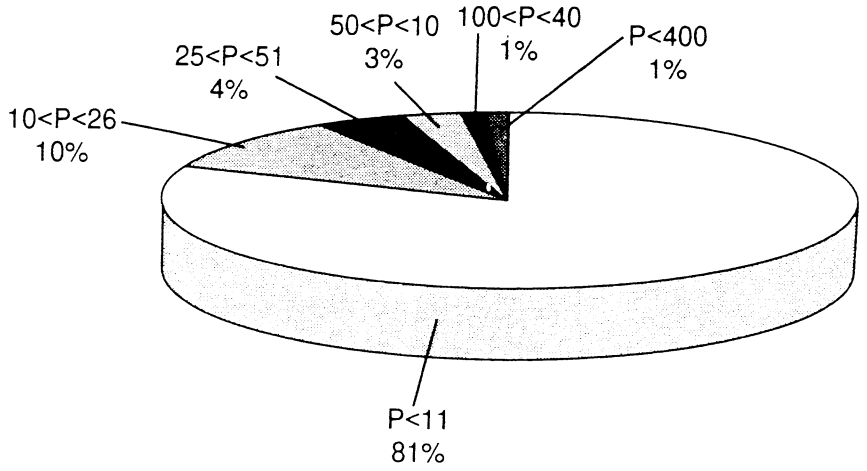

Figure 8.26: Bieman–Schultz results. *P* represents the number of complete paths required to satisfy the all *du*-paths strategy.

EXAMPLE 8.35: Figure 8.26 summarizes the results of an empirical study of the all *du*-paths strategy. Bieman and Schultz looked at a commercial system consisting of 143 modules. For each module, they computed *P*, the minimum number of paths required to satisfy all *du*-paths. Bieman and Schultz found that, for 81% of all modules, *P* was less than 11, and in only one module was *P* prohibitively large. Thus, the strategy seems to be practical for almost all modules. Moreover, Bieman and Schultz noted that the module with excessively high *P* was known to be the "rogue" module of the system (Bieman and Schultz, 1992). Thus, *P* may be a useful quality assurance measure, especially when used with outlier analysis as discussed in Chapter 6. To assure quality, we may want to review (and, if necessary, rewrite) those few modules with infeasible values of *P*, since it is likely that they are overly complex.

8.4 OBJECT-ORIENTED METRICS

The rise in popularity of object-oriented methods raises questions about how we measure object oriented structures. Early measures for object orientation focused on cost and effort prediction (Pfleeger, 1989; Pfleeger and Palmer, 1990), and size was the only internal attribute addressed. Many of the design measures presented so far in this chapter are derived from traditional design techniques; some, such as coupling and cohesion, can be interpreted easily for object-oriented approaches (Pfleeger, 1991), but others have been developed specifically for object orientation.

Chidamber and Kemerer have suggested measures for object-oriented systems, and we discuss them in depth (Chidamber and Kemerer, 1994). The basis for the empirical relation systems in the Chidamber–Kemerer work is the set of "ontological principles"

proposed by Bunge and later applied to object-oriented systems by Yand and Weber. (Bunge, 1979; Yand and Weber, 1990) In this work, the world is viewed as being composed of **substantial individuals** that possess a finite set of **properties**. Collectively, a substantial individual and its properties constitute an **object**. A **class** is a set of objects that have common properties, and a **method** is an operation on an object that is defined as part of the declaration of the class.

Attributes such as coupling, cohesion, object complexity, and scope of properties are then defined in Bunge's "ontological" terms.

> **EXAMPLE 8.36:** In Bunge's terminology, two objects are coupled if and only if at least one of them acts upon the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a substantial individual traverses in time.

Chidamber and Kemerer use these notions to define a number of metrics that are claimed to relate to some of these attributes:

Metric 1: weighted methods per class (WMC)

This metric is intended to relate to the notion of complexity. For a class C with methods $M_1, M_2, \ldots, M_n$, weighted respectively with "complexity" $c_1, c_2, \ldots, c_n$, the measure is calculated as

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

Metric 2: depth of inheritance tree (DIT)

In an object-oriented design, the application domain is modeled as a hierarchy of classes. This hierarchy can be represented as a tree, called the **inheritance tree**. The nodes in the tree represent classes, and for each such class, the DIT metric is the length of the maximum path from the node to the root of the tree. This measure relates to the notion of scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class.

Metric 3: number of children (NOC)

This metric relates to a node (class) of the inheritance tree. It is the number of immediate successors of the class.

Metric 4: coupling between object classes (CBO)

For a given class, this measure is defined to be the number of other classes to which the class is coupled.

Metric 5: response for class (RFC)

This measure captures the size of the response set of a class. The response set of a class consists of all the methods called by local methods. RFC is the number of local methods plus the number of methods called by local methods.

Metric 6: lack of cohesion metric (LCOM)

The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class. LCOM is defined as the number of disjoint (that is, non-intersecting) sets of local methods.

Li and Henry used these six metrics plus several others, including some simple size metrics (such as the number of attributes plus the number of local methods), to determine whether object-oriented metrics could predict maintenance effort. For the weights in Metric 1 (WMC), they used cyclomatic number. On the basis of an empirical study using regression analysis, they concluded that these measures are indeed useful. In particular, they claim that the Chidamber–Kemerer metrics "contribute to the prediction of maintenance effort over and beyond what can be predicted using size metrics alone" (Li and Henry, 1993).

While the use of object-oriented metrics looks promising, there is as yet no widespread agreement on what should be measured in object-oriented systems and which metrics are appropriate. Indeed, Churcher and Shepperd criticized the Chidamber–Kemerer metrics on the grounds that there is as yet no consensus on the underlying attributes. For example, they argue that even the notion of "methods per class" is ambiguous (Churcher and Shepperd, 1995).

Still, these measures can be useful when restricted to locally specified, commonly accepted definitions of the underlying terms. For example, Lorenz reports that the analysis of object-oriented code at IBM has led to several recommendations about design, including maximum sizes for number of methods per class, and restrictions on the number of different message types sent and received (Lorenz, 1993).

8.5 DATA STRUCTURE

We have seen how information flow (sometimes also called data flow) can be measured, so that we have some sense of how data interact with a system or module. However, there have been few attempts to define measures of actual data items and their structure. In this section, we examine possible data-structure measures to see what they can tell us about system products.

In principle, measuring data structure should be straightforward. As with control flow and data flow, we can consider both local and global measures. Locally, we want to measure the amount of structure in each data item. Unfortunately, there has been little work in this area. Elliott has used a graph-theoretic approach to analyze and measure properties of individual data structures, using the same kind of decomposition that we saw for control structure (Elliott, 1988). He viewed the simple data types (such as integers, characters, and Booleans) as primes, and then considered the various operations that enable us to build more complex data structures (such as the operation of forming an array of a given type). Data structure measures can then be defined hierarchically in terms of values for the primes and values associated

8.6 DIFFICULTIES WITH GENERAL "COMPLEXITY" MEASURES

We have discussed the need for viewing "complexity" as the combination of several attributes, and we have shown the importance of examining each attribute separately, so that we can understand exactly what it is that is responsible for the overall "complexity." Nevertheless, practitioners and researchers alike find great appeal in generating a single, comprehensive measure to express "complexity." The single measure is expected to have powerful properties, being an indicator of such diverse notions as comprehensibility, correctness, maintainability, reliability, testability, and ease of implementation. Thus, a high value for "complexity" should indicate low comprehensibility, low reliability, and so on. Sometimes this measure is called a "quality" measure, as it purportedly relates to external product attributes such as reliability and maintainability. Here, a high "quality" measure suggest a low-quality product.

The danger in attempting to find measures to characterize a large collection of different attributes is that often the measures address conflicting goals, counter to the representational theory of measurement.

EXAMPLE 8.39: Suppose we define a measure, M, that characterizes the quality of people. If M existed, it would have to satisfy at least the following conditions:

$M(\text{A}) > M(\text{B})$ whenever A is stronger than B and
$M(\text{A}) > M(\text{B})$ whenever A is more intelligent than B

The fact that some highly intelligent people are very weak physically ensures that no M can satisfy both these properties.

Example 8.39 illustrates clearly why single-valued measures of software "complexity" are doomed to failure. Consider, for example, the list of properties in Table 8.3. Proposed by Weyuker, they are described as properties that should be satisfied by any good "complexity" metric (Weyuker, 1988). In the table, P, Q, R denote any program blocks.

Zuse has used the representational theory of measurement to prove that some of these properties are contradictory; they can never all be satisfied by a single-valued measure (Zuse, 1992).

EXAMPLE 8.40: We can see intuitively why Properties 5 and 6 in Table 8.3 are mutually incompatible. Property 5 asserts that adding code to a program cannot decrease its complexity. This property reflects the view that program size is a key factor in its complexity. We can also conclude from Property 5 that low comprehensibility is not a key factor in complexity. This statement is made because it is widely believed that, in certain cases, we can understand a program more easily as we see more of it. Thus, while a complexity measure M based primarily on size should satisfy property 5, a complexity measure M based primarily on comprehensibility cannot satisfy property 5.

Table 8.3: Weyuker's properties for any software complexity metric *M*

Property 1:	There are programs P and Q for which $M(P) \neq M(Q)$.
Property 2:	If c is a non-negative number, then there are only finitely many programs P for which $M(P) = c$.
Property 3:	There are distinct programs P and Q for which $M(P) = M(Q)$.
Property 4:	There are functionally equivalent programs P and Q for which $M(P) \neq M(Q)$.
Property 5:	For any program bodies P and Q, we have $M(P) \leq M(P;Q)$ and $M(Q) \leq M(P;Q)$.
Property 6:	There exist program bodies P, Q, and R such that $M(P) = M(Q)$ and $M(P;R) \neq M(Q;R)$.
Property 7:	There are program bodies P and Q such that Q is formed by permuting the order of the statements of P and $M(P) \neq M(Q)$.
Property 8:	If P is a renaming of Q, then $M(P) = M(Q)$.
Property 9:	There exist program bodies P and Q such that $M(P)+M(Q) < M(P;Q)$.

On the other hand, Property 6 asserts that we can find two program bodies of equal complexity which, when separately concatenated to a same third program, yield programs of different complexity. Clearly this property has much to do with comprehensibility and little to do with size.

Thus, properties 5 and 6 are relevant for very different, and incompatible, views of complexity. They cannot both be satisfied by a single measure that captures notions of size and low comprehensibility. The above argument is not formal. However, Zuse reinforces the incompatibility; he proves formally that while Property 5 explicitly requires the ratio scale for *M*, Property 6 explicitly excludes the ratio scale.

Cherniavsky and Smith also offer a critique of Weyuker's properties (Cherniavsky and Smith, 1991). They define a code-based "metric" that satisfies all of Weyuker's properties but, as they rightly claim, is not a sensible measure of complexity. They conclude that axiomatic approaches may not work.

Cherniavsky and Smith have correctly found a problem with the Weyuker axioms. But they present no justification for their conclusion about axiomatic approaches in general. They readily accept that Weyuker's properties are not claimed to be complete. But what they fail to observe is that Weyuker did not propose that the axioms were sufficient; she only proposed that they were necessary. The Cherniavsky and Smith "metric" is not a real measure in the sense of measurement theory, since it does not capture a specific attribute. Therefore, showing that the "metric" satisfies a set of axioms necessary for any measure proves nothing about real measures.

These problems could have been avoided by heeding a basic principle of measurement theory: defining a numerical mapping does not in itself constitute measurement. Software engineers often use the word "metric" for any number extracted from a software entity. But while every measure is a metric, the converse is