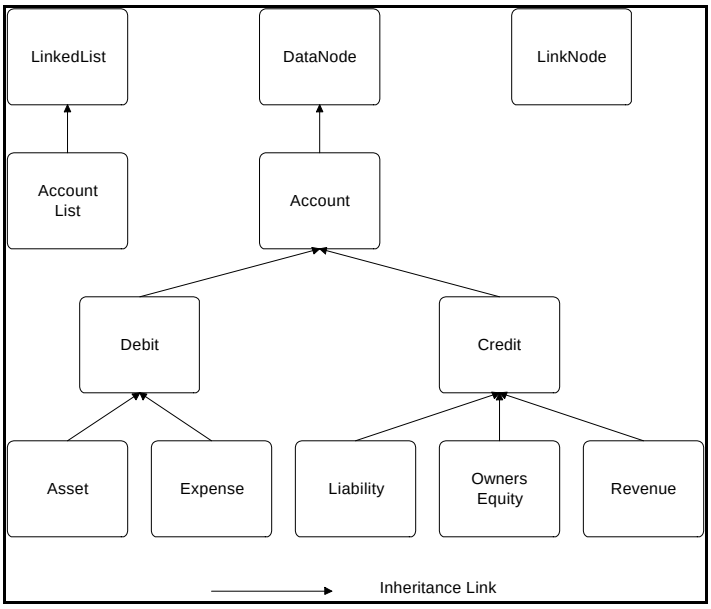

Figure 3: Account System Object Composition Without Polymorphism and Without Inheritance.

Object: AccountList
Variables: BegOfListPtr
EndOfListPtr
Methods: Find
Constructor -- LinkedList
Destructor -- ~LinkedList
Insert
Delete
Update
Print

Object: LinkNode
Variables: DataPart
PrevLink
NextLink
Methods: GetLinkKey
GetLinkNode
Constructor -- LinkNode
Destructor -- ~LinkNode

Object: Asset
Variables: Number
Name
Balance
Methods: GetAssetKey
GetAssetAccount
GetAssetBalance
UpdateAsset
PrintAsset
Constructor -- Asset

Object: Liability
Variables: Number
Name
Balance
Methods: GetLiabilityKey
GetLiabilityAccount
GetLiabilityBalance
UpdateLiability
PrintLiability
Constructor -- Liability

Object: OwnersEquity
Variables: Number
Name
Balance
Methods: GetOwnersEquityKey
GetOwnersEquityAccount
GetOwnersEquityBalance
UpdateOwnersEquity
PrintOwnersEquity
Constructor -- OwnersEquity

Object: Revenue
Variables: Number
Name
Balance
Methods: GetRevenueKey
GetRevenueAccount
GetRevenueBalance
UpdateRevenue
PrintRevenue
Constructor -- Revenue

Object: Expense
Variables: Number
Name
Balance
Methods: GetExpenseKey
GetExpenseAccount
GetExpenseBalance
UpdateExpense
PrintExpense
Constructor -- Expense

Figure 4: Account System Object Composition With Polymorphism and Without Inheritance.

Object: AccountList
Variables: BegOfListPtr
EndOfListPtr
Methods: Find
Constructor -- LinkedList
Destructor -- ~LinkedList
Insert
Delete
Update
Print

Object: LinkNode
Variables: DataPart
PrevLink
NextLink
Methods: GetKey
GetLinkNode
Constructor -- LinkNode
Destructor -- ~LinkNode

Object: Asset
Variables: Number
Name
Balance
Methods: GetKey
GetDataAccount
GetBalance
Update
Print
Constructor -- Asset

Object: Liability
Variables: Number
Name
Balance
Methods: GetKey
GetDataData
GetBalance
Update
Print
Constructor -- Liability

Object: OwnersEquity
Variables: Number
Name
Balance
Methods: GetKey
GetDataNode
GetBalance
Update
Print
Constructor -- OwnersEquity

Object: Revenue
Variables: Number
Name
Balance
Methods: GetKey
GetDataNode
GetBalance
Update
Print
Constructor -- Expense

Object: Expense
Variables: Number
Name
Balance
Methods: GetKey
GetDataNode
GetBalance
Update
Print
Constructor -- Expense

Figure 5: Account System Object Composition Without Polymorphism and With Inheritance.

Object: LinkedList
Variables: BegOfListPtr
EndOfListPtr
Methods: Find
Constructor -- LinkedList
Destructor -- ~LinkedList
Insert
Delete

Object: LinkNode
Variables: DataPart
PrevLink
NextLink
Methods: GetLinkKey
GetLinkNode
Constructor -- LinkNode
Destructor -- ~LinkNode

Object: DataNode
Variables:
Methods: GetDataNode

Object: AccountList
Variables:
Methods: Print
Insert
Constructor -- AccountList
Update

Object: Account
Variables: Number
Name
Balance
Mehtods: GetAccountKey
GetAccountBalance
UpdateAccount
PrintAccount
Constructor -- Account

Object: Debit
Variables:
Methods: Constructor -- Debit
PrintDebit

Object: Credit
Variables:
Methods: GetCreditBalance
UpdateCredit
Constructor -- Credit

Object: Asset
Variables:
Methods: Constructor -- Asset

Object: Liability
Variables:
Methods: Constructor -- Liability

Object: OwnersEquity
Variables:
Methods: Constructor -- OwnersEquity

Object: Revenue
Variables:
Methods: Constructor -- Revenue

Object: Expense
Variables:
Methods: Constructor -- Expense

Figure 6: Account System Object Composition With Polymorphism and With Inheritance.

Object: LinkedList
Variables: BegOfListPtr
EndOfListPtr
Methods: Find
Constructor -- LinkedList
Destructor -- ~LinkedList
Insert
Delete

Object: LinkNode
Variables: DataPart
PrevLink
NextLink
Methods: GetKey
GetLinkNode
Constructor -- LinkNode
Destructor -- ~LinkNode

Object: DataNode
Variables:
Methods: GetDataNode
GetKey

Object: AccountList
Variables:
Methods: Print
Insert
Constructor -- AccountList Update

Object: Account
Variables: Number
Name
Balance
Mehtods: GetAccountKey
GetAccountBalance
UpdateAccount
PrintAccount
Constructor -- Account

Object: Debit
Variables:
Methods: Constructor -- Debit
PrintDebit

Object: Credit
Variables:
Methods: GetBalance
Update
Constructor -- Credit
Print

Object: Asset
Variables:
Methods: Constructor -- Asset

Object: Liability
Variables:
Methods: Constructor -- Liability

Object: OwnersEquity
Variables:
Methods: Constructor -- OwnersEquity

Object: Revenue
Variables:
Methods: Constructor -- Revenue

Object: Expense
Variables:
Methods: Constructor -- Expense

Table 1: Results of PC-Metrics for Four Versions at the System Level.

Count or Metric	No Poly No Inher	Poly No Inher	No Poly Inher	Poly & Inher
# of Classes	7	7	12	12
Members/Class	7	7	2	2
# Methods	42	42	30	30
n_1	75	53	58	53
n_2	58	56	56	56
N_1	955	735	541	521
N_2	453	359	254	245
N	1,408	1,094	795	766
V	9,934	7,404	5,432	5,184
V(G) Sum	95	71	57	55
File V(G)	54	30	28	26
Total SLOC	676	626	539	534
Exec. SLOC	202	181	126	124

Table 2: Results of PC-Metrics at the Object Level.

a. Asset and Expense.

Count or Metric	No Poly No Inher	Poly No Inher	No Poly Inher	Poly & Inher
# Attributes	3	3	0	0
# Methods	6	6	1	1
n_1	18	18	1	1
n_2	16	16	1	1
N_1	60	60	1	1
N_2	29	29	1	1
N	89	89	2	2
V	453	453	1	1
V(G) Sum	7	7	1	1
File V(G)	2	2	1	1
Total SLOC	68	68	8	8
Exec. SLOC	16	16	0	0

b. Liability, OwnersEquity, and Revenue.

Count or Metric	No Poly No Inher	Poly No Inher	No Poly Inher	Poly & Inher
# Attributes	3	3	0	0
# Methods	6	6	1	1
n_1	19	19	1	1
n_2	16	16	1	1
N_1	62	62	1	1
N_2	30	30	1	1
N	92	92	2	2
V	472	472	1	1
V(G) Sum	7	7	1	1
File V(G	2	2	1	1
Total SLOC	68	68	8	8
Exec. SLOC	16	16	0	0

c. AccountList Object values.

Count or Metric	No Poly No Inher	Poly No Inher	No Poly Inher	Poly & Inher
# Attributes	2	2	0	0
# Methods	7	7	4	4
n_1	58	41	41	38
n_2	42	41	34	34
N_1	538	347	263	241
N_2	275	193	139	130
N	813	540	402	371
V	5,401	3,433	2,504	2,289
V(G) Sum	44	25	19	17
File V(G)	38	19	16	14
Total SLOC	232	193	132	126
Exec. SLOC	100	84	58	56

d. LinkNode Object Values.

Count or Metric	No Poly No Inher	Poly No Inher	No Poly Inher	Poly & Inher
# Attributes	3	3	3	3
# Methods	4	4	4	4
n_1	20	9	9	9
n_2	5	4	4	4
N_1	65	20	20	20
N	19	8	8	8
N	84	28	28	28
V	390	104	104	104
V(G) Sum	9	4	4	4
File V(G)	6	1	1	1
Total SLOC	56	45	45	45
Exec. SLOC	11	6	6	6