the parent class can be diluted if some of the children are not appropriate members of the parent class. As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

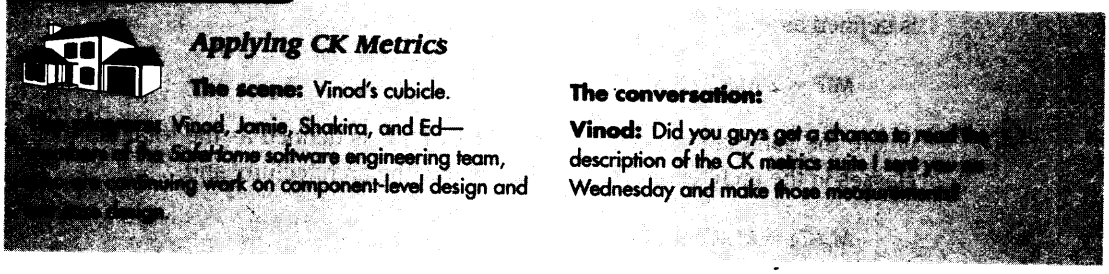
Coupling between object classes (CBO). The CRC model (Chapter 8) may be used to determine the value for CBO. In essence, CBO is the number of collaborations listed for a class on its CRC index card.[11] As CBO increases, it is likely that the reusability of a class will decrease. High values of CBO also complicate modifications and the testing that ensues when modifications are made. In general, the CBO values for each class should be kept as low as is reasonable. This is consistent with the general guideline to reduce coupling in conventional software.

The concepts of coupling and cohesion apply to both conventional and OO software. Keep class coupling low and class and operation cohesion high.

Response for a class (RFC). The response set of a class is "a set of methods that can potentially be executed in response to a message received by an object of that class" [CHI94]. RFC is the number of methods in the response set. As RFC increases, the effort required for testing also increases because the test sequence (Chapter 14) grows. It also follows that, as RFC increases, the overall design complexity of the class increases.

Lack of cohesion in methods (LCOM). Each method within a class, **C**, accesses one or more attributes (also called *instance variables*). LCOM is the number of methods that access one or more of the same attributes.[12] If no methods access the same attributes, then LCOM = 0. To illustrate the case where LCOM $\neq$ 0, consider a class with six methods. Four of the methods have one or more attributes in common (i.e., they access common attributes). Therefore, LCOM = 4. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. Although there are cases in which a high value for LCOM is justifiable, it is desirable to keep cohesion high; that is, keep LCOM low.[13]

SAFEHOME



Applying CK Metrics

The scene: Vinod's cubicle.

Vinod, Jamie, Shakira, and Ed—

SafeHome software engineering team,

work on component-level design and

The conversation:

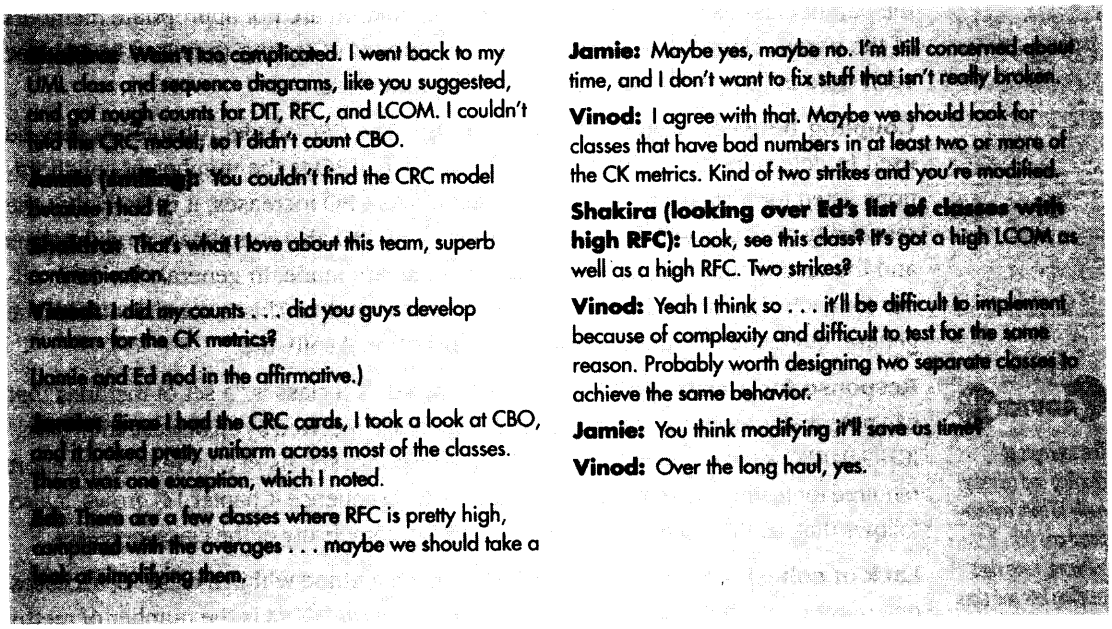
Vinod: Did you guys get a chance to read the description of the CK metrics suite I sent you on Wednesday and make those

---

11 If CRC index cards are developed manually, completeness and consistency must be assessed before CBO can be determined reliably.

12 The formal definition is a bit more complex. See [CHI94] for details.

13 The LCOM metric provides useful insight in some situations, but it can be misleading in others. For example, keeping coupling encapsulated within a class increases the cohesion of the system as a whole. Therefore, in at least one important sense, higher LCOM actually suggests that a class may have higher cohesion, not lower.



15.4.4 Class-Oriented Metrics—The MOOD Metrics Suite

Harrison, Counsell, and Nithi [HAR98] propose a set of metrics for object-oriented design that provide quantitative indicators for OO design characteristics. A small sampling of MOOD metrics follows:

Method inheritance factor (MIF). The degree to which the class architecture of an OO system makes use of inheritance for both methods (operations) and attributes is defined as

$$\text{MIF} = \Sigma\, M_i(C_i) / \Sigma\, M_a(C_i)$$

where the summation occurs over $i = 1$ to T_c. T_c is defined as the total number of classes in the architecture; C_i is a class within the architecture; and

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

where

$M_a(C_i)$ = the number of methods that can be invoked in association with C_i.
$M_d(C_i)$ = the number of methods declared in the class C_i.
$M_i(C_i)$ = the number of methods inherited (and not overridden) in C_i.

The value of MIF (the attribute inheritance factor, AIF, is defined in an analogous manner) provides an indication of the impact of inheritance on the OO software.

> "Analyzing OO software in order to evaluate its quality is becoming increasingly important as the [OO] paradigm continues to increase in popularity."
>
> Rachel Harrison et al.

Coupling factor (CF). Earlier in this chapter we noted that coupling is an indication of the connections between elements of the OO design. The MOOD metrics suite defines coupling in the following way:

$$CF = \Sigma_i \Sigma_j \, is\_client\,(C_i, C_j)/(T_c^2 - T_c)$$

where the summations occur over $i = 1$ to T_C and $j = 1$ to T_C. The function

is_client = 1, if and only if a relationship exists between the client class, C_c, and the server class, C_s, and $C_c \neq C_s$
= 0, otherwise

Although many factors affect software complexity, understandability, and maintainability, it is reasonable to conclude that, as the value for CF increases, the complexity of the OO software will also increase, and understandability, maintainability, and the potential for reuse may suffer as a result.

Harrison and her colleagues [HAR98] present a detailed analysis of MIF and CF, along with other metrics and examine their validity for use in the assessment of design quality.

15.4.5 OO Metrics Proposed by Lorenz and Kidd

In their book on OO metrics, Lorenz and Kidd [LOR94] divide class-based metrics into four broad categories that each have a bearing on component-level design: size, inheritance, internals, and externals. Size-oriented metrics for an OO design class focus on counts of attributes and operations for an individual class and average values for the OO system as a whole. Inheritance-based metrics focus on the manner in which operations are reused through the class hierarchy. Metrics for class internals look at cohesion and code-oriented issues, and external metrics examine coupling and reuse. A sampling of metrics proposed by Lorenz and Kidd follows:

During review of the analysis model, CRC index cards will provide a reasonable indication of expected values for CS. If you encounter a class with a large number of responsibilities, consider partitioning it.

Class size (CS). The overall size of a class can be determined with the following measures:

- The total number of operations (both inherited and private instance operations) that are encapsulated within the class.
- The number of attributes (both inherited and private instance attributes) that are encapsulated by the class.

The WMC metric proposed by Chidamber and Kemerer (Section 15.4.3) is also a weighted measure of class size. As we noted earlier, large values for CS indicate that

where the degree of coupling increases as the measures in Equation (15-6) increase.

Complexity metrics. A variety of software metrics can be computed to determine the complexity of program control flow. Many of these are based on the flow graph. As we discussed in Chapter 14, a graph is a representation composed of nodes and links (also called edges). When the links (edges) are directed, the flow graph is a directed graph.

McCabe and Watson [MCC94] identify a number of important uses for complexity metrics:

> Complexity metrics can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of source code [or procedural design information]. Complexity metrics also provide feedback during the software project to help control the [design activity]. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability.

Cyclomatic complexity is only one of a large number of complexity metrics.

The most widely used (and debated) complexity metric for computer software is cyclomatic complexity, originally developed by Thomas McCabe [MCC76], [MCC89] and discussed in detail in Chapter 14.

Zuse ([ZUS90], [ZUS97]) presents an encyclopedic discussion of no fewer than 18 different categories of software complexity metrics. The author presents the basic definitions for metrics in each category (e.g., there are a number of variations on the cyclomatic complexity metric) and then analyzes and critiques each. Zuse's work is the most comprehensive published to date.

15.4.7 Operation-Oriented Metrics

Because the class is the dominant unit in OO systems, fewer metrics have been proposed for operations that reside within a class. Churcher and Shepperd [CHU95] discuss this when they state: "Results of recent studies indicate that methods tend to be small, both in terms of number of statements and in logical complexity [WIL93], suggesting that the connectivity structure of a system may be more important than the content of individual modules." However, some insights can be gained by examining average characteristics for methods (operations). Three simple metrics, proposed by Lorenz and Kidd [LOR94], are appropriate:

Average operation size (OS_{avg}). Although lines of code could be used as an indicator for operation size, the LOC measure suffers from a set of problems discussed in Chapter 22. For this reason, the number of messages sent by the operation provides an alternative for operation size. As the number of messages sent by a single operation increases, it is likely that responsibilities have not been well-allocated within a class.

Operation complexity (OC). The complexity of an operation can be computed using any of the complexity metrics proposed for conventional software [ZUS90]. Be-

cause operations should be limited to a specific responsibility, the designer should strive to keep OC as low as possible.

Average number of parameters per operation (NP_{avg}). The larger the number of operation parameters, the more complex the collaboration between objects. In general, NP_{avg} should be kept as low as possible.

15.4.8 User Interface Design Metrics

Although there is significant literature on the design of human/computer interfaces (Chapter 12), relatively little information has been published on metrics that would provide insight into the quality and usability of the interface.

Sears [SEA93] suggests that *layout appropriateness* (LA) is a worthwhile design metric for human/computer interfaces. A typical GUI uses layout entities—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the "cost" of the transition from one layout entity to the next all contribute to the appropriateness of the interface.

> "You can learn at least one principal of user interface design by loading a dishwasher. If you crowd a lot in there, nothing gets very clean."
>
> Author unknown

Interface design metrics are fine, but above all else, be absolutely sure that your end-users like the interface and are comfortable with the interactions required.

Kokol and his colleagues [KOK95] define a cohesion metric for UI screens that measures the relative connection of on-screen content to other on-screen content. If data (or other content) presented on a screen belongs to a single major data object (as defined within the analysis model), UI cohesion for that screen is high. If many different types of data or content are presented and these data are related to different data objects, UI cohesion is low. The authors provide empirical models for cohesion [KOK95].

In addition, direct measures of UI interaction can focus on the measurement of time required to achieve a specific scenario or operation, time required to recover from an error condition, counts of specific operations or tasks required to achieve a use-case, the number of data or content objects presented on a screen, text density and size, and many others. However, these direct measures must be organized to create meaningful UI metrics that will lead to improved UI quality and/or improved usability.

It is important to note that the selection of a GUI design can be guided with metrics such as LA or UI screen cohesion, but the final arbiter should be user input based on GUI prototypes. Nielsen and Levy [NIE94] report that "one has a reasonably large chance of success if one chooses between interface [designs] based solely on users' opinions. Users' average task performance and their subjective satisfaction with a GUI are highly correlated."

Binder [BIN94] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system. The metrics consider aspects of encapsulation and inheritance. A sampling follows:

OO testing can be quite complex. Metrics can assist you in targeting testing resources at threads, scenarios, and packages of classes that are "suspect" based on measured characteristics. Use them.

Lack of cohesion in methods (LCOM).[15] The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

Percent public and protected (PAP). This metric indicates the percentage of class attributes that are public or protected. High values for PAP increase the likelihood of side effects among classes because public and protected attributes lead to high potential for coupling (Chapter 9).[16] Tests must be designed to ensure that such side effects are uncovered.

Public access to data members (PAD). This metric indicates the number of classes (or methods) that can access another class's attributes, a violation of encapsulation. High values for PAD lead to the potential for side effects among classes. Tests must be designed to ensure that such side effects are uncovered.

Number of root classes (NOR). This metric is a count of the distinct class hierarchies that are described in the design model. Test suites for each root class and the corresponding class hierarchy must be developed. As NOR increases, testing effort also increases.

Fan-in (FIN). When used in the OO context, fan-in for the inheritance hierarchy is an indication of multiple inheritance. FIN > 1 indicates that a class inherits its attributes and operations from more than one root class. FIN > 1 should be avoided when possible.

Number of children (NOC) and depth of the inheritance tree (DIT).[17] As we discussed in Chapter 14, superclass methods will have to be retested for each subclass.

15.7 METRICS FOR MAINTENANCE

All of the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

---

15 See Section 15.4.3 for a description of LCOM.

16 Some people promote designs in which none of the attributes are public or private; that is, PAP = 0. This implies that all attributes must be accessed in other classes via methods.

17 See Section 15.4.3 for a description of NOC and DIT.

M_T = the number of modules in the current release
F_c = the number of modules in the current release that have been changed
F_a = the number of modules in the current release that have been added
F_d = the number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$SMI = [M_T - (F_a + F_c + F_d)]/M_T$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

SOFTWARE TOOLS

Product Metrics

Objective: To assist software engineers in developing meaningful metrics that assess the work products produced during analysis and design modeling, source code generation, and testing.

Mechanics: Tools in this category span a broad array of metrics and are implemented either as standalone applications or (more commonly) as functionality that exists within tools for analysis and design, coding or testing. In most cases, the metrics tool analyzes a representation of the software (e.g., a UML model or source code) and develops one or more metrics as a result.

Representative Tools[18]

Krakatau Metrics, developed by Power Software (www.powersoftware.com/products), computes complexity, Halstead, and related metrics for C/C++ and Java.

Metrics4C, developed by +1 Software Engineering (www.plus-one.com/Metrics4C_fact_sheet.html), computes a variety of architectural, design, and code-oriented metrics as well as project-oriented metrics.

Rational Rose, developed by Rational Corporation (www.rational.com), is a comprehensive tool set for UML modeling that incorporates a number of metrics analysis features.

RSM, developed by M-Squared Technologies (msquaredtechnologies.com/m2rsm/index.html), computes a wide variety of code-oriented metrics for C, C++ and Java.

Understand, developed by Scientific Toolworks, Inc. (www.scitools.com), calculates code-oriented metrics for a variety of programming languages.

15.8 SUMMARY

Software metrics provide a quantitative way to assess the quality of internal product attributes, thereby enabling a software engineer to assess quality before the product is built. Metrics provide the insight necessary to create effective analysis and design models, solid code, and thorough tests.

18 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

To be useful in a real world context, a software metric must be simple and computable, persuasive, consistent, and objective. It should be programming language independent and provide effective feedback to the software engineer.

Metrics for the analysis model focus on function, data, and behavior—the three components of the analysis model. Metrics for design consider architecture, component-level design, and interface design issues. Architectural design metrics consider the structural aspects of the design model. Component-level design metrics provide an indication of module quality by establishing indirect measures for cohesion, coupling, and complexity. User interface design metrics provide an indication of the ease with which a GUI can be used.

Metrics for OO systems focus on measurement that can be applied to the class and design characteristics—localization, encapsulation, information hiding, inheritance, and object abstraction techniques—that make the class unique.

Halstead provides an intriguing set of metrics at the source code level. Using the number of operators and operands present in the code, a variety of metrics are developed to assess program quality.

Few product metrics have been proposed for direct use in software testing and maintenance. However, many other product metrics can be used to guide the testing process and as a mechanism for assessing the maintainability of a computer program. A wide variety of OO metrics have been proposed to assess the testability of an OO system.

REFERENCES

[ALB79] Albrecht, A. J., "Measuring Application Development Productivity," *Proc. IBM Application Development Symposium,* Monterey, CA, October 1979, pp. 83–92.

[ALB83] Albrecht, A. J., and J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Engineering,* November 1983, pp. 639–648.

[BAS84] Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering,* vol. SE-10, 1984, pp. 728–738.

[BER95] Berard, E., "Metrics for Object-Oriented Software Engineering," an Internet posting on comp.software-eng, January 28, 1995.

[BIE94] Bieman, J. M., and L. M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Engineering,* vol. SE-20, no. 8, August 1994, pp. 308–320.

[BIN94] Binder, R. V., "Object-Oriented Software Testing," *CACM,* vol. 37, no. 9, September 1994, p. 29.

[BRI96] Briand, L. C., S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. Software Engineering,* vol. SE-22, no. 1, January 1996, pp. 68–85.

[CAR90] Card, D. N., and R. L. Glass, *Measuring Software Design Quality,* Prentice-Hall, 1990.

[CAV78] Cavano, J. P., and J. A. McCall, "A Framework for the Measurement of Software Quality," *Proc. ACM Software Quality Assurance Workshop,* November 1978, pp. 133–139.

[CHA89] Charette, R. N., *Software Engineering Risk Analysis and Management,* McGraw-Hill/Intertext, 1989.

[CHI94] Chidamber, S. R., and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Engineering,* vol. SE-20, no. 6, June 1994, pp. 476–493.