a second problem. Now that the group has seen two problems related to the same aspect of the design, comments likely will come thick and fast, with interruptions every few sentences. In a few minutes, this whole area of the design could be thoroughly explored, and any problems would be obvious.

The time and location of the inspection should be planned to avoid all outside interruptions. The optimal amount of time for the inspection session appears to be from 90 to 120 minutes. Since the session is a mentally taxing experience, longer sessions tend to be less productive. Most inspections proceed at a rate of approximately 150 program statements per hour. For that reason, large programs should be examined in multiple inspections, each inspection dealing with one or several modules or subroutines.

Note that for the inspection process to be effective, the appropriate attitude must be established. If the programmer views the inspection as an attack on his or her character and adopts a defensive posture, the process will be ineffective. Rather, the programmer must approach the process with an egoless attitude and must place the process in a positive and constructive light. The objective of the inspection is to find errors in the program, thus improving the quality of the work. For this reason, most people recommend that the results of an inspection be a confidential matter, shared only among the participants. In particular, if managers somehow make use of the inspection results, the purpose of the process can be defeated.

The inspection process also has several beneficial side effects in addition to its main effect of finding errors. For one thing, the programmer usually receives feedback concerning programming style, choice of algorithms, and programming techniques. The other participants gain in a similar way by being exposed to another programmer’s errors and programming style. Finally, the inspection process is a way of identifying early the most error-prone sections of the program, helping to focus more attention on these sections during the computer-based testing processes (one of the testing principles of Chapter 2).

An Error Checklist for Inspections

An important part of the inspection process is the use of a checklist to examine the program for common errors. Unfortunately, some checklists concentrate more on issues of style than on errors (for example, “Are comments accurate and meaningful?” and “Are if-else, code blocks, and do-while groups aligned?”), and the error checks are too nebulous to be useful (such as “Does the code meet the design requirements?”). The checklist in this section was compiled after many years of study of software errors. The checklist is largely language independent, meaning that most of the errors can occur with any programming language. You may wish to supplement this list with errors peculiar to your programming language and with errors detected after using the inspection process.

Data Reference Errors

1. Does a referenced variable have a value that is unset or uninitialized? This probably is the most frequent programming error: it occurs in a wide variety of circumstances. For each reference to a data item (variable, array element, field in a structure), attempt to “prove” informally that the item has a value at that point.
2. For all array references, is each subscript value within the defined bounds of the corresponding dimension?
3. For all array references, does each subscript have an integer value? This is not necessarily an error in all languages, but it is a dangerous practice.
4. For all references through pointer or reference variables, is the referenced memory currently allocated? This is known as the “dangling reference” problem. It occurs in situations where the lifetime of a pointer is greater than the lifetime of the referenced memory. One situation occurs where a pointer references a local variable within a procedure, the pointer value is assigned to an output parameter or a global variable, the procedure returns (freeing the referenced loca-
tion), and later the program attempts to use the pointer value. In a manner similar to checking for the prior errors, try to prove informally that, in each reference using a pointer variable, the reference memory exists.

5. When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names? Situations to look for are the use of the EQUVALECE statement in FORTRAN, and the REDEFINES clause in COBOL. As an example, a FORTRAN program contains a real variable A and an integer variable B; both are made aliases for the same memory area by using an EQUVALECE statement. If the program tries to use a value into A and then references variable B, an error is likely present since the machine would use the floating-point bit representation in the memory area as an integer.

6. Does a variable's value have a type or attribute other than what the compiler expects? This situation might occur where a C, C++, or COBOL program reads a record into memory and references it by using a structure, but the physical representation of the record differs from the structure definition.

7. Are there any explicit or implicit addressing problems if, on the machine being used, the units of memory allocation are smaller than the units of memory accessibility? For instance, in some environments, fixed-length bit strings do not necessarily begin on byte boundaries, but addresses only point to byte boundaries. If a program computes the address of a bit string and later refers to the string through this address, the wrong memory location may be referenced. This situation also could occur when passing a bit-string argument to a subroutine.

8. If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects? An example of such an error is where a C++ pointer upon which a data structure is based is assigned the address of a different data structure.

9. If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?

10. When indexing into a string, are the limits of the string off-by-one errors in indexing operations or in subscript references to arrays?

11. For object-oriented languages, are all inheritance requirements met in the implementing class?

Data-Declaration Errors

1. Have all variables been explicitly declared? A failure to do so is not necessarily an error, but it is a common source of trouble. For instance, if a program subroutine receives an array parameter, and fails to define the parameter as an array (as in a DIMENSION statement, for example), a reference to the array (such as $a[11]$) is interpreted as a function call, leading to the machine's attempting to execute the array as a program. Also, if a variable is not explicitly declared in an inner procedure or block, is it understood that the variable is shared with the enclosing block?

2. If all attributes of a variable are not explicitly stated in the declaration, are the defaults well understood? For instance, the default attributes received in Java are often a source of surprise.

3. Where a variable is initialized in a declarative statement, is it properly initialized? In many languages, initialization of arrays and strings is somewhat complicated and, hence, error prone.

4. Is each variable assigned the correct length and datatype?

5. Is the initialization of a variable consistent with its memory type? For instance, if a variable in a FORTRAN subroutine needs to be reinitialized each time the subroutine is called, it must be initialized with an assignment statement rather than a DATA statement.
6. Are there any variables with similar names (volt and volts, for example)? This is not necessarily an error, but it should be seen as a warning that the names may have been confused somewhere within the program.

**Computation Errors**

1. Are there any computations using variables having inconsistent (such as nonarithmetic) datatypes?

2. Are there any mixed-mode computations? An example is the addition of a floating-point variable to an integer variable. Such occurrences are not necessarily errors, but they should be explored carefully to ensure that the language's conversion rules are understood. Consider the following Java snippet showing the rounding error that can occur when working with integers:

   ```java
   int x = 1;
   int y = 2;
   int z = 0;
   z = x/y;
   System.out.println("z = " + z);
   OUTPUT:
   z = 0
   ```

3. Are there any computations using variables having the same datatype but different lengths?

4. Is the datatype of the target variable of an assignment smaller than the datatype or result of the right-hand expression?

5. Is an overflow or underflow expression possible during the computation of an expression? That is, the end result may appear to have valid value, but an intermediate result might be too big or too small for the programming language's datatypes.

6. Is it possible for the divisor in a division operation to be zero?

7. If the underlying machine represents variables in base-2 form, are there any sequences of the resulting inaccuracy? That is, 10 * 0.1 is rarely equal to 1.0 on a binary machine.

8. Where applicable, can the value of a variable go outside the meaningful range? For example, statements assigning a value to the variable probability might be checked to ensure that the assigned value will always be positive and not greater than 1.0.

9. For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?

10. Are there any invalid uses of integer arithmetic, particularly divisions? For instance, if `i` is an integer variable, whether the expression `2*i/2` depends on whether `i` has an even value and whether the multiplication or division is performed first.

**Comparison Errors**

1. Are there any comparisons between variables having different datatypes, such as comparing a character string to an address, date, or number?

2. Are there any mixed-mode comparisons or comparisons between variables of different lengths? If so, ensure that the conversion rules are well understood.

3. Are the comparison operators correct? Programmers frequently confuse such relations as at most, at least, greater than, not less than, less than or equal.

4. Does each Boolean expression state what it is supposed to state? Programmers often make mistakes when writing logical expressions involving and, or, and not.

5. Are the operands of a Boolean operator Boolean? Have comparison and Boolean operators been erroneously mixed together? This represents another frequent class of mistakes. Examples of a few typical mistakes are illustrated here. If you want to determine whether `i` is between 2 and 10, the
expression \( x \leq 10 \) is incorrect; instead, it should be \( x \leq 11 \) and \( x \leq 10 \). If you want to determine whether \( z \) is greater than \( x \) or \( y \), \( z > x \) \( \&\& \) \( z > y \) is incorrect; instead, it should be \( z > x \) \( \| \) \( z > y \). If you want to compare three numbers for equality, \( x = y = c \) does something quite different. If you want to test the mathematical relation \( x > y > z \), the correct expression is \( x = y \&\& y = z \).

6. Are there any comparisons between fractional or floating-point numbers that are represented in base-2 by the underlying machine? This is an occasional source of errors because of truncation and base-2 approximations of base-10 numbers.

7. For expressions containing more than one Boolean operator, are the assumptions about the order of evaluation and the precedence of operators correct? That is, if you see an expression such as \( (a = 2) \&\& (b = 2) \| (c = 3) \), is it well understood whether the \&\& or the \| is performed first?

8. Does the way in which the compiler evaluates Boolean expressions affect the program? For instance, the statement

\[
\text{if} \,(x = 0) \&\& (x > 2) 
\]

may be acceptable for compilers that end the test as soon as one side of an and is false, but may cause a division-by-zero error with other compilers.

**Control-Flow Errors**

1. If the program contains a multiway branch such as a computed \( 60 \) to, can the index variable ever exceed the number of branch possibilities? For example, in the statement

\[
60 \text{ to} \{200,300,400\}, \text{i} 
\]

will \( i \) always have the value of 1, 2, or 3?

2. Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate.

3. Will the program, module, or subroutine eventually terminate?

4. Is it possible that, because of the conditions upon entry, a loop will never execute? If so, does this represent an oversight? For instance, if you had the following loops headed by the following statements:

\[
\text{for } (\text{i} = 1; \text{i} < 2; \text{i}++) \{
\text{...}
\}
\]

\[
\text{while } (\text{NOTFOUND}) \{
\text{...}
\}
\]

what happens if NOTFOUND is initially false or if \( x \) is greater than \( z \)?

5. For a loop controlled by both iteration and a Boolean condition (a searching loop, for example) what are the consequences of loop fall-through? For example, for the pseudo-code loop headed by

\[
\text{DO } i = 1 \text{ TO } \text{TABLESIZE} \text{ WHILE } (\text{NOTFOUND})
\]

what happens if NOTFOUND never becomes false?

6. Are there any off-by-one errors, such as one too many or too few iterations? This is a common error in zero-based loops. You will often forget to count "0" as a number. For example, if you want to create Java code for a loop that counted to 10, the following would be wrong, as it counts to 11:

\[
\text{for } (\text{i} = 0; \text{i} < 10; \text{i}++) \{
\text{System.out.println(i);} 
\}
\]

Correct, the loop is iterated 10 times:

\[
\text{for } (\text{i} = 0; \text{i} < 10; \text{i}++) \{
\text{System.out.println(i);} 
\}
7. If the language contains a concept of statement groups or code blocks (e.g., do while or if...), is there an explicit while for each group and do the do’s correspond to their appropriate groups? Or is there a closing bracket for each open bracket? Most modern compilers will complain of such mismatches.

8. Are there any nonexhaustive decisions? For instance, if an input parameter’s expected values are 1, 2, or 3, does the logic assume that it must be 3 if it is not 1 or 2? If so, is the assumption valid?

**Interface Errors**

1. Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules? Also, is the order correct?

2. Do the attributes (e.g., datatype and size) of each parameter match the attributes of each corresponding argument?

3. Does the units system of each parameter match the units system of each corresponding argument? For example, is the parameter expressed in degrees but the argument expressed in radians?

4. Does the number of arguments transmitted by this module to another module equal the number of parameters expected by that module?

5. Do the attributes of each argument transmitted to another module match the attributes of the corresponding parameter in that module?

6. Does the units system of each argument transmitted to another module match the units system of the corresponding parameter in that module?

7. If built-in functions are invoked, are the number, attributes, and order of the arguments correct?

8. If a module or class has multiple entry points, is a parameter ever referenced that is not associated with the current point of entry? Such an error exists in the second assignment statement in the following PL/1 program:

   ```pli
   PROGML : 
   INIT: 
   RETURN
   ENTRY (Y, Z)
   Y+1:
   END:
   ```

9. Does a subroutine alter a parameter that is intended to be only an input value?

10. If global variables are present, do they have the same definition and attributes in all modules that reference them?

11. Are constants ever passed as arguments? In some FORTRAN implementations a statement such as

   ```fortran
   CALL SUBR, 3
   ```

   is dangerous, since if the subroutine SUBR assigns a value to its second parameter, the value of the constant 3 will be altered.

**Input/Output Errors**

1. If files are explicitly declared, are their attributes correct?

2. Are the attributes on the file’s OPEN statement correct?

3. Does the format specification agree with the information in the I/O statement? For instance, in FORTRAN, does each FORMAT statement agree (in terms of the number and attributes of the items) with the corresponding READ or WRITE statement?

4. Is there sufficient memory available to hold the file your program will read?

5. Have all files been opened before use?

6. Have all files been closed after use?

7. Are end-of-file conditions detected and handled correctly?

8. Are I/O error conditions handled correctly?

9. Are there spelling or grammatical errors in any text that is printed or displayed by the program?
Table 3.1

**Inspection Error Checklist Summary, Part 1**

<table>
<thead>
<tr>
<th>Data Reference</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Unset variable used?</td>
<td>1. Computations on nonarithmetic variables?</td>
</tr>
<tr>
<td>3. Non integer subscripts?</td>
<td>3. Computations on variables of different lengths?</td>
</tr>
<tr>
<td>4. Dangling references?</td>
<td>4. Target size less than size of assigned value?</td>
</tr>
<tr>
<td>5. Correct attributes when aliasing?</td>
<td>5. Intermediate result overflow or underflow?</td>
</tr>
<tr>
<td>6. Record and structure attributes match?</td>
<td>6. Division by zero?</td>
</tr>
<tr>
<td>8. Based storage attributes correct?</td>
<td>8. Variable's value outside of meaningful range?</td>
</tr>
<tr>
<td>10. Off-by-one errors in indexing or subscripting operations?</td>
<td>10. Integer divisions correct?</td>
</tr>
<tr>
<td>11. Are inheritance requirements met?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Declaration</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. All variables declared?</td>
<td>1. Comparisons between inconsistent variables?</td>
</tr>
<tr>
<td>3. Arrays and strings initialized properly?</td>
<td>3. Comparison relationships correct?</td>
</tr>
<tr>
<td>4. Correct lengths, types, and storage classes assigned?</td>
<td>4. Boolean expressions correct?</td>
</tr>
<tr>
<td>5. Initialization consistent with storage class?</td>
<td>5. Comparison and Boolean expressions mixed?</td>
</tr>
<tr>
<td>6. Any variables with similar names?</td>
<td>6. Comparisons of base-2 fractional values?</td>
</tr>
<tr>
<td>8. Compiler evaluation of Boolean expressions understood?</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2

**Inspection Error Checklist Summary, Part II**

<table>
<thead>
<tr>
<th>Control Flow</th>
<th>Input/Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Multiway branches exceeded?</td>
<td>1. File attributes correct?</td>
</tr>
<tr>
<td>2. Will each loop terminate?</td>
<td>2. OPEN statements correct?</td>
</tr>
<tr>
<td>4. Any loop bypasses because of entry conditions?</td>
<td>4. Buffer size matches record size?</td>
</tr>
<tr>
<td>5. Are possible loop fall-throughs correct?</td>
<td>5. Files opened before use?</td>
</tr>
<tr>
<td>6. Off-by-one iteration errors?</td>
<td>6. Files closed after use?</td>
</tr>
<tr>
<td>7. DO/END statements match?</td>
<td>7. End-of-file conditions handled?</td>
</tr>
<tr>
<td>8. Any nonexhaustive decisions?</td>
<td>8. I/O errors handled?</td>
</tr>
<tr>
<td>9. Any textual or grammatical errors in output information?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface</th>
<th>Other Checks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Number of input parameters equal to number of arguments?</td>
<td>1. Any unreferenced variables in cross-reference listing?</td>
</tr>
<tr>
<td>2. Parameter and argument attributes match?</td>
<td>2. Attribute list what was expected?</td>
</tr>
<tr>
<td>3. Parameter and argument units system match?</td>
<td>3. Any warning or informational messages?</td>
</tr>
<tr>
<td>4. Number of arguments transmitted to called modules equal to number of parameters?</td>
<td>4. Input checked for validity?</td>
</tr>
<tr>
<td>5. Attributes of arguments transmitted to called modules equal to attributes of parameters?</td>
<td>5. Missing function?</td>
</tr>
<tr>
<td>6. Units system of arguments transmitted to called modules equal to units system of parameters?</td>
<td></td>
</tr>
<tr>
<td>7. Number, attributes, and order of arguments to built-in functions correct?</td>
<td></td>
</tr>
<tr>
<td>8. Any references to parameters not associated with current point of entry?</td>
<td></td>
</tr>
<tr>
<td>9. Input-only arguments altered?</td>
<td></td>
</tr>
<tr>
<td>10. Global variable definitions consistent across modules?</td>
<td></td>
</tr>
<tr>
<td>11. Constants passed as arguments?</td>
<td></td>
</tr>
</tbody>
</table>
Other Checks

1. If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.
2. If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.
3. If the program compiled successfully, but the computer produced one or more “warning” or “informational” messages, check each one carefully. Warning messages are indications that the compiler suspects that you are doing something of questionable validity; all of these suspicions should be reviewed. Informational messages may list undeclared variables or language uses that impede code optimization.
4. Is the program or module sufficiently robust? That is, does it check its input for validity?
5. Is there a function missing from the program?

This checklist is summarized in Tables 3.1 and 3.2 on pages 36–37.

Walkthroughs

The code walkthrough, like the inspection, is a set of procedures and error-detection techniques for group code reading. It shares much in common with the inspection process, but the procedures are slightly different, and a different error-detection technique is employed.

Like the inspection, the walkthrough is an uninterrupted meeting of one to two hours in duration. The walkthrough team consists of three to five people. One of these people plays a role similar to that of the moderator in the inspection process, another person plays the role of a secretary (a person who records all errors found), and a third person plays the role of a tester. Suggestions are to who the three to five people should be vary. Of course, the programmer is one of those people. Suggestions for the other participants include (1) a highly experienced programmer, (2) a programming-language expert, (3) a new programmer (to give a fresh, unbiased outlook), (4) the person who will eventually maintain the program, (5) someone from a different project, and (6) someone from the same programming team as the programmer.

The initial procedure is identical to that of the inspection process: The participants are given the materials several days in advance to allow them to bone up on the problem. However, the procedure in the meeting is different. Rather than simply reading the program or using error checklists, the participants “play computer.” The person designated as the tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs (and expected outputs) for the program or module. During the meeting, each test case is mentally executed. That is, the test data are walked through the logic of the program. The state of the program (i.e., the values of the variables) is monitored on paper or whiteboard.

Of course, the test cases must be simple in nature and few in number, because people execute programs at a rate that is many orders of magnitude slower than a machine. Hence, the test cases themselves do not play a critical role; rather, they serve as a vehicle for getting started and for questioning the programmer about his or her logic and assumptions. In most walkthroughs, more errors are found during the process of questioning the programmer than are found directly by the test cases themselves.

As in the inspection, the attitude of the participants is critical. Comments should be directed toward the program rather than the programmer. In other words, errors are not viewed as weaknesses in the person who committed them. Rather, they are viewed as being inherent in the difficulty of the program development.

The walkthrough should have a follow-up process similar to that described for the inspection process. Also, the side effects observed from inspections (identification of error-prone sections and education in errors, style, and techniques) also apply to the walkthrough process.