Design and Code Complexity Metrics for OO Classes

Letha Etzkorn, Jagdish Bansiya, and Carl Davis

The University of Alabama in Huntsville

{letzkorn, jbansiya, cdavis} @cs.uah.edu

Software complexity metrics for the procedural development paradigm have been extensively studied. Metrics such as McCabe's cyclomatic complexity metric[1] and Halstead's Software Science metrics[2] are well known and frequently used to measure software complexity in the procedural paradigm. All the traditional, procedural complexity metrics measured the complexity of the code in a particular software product.

More recently, software metrics that are tailored to the measurement of complexity in the object-oriented paradigm have been developed. The use of the object-oriented paradigm has allowed for the development of class complexity metrics that will operate at design time, and that can predict the complexity that will be present in a fully implemented class. Thus, for OO software, there are two categories of complexity metrics:

- code complexity metrics
- design complexity metrics

The primary advantage of design complexity metrics over code complexity metrics is that they can be used at design time, prior to code implementation. This permits the quality of designs to be analyzed as the designs are developed, which allows improvement of the design prior to implementation.

In this article, we will examine the implementation and usage of various OO

code and design complexity metrics. We will introduce a new OO code metric, Average Method Complexity (AMC), that measures certain aspects of code complexity not handled by current code complexity metrics. We will also discuss a new OO design metric, called Class Design Entropy (CDE), that uses the information content of a class as a measure of complexity. Over a set of C++ classes, we will compare the complexity determinations of several OO code and design metrics to the complexity ratings of a team of highly trained C++ experts. Finally, we will discuss some of the measurement differences between OO design metrics and OO complexity metrics.

OO CODE COMPLEXITY METRICS

One OO complexity metric that has been widely examined is the Weighted Methods per Class (WMC) metric[3,4]. The definition of this metric is as follows[4]:

Consider a Class C_1, with methods $M_1,...M_n$ that are defined in the class. Let $c_1...c_n$ be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^{n} c_i$$

If all method complexities are considered to be unity, then WMC = n, the number of methods.

In the case when WMC is equal to the number of methods (since all the method complexities are unity), then WMC can be considered to be a design complexity metric, since the number of methods of a class is determined from the class header, and is thus available at design time. However, if the method complexities are not considered to be

unity, and are measured using a standard procedural complexity metric (such as McCabe's cyclomatic complexity), then WMC is an OO code complexity metric. A closer specification of the WMC code complexity metric occurs in Li and Henry[5,6]:

The WMC is calculated as the sum of McCabe's cyclomatic complexity of each local method: WMC = summation of McCabe's cyclomatic complexity of all local methods, ranging from 0 to N, where N is a positive integer.

WMC Implementation

There are some variations on the WMC metric that are independent of which definition is used. Some of these variations include the determination of exactly which member functions take part in the calculations.

Churcher and Shepperd discussed some of these implementation variations for the Chidamber and Kemerer WMC definition[7]. They discussed at length the different ways in which the number of methods of a class, which is integral to the calculation of the WMC metric, can be counted. To discuss some of the issues raised by Churcher and Shepperd, the C++ code in Figure 1 is provided. This code was drawn from a graphical user interface system[8].

One of the questions raised by Churcher and Shepperd is whether or not methods from inherited classes should be included in the count of methods for the current class. In Figure 1, the class GnCommand is derived from the class GnObject. In this case the question is whether methods in GnObject should be included in a count of methods for GnCommand. If so, then

```
class GnCommand : public GnObject {
   friend class GnHistoryTool;
   friend class GnView;
   friend class GnApplication;
   friend class GnDocument;
  protected:
   GnDocument *document;
   GnView     *view;
   int        view_x_offset;
   int        view_y_offset;
  public:
   GnCommand(GnDocument *, GnView * = 0, int create_checkpoint = 0 );
  public:
   virtual char *name() = 0;
   virtual void submit();
  protected: // IO (optional)
   virtual int IsStorable();
   virtual int WriteToStream(ostream &p_ostr);
   virtual int ReadFromStream(istream &p_istr);
  protected:
   virtual int executable()    { return(1); };
   virtual int undoable()      { return(1); };
   virtual int clock_cursor()  { return(0); };
   virtual int causes_change() { return(1); };

   virtual void doit ();
   virtual void redoit ();
   virtual void undoit ();
   virtual void commit ( int undone = 1 );

   virtual void scroll_before_redo();
   virtual void scroll_before_undo ();

   void submit_to_framework ();
   void execute_undoit();
   void execute_redoit();

   void undoit_before();
   void redoit_before();
  private:
   META_DEF_1(GnCommand,GnObject);
};
```

Figure 1. An Illustration of WMC Implementation Issues

should any private methods in GnObject (which are not directly accessible by GnCommand) be included in the count for GnCommand?

Another question raised by Churcher and Shepperd is whether methods should be counted before or after a pre-processor has been applied to the code. In Figure 1,

the macro META_DEF_1 will result in the definition of several more methods; however, this method definition will not occur until after preprocessing.

There are several friend classes defined in the class shown in Figure 1. Churcher and Shepperd mention briefly that a C++ friend class may subvert the normal C++ access rules. Therefore should methods from a friend class be counted in the method count for the current class?

Other issues that Churcher and Shepperd were concerned with was whether overloaded methods that employ the same name (such as multiple constructors) should be counted as multiple methods or as a single method, and whether or not operators should be counted as a method.

Chidamber and Kemerer's response[9] to Churcher and Shepperd indicated that their principle was that methods which required additional design effort and are defined in the class should be counted, inherited methods and methods from friend classes are not defined in the class, and so should not be included. Also, all distinct methods and operators in the class should be counted even when they share an operator. Methods should be counted prior to preprocessing.

Basili et. al.[10] mention that they believe the different counting rules proposed by Churcher and Shepperd correspond to different metrics, all of which are similar to the WMC metric but not the same. They say that each counting rule should be separately validated.

Given these decisions, we would like to pose an additional implementation question. First, if methods are to be

```
class wxWindow: public wxbWindow
{
 public:
  // When doing globale cursor changes, the current cursor may need to be saved for each window
  Cursor currentWindowCursor;
#ifdef wx_xview
  Xv_opaque dropSite;
#endif
  // Constructors/Destructors
  wxWindow(void);
  ~wxWindow(void);
  void GetSize(int *width, int *height);
  void GetPosition(int *x, int *y);
  void GetClientSize(int *width, int *height); // Size client can use
            . . .
  // Sometimes in Motif there are problems popping up a menu
  // (for unknown reasons); use this instead when this happens.
#ifdef wx_motif
  Bool FakePopupMenu(wxMenu *menu, float x, float y);
#endif

  void Show(Bool show);
  wxCursor *SetCursor(wxCursor *cursor);
  void SetColourMap(wxColourMap *cmap);

  float GetCharHeight(void);
  float GetCharWidth(void);
  void GetTextExtent(const char *string, float *x, float *y, float *descent = NULL, float *externalLeading = NULL);

  int CharCodeXToWX(KeySym keySym);
  KeySym CharCodeWXToX(int id);

#ifdef wx_motif
  virtual Bool PreResize(void);
  virtual void PostDestroyChildren(void);
  int wxType;
#endif
  // Get the underlying X window
  virtual Window GetXWindow(void);
  virtual Display *GetXDisplay(void);
};
#endif // IN_CPROTO
#endif
```

Figure 2. An Example of Conditional Compilation in C++

counted prior to pre-processing (that is, based on the source code itself), how should conditional compilation (such that certain methods are defined in one path of the conditional compilation, whereas others are defined in another conditional compilation path) be handled? This is a commonly-occurring phenomenon. The

code in Figure 2, which is drawn from illustrates such an example[11]. In this example, note that if the compilation flag wx_motif is defined instead of the flag wx_xview, then three additional member functions are defined. However, if these functions are counted prior to preprocessing, then the wx_motif-specific methods would be counted, even when the wx_xview compilation is intended. Our feeling is that the WMC metric normally should be calculated for a particular compilation. Thus at least some preprocessing of conditional compilation operators is required prior to the calculation of the WMC metric.

Average Method Complexity

During the course of a study that compared the complexity values of OO complexity metrics to the complexity determinations of a team of highly trained C++ experts (this study will be discussed another graphical user interface package, in detail later in this article), an anomaly was noted when comparing the experts' complexity rating of certain classes to that of the code complexity WMC metric (the Li and Henry version of the WMC metric). The experts rated certain classes as not complex, even though the classes had a very large number of member functions, which is contradictory to the way any definition of the WMC metric works. In these cases, the member functions in general were very simple member functions, but due to the large number of member functions the WMC value, which is additive, tended to be a fairly large number. Our observation was that in such cases the mental complexity measure employed by the experts was not additive. Thus an average measure would better reflect the complexity of that type of class. For this reason we

define a new metric, Average Method Complexity (AMC):

Let $c_1, \ldots c_n$ be the static complexity of the methods. Then

$$AMC = (1/n) \sum_{i=1}^{n} c_i$$

In this metric, the complexity of each method should never be unity; rather, it should be measured using a static complexity metric such as the McCabe's cyclomatic complexity metric.

The use of the AMC metric gives a better indication of the complexity of a class with a large number of uncomplex member functions than does the code complexity metric WMC (the Li and Henry WMC), which adds the McCabe's cyclomatic complexity numbers of all member functions in the class.

OO DESIGN COMPLEXITY MEASURES

The definition of the WMC metric, with the complexity of each method set to unity such that WMC is the number of methods in a class is a design complexity metric, since it can be calculated based on the class definition alone. The purpose of the design complexity metric for a class is to predict the complexity of the implemented class. However, the WMC metric is simply a count of the number of member functions in a class, and in some cases is not a particularly good predictor of the eventual complexity of an implemented class. For example, consider a class A that has a small number (n) of very simple member functions, and another class B that has a large number(n) of very complex member functions. Since the number of member functions is the same, at design time the WMC metric would predict the same complexity for each of the two classes,

whereas obviously class B is considerably more complex than class A.

These weaknesses of the WMC metric as a design complexity metric have been addressed by the Class Design Entropy (CDE) metric, which is a measure of the information content of a class[12].

Entropy, defined in terms of the information content of software, has been used to measure code complexity in the procedural paradigm for C programs[13], and for FORTRAN and COBOL programs[14]. Torress and Samadzadeh showed that entropy provides a measure of program control complexity, and showed an inverse relationship between the information content of software subsystems and their reusability[14]. All of this work was based on the work by Zweben and Halstead, which showed that operator usage in a program could be used to measure a program's information content[15,16]. Operators were defined to be either a special symbol, reserved word, or function call.

However, in order to determine the information content of OO designs, this operator-based approach is not appealing, since the goal is to assess a design prior to implementation. Thus, in the CDE design metric, the term "operators" is redefined as "special symbols which refer to simple strings, or user-defined strings such as object, operation, attribute, or association names." Thus a string, a sequence of alphabetic characters, is defined to be a special symbol, and is the basic unit of information for the CDE design metric. The amount of information conveyed by a string S_i, is inversely related to its probability (P_i) of occurring. The amount of information, I_i, in abstract units called "bits", conveyed by a single string S_i, with probability of occurrence P_i, is[12]:

$$\mathbf{I_i = - \log_2 P_i}$$

Information is additive—the information conveyed by two strings is the sum of their individual information content.

The probability, P_i, of the i^{th} most frequently occurring string is equal to the percentage of total string occurrences it contributes:

$$\mathbf{P_i = f_i / N_1}$$

where N_1 is the total number of non-unique name string operators used in the class definition, and f_i is the number of occurrences of the i^{th} most frequently occurring operator. Therefore, the average amount of information contributed by each symbol operator in a class design, called the empirical object-class design entropy, is:

$$\mathbf{H = - \sum_{i=1}^{n_1} P_i \log_2 P_i}$$

where n_1 is the total number of unique symbol operators

The Class Design Entropy (CDE) design complexity metric is defined as:

$$\mathbf{CDE = - \sum_{i=1}^{n_1} (f_i / N_1) \log_2 (f_i / N_1)}$$

where:

n_1 is the number of unique special string names,

N_1 is the total number of non-unique string names, and

f_i, $1 \leq i \leq n$, is the frequency of occurrence of the i^{th} special string name.

The CDE metric measures the average entropy of a class design, and gives an indication of the design complexity of the class. This is an ordinal measure that associates a number based on the usage and frequency of symbol operators in a class definition. It should be noted that this measure only provides for a relative comparison of complexity between two or more classes. No information can be drawn as to the scale, or level, of complexity that goes with each value of

the metric. Thus the CDE metric is not recommended for comparing classes from different design domains, since the operator sets used by the designs would be different. However, the CDE metric provides a way to order classes according to their relative complexity in a given design domain. This will allow identification of classes whose implementation would be overly complex, and those that are relatively trivial.

STUDY OF OO COMPLEXITY METRICS

Various classes and hierarchies of classes were chosen from three different C++ Graphical User Interface packages. Each class was examined by a team of highly trained C++ and GUI domain experts. The experts rated each class for complexity on a scale where not complex = 100%, fairly complex = 50%, and very complex = 0%. Then the AMC metric and the Li and Henry version of the WMC metric (both OO code complexity metrics) were calculated using the **PATR**icia system[17,18,19] (**P**rogram **A**nalysis **T**ool for **R**euse), and the CDE metric (an OO design metric) was calculated using **QMOOD++**[20,21,22]. The values of these metrics for each class examined are shown in Table 1. The rankings of the classes for each metric are shown in Table 2.

In Table 1, note that for class # 17 the AMC ranking is much closer to the expert's ranking than is the WMC ranking. This class is an example of a class with a large number of very simple member functions (see earlier discussion on the AMC metric). Also, for classes 1,4,5,9,10, 14, and 16, the AMC metric ranking better matches the experts' ranking. In some cases the AMC metric measures complexity better than does

the WMC metric; however, this is not true for all types of classes. Thus, AMC is not intended primarily as a replacement for the WMC metric (although since, as we will see later, it correlates well with the experts' determination of complexity it could be used as such), but rather as an additional way to examine particular classes for complexity.

Spearman's rank correlation was used to determine the correlation of the nonparametric data in Table 1. The correlation coefficient, r_s, is a measure of the ability of one rank-variable to predict the value of another rank-variable. If E_1 and E_2 are two independent evaluations of 'n' items that are to be correlated, then the values of E_1 and E_2 are ranked (either in increasing or decreasing order) from 1 to 'n' according to their relative size within the evaluations. For each E_1, E_2 pair in the relative rankings, the difference in the ranks 'd' is computer. The sum of all the d^2s, denoted Σd^2 is used to compute r_s according to the formula:

$$r_s = 1 - \frac{6 \Sigma d^2}{n (n^2 - 1)} \qquad -1.00 \leq r_s \leq 1.00$$

Using Spearman's rank correlation coefficient, each of the metrics, CDE, code complexity WMC (Li and Henry WMC), and AMC, was correlated separately to the experts' evaluations. Additionally, the CDE metric, which is a design complexity metric, was compared to the code complexity WMC metric.

Class Names	Class Numbers	Expert Ratings	CDE Metric Values	Li & Henry WMC Values	AMC Metric Values
wxObject	1	0.95	1.49	2	1
wxTimer	2	0.94	1.85	6	1.5
wxbTimer	3	0.99	2.36	4	0.67
wxEvent	4	0.98	2.59	6	0.75
GnContracts	5	0.97	2.88	3	1
wxMenu	6	0.5	2.92	36	2.77
GnMouseDownCommand	7	0.71	3.05	33	1.57
GnObject	8	0.96	3.07	10	1.11
wxbButton	9	0.9	3.16	8	1
wxButton	10	0.79	3.17	16	1.23
wxWindow	11	0.43	3.24	31	1.03
wxbMenu	12	0.86	3.34	26	2
wxbWindow	13	0.89	3.38	38	0.78
wxItem	14	0.64	3.4	21	1.24
GnCommand	15	0.92	3.67	28	1.27
wxMouseEvent	16	0.93	2.35	48	1.71
wxbItem	17	1	3.48	19	0.73

Table 1. Class Complexity Metric Values

Class Names	Class Numbers	Expert Rank	CDE Rank	WMC Rank	AMC Rank
WxObject	1	12	17	17	12
WxTimer	2	11	16	14	5
WxbTimer	3	16	14	15	17
WxEvent	4	15	13	13	15
GnContracts	5	14	12	16	13
WxMenu	6	2	11	3	1
GnMouseDownCommand	7	4	10	4	4
GnObject	8	13	9	11	9
WxbButton	9	8	8	12	11
WxButton	10	5	7	10	8
WxWindow	11	1	6	5	10
WxbMenu	12	6	5	7	2
WxbWindow	13	7	4	2	14
WxItem	14	3	3	8	7
GnCommand	15	9	1	6	6
WxMouseEvent	16	10	10	1	3
WxbItem	17	17	17	9	16

Table 2. Relative Ranking of the 17 Classes Based on the Metric Values

Complexity Correlation	Experts with CDE	Experts with WMC	Experts with AMC	CDE with WMC
Using the first 15 Classes	0.56	0.76	0.79	0.63
Using all 17 Classes	0.34	0.63	0.68	0.44

Table 3. Correlations Between Complexity Measures

We wish to test the hypothesis that there is a significant correlation between the current metric data set (either CDE, WMC, or AMC) and the experts' evaluations.

Hypothesis 1

H_0: $\rho = 0$. There is no significant correlation between the current metric and the experts' evaluations.

H_1: $\rho \neq 0$. There is a significant correlation between the current metric and the experts evaluations.

We also wish to test the hypothesis that there is a significant correlation between the design complexity metric CDE and the code complexity version of WMC.

Hypothesis 2

H_0: $\rho = 0$. There is no significant correlation between CDE and WMC.

H_1: $\rho \neq 0$. There is a significant correlation between CDE and WMC.

The rankings of the data used to calculate the Spearman's rank correlation coefficient are shown in Table 2. The Spearman's rank correlation coefficient for each test performed are shown in Table 3.

Complexity Metric Difficulties That Can affect Analysis

All the complexity metrics examined had difficulties with certain types of classes:

- abstract classes consisting solely of virtual functions
- classes with a number of member functions, where one particular member function was

much more complex than the other member functions.

In the case of a class consisting solely of virtual functions, the following problems were identified for each metric:

- The code complexity WMC metric, since it calculates the McCabe's cyclomatic complexity for each member function, calculates an empty virtual function as cyclomatic complexity = 1. Thus empty virtual functions add to the complexity measure of a class. The AMC metric has a similar problem, although due to the averaging nature of the metric the problem is not as bad as with the WMC metric.
- The design complexity WMC metric can include pure virtual functions in the method count, which can give an erroneous prediction of the implementation complexity of a class. For example, a function such as:

virtual void func1(void) = 0;

could be included in the method count. It is possible that a rule should be instituted for the measurement of the design complexity WMC metric, that pure virtual functions should not be included in the method count. (Empty virtual functions could eventually have default code added during the implementation process, and thus probably should take part in the WMC complexity calculation).

- The CDE metric, since it looks at string names, would consider each virtual function and its parameters as separate strings. This could give an erroneous prediction of the implementation complexity of an abstract class.

In the case of a class with a number of member functions, where one particular member function was much more complex than the other member functions:

- The design complexity WMC metric has the difficulty discussed earlier in this article, that it cannot differentiate at design time between a member function that will be implemented with complex code, and a member function whose implementation will be uncomplex.

For these reasons, the statistical analysis was performed with and without classes including the characteristics discussed above—thus Table 3 is shown with two categories, one with 17 classes, and one with 15 classes.

Results of Complexity Metrics Study

For a sample size of 17 and α = 5% (0.05), the Spearman's cutoff for accepting H_0 in Hypothesis 1 is 0.48. Since the computed r_s in the correlations for code complexity WMC and AMC is well above the cutoff, the null hypothesis H_0 of no correlation between WMC and the experts' evaluations, and of AMC and the experts' evaluations is rejected. For CDE, no significant correlation is shown between CDE and the experts evaluations. This is due to the difficulties discussed with the CDE metric in measuring abstract classes (similar to difficulties that other complexity metrics have with the same kind of class).

For the second test we remove the problem classes, which indeed can be considered to be statistical outliers. For a sample size of 15 and α = 5% (0.05), the Spearman's cutoff for accepting H_0 in Hypothesis 1 is 0.525. Since the

computed r_s in the correlations for code complexity WMC, AMC, and CDE is well above the cutoff, the null hypothesis H_0 of no correlation between WMC and the experts' evaluations, between AMC and the experts' evaluations, and between CDE and the experts' evaluations, is rejected. This indicates that there is indeed a significant correlation between WMC and the expert's evaluations, between AMC and the expert's evaluations, and between CDE and the expert's evaluations. A similar argument is made for Hypothesis 1, which compares CDE to WMC. Thus there is a significant correlation between CDE and WMC.

CONCLUSIONS

There are two different categories of OO complexity measures: code complexity measures and design complexity measures. Each type of complexity measure has advantages and disadvantages.

The Average Method Complexity (AMC) metric solves one particular problem with the code complexity WMC metric, that is, that it does not correctly measure the complexity of a class with a large number of uncomplex member functions. The AMC metric can be used itself as a separate code complexity measure, or it can be used in addition to the code complexity WMC metric to give two different views of the complexity of a class. The AMC metric correlates well with experts' evaluations of complexity.

The Class Design Entropy (CDE) metric solves a particular problem with the design complexity WMC metric, in that it can differentiate somewhat between the complexity of different member functions which have not yet been implemented, whereas the design

complexity WMC metric would simply count each separate member function equally. The CDE metric correlates well with experts' evaluations of complexity, and with the code complexity WMC metric, when certain types of fringe classes are not included.

References

1. McCabe, T.J. A complexity measure, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 2(4): 308-320, 1976.
2. Halstead, M.H. ELEMENTS OF SOFTWARE SCIENCE, Elsevier North-Holland, New York, 1977.
3. Chidamber, S.R., and C.F. Kemerer. Towards a metrics suite for object-oriented design, PROCEEDINGS: OOPSLA '91, July 1991, pp. 197-211.
4. Chidamber, S.R., and C.F. Kemerer. A metrics suite for object-oriented design, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 20(6) June 1994, pp. 476-493.
5. Li, W. and S. Henry. Maintenance metrics for the object-oriented paradigm, PROCEEDINGS OF THE FIRST INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, May 21-22, 1993, pp. 52-60.
6. Li, W., S. Henry, D. Kafury, and R. Schulman. Measuring object-oriented design, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, JULY/AUGUST 1995, pp. 48-55.
7. Churcher, N., and Shepperd, M.J., Comments on 'A Metrics Suite for Object Oriented Design', IEEE Transactions on Software Engineering, 21(3), March, 1995, pp. 263-265.
8. Babatz, R., Baecker, A., GINA Manual, Version 2.0, German National Research Institute for

Computer Science, anonymous ftp at ftp.gmd.de, directory gmd/ginaplus, 1991.

9. Chidamber, S. and Kemerer, C., Authors reply to "Comments on 'A Metrics Suite for Object-Oriented Design'", IEEE Transactions on Software Engineering, 21(3), March 1995, pp. 263-265.
10. Basili, V., L. Briand, and W.L. Melo, A validation of object-oriented metrics as quality indicators, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 22(10), October 1996, pp.751-761.
11. Smart, J., wxWindows Users Manual, Version 1.60, Artificial Intelligence Applications Institute, University of Edinburgh, Scotland, UK, http://www.aiai.ed.ac.uk/~jacs/wx, 1994.
12. Bansiya, J., and Davis, C., "An Entropy Based Complexity Measure For Object-Oriented Designs," Submitted to the journal Theory and Practice of Object Systems, November 1996.
13. Harrison, W., An Entropy-Based Measure of Software Complexity, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 18(11), Nov. 1992, pp. 1025-1029.
14. Torres, W.R., and Samadzadeh, M.H., Software Reuse and Information Theory Based Metrics, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 17(11), Nov. 1991, pp.1025-1029.
15. Davis, J., and LeBlanc, R., A Study of the Applicability of Complexity Measures, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 14(9), Sept., 1988, pp. 1366-1372.
16. Zweben, S., and Halstead, M., The Frequency Distribution of Operators in PL/1 Programs, IEEE

TRANSACTIONS ON SOFTWARE ENGINEERING, 5(11), Nov. 1979, pp. 91-95.

17. Etzkorn, L., Davis, C., and Li, W., "A Practical Look at the Lack of Cohesion in Methods Metric," Journal of Object-Oriented Programming, accepted, to appear 1998.
18. Etzkorn, L.H., and Davis, C.G., "Automatically Identifying Reusable Components in Object-Oriented Legacy Code," IEEE Computer, 30(10), October, 1997, pp. 66-71.
19. Etzkorn, Letha, A Metrics-Based Approach to the Automated Identification of Object-Oriented Reusable Software Components, doctoral dissertation, The University of Alabama in Huntsville, 1997.
20. Bansiya, J., Etzkorn, L., Davis, C., and Li, W., "A Class Cohesion Metric for Object-Oriented Design," Journal of Object-Oriented Programming, accepted, to appear 1998.
21. Bansiya, Jagdish, A Hierarchical Model for Quality Assessment of Object-Oriented Designs, doctoral dissertation, The University of Alabama in Huntsville, 1997.
22. Bansiya, J., and Davis, C., “Automated Metrics for Object-Oriented Development,” Dr. Dobb’s Journal, Vol. 272, December 1997, pp. 42-48.