♦ A Software Fault Prevention Approach in Coding and Root Cause Analysis

Weider D. Yu

The complexity of Lucent Technologies' advanced 5ESS® switching system makes the cost of finding, fixing, and retesting a software fault very high. The current 5ESS system contains several million lines of source code, which provide many complicated real-time switching function features. As customer demands for ever-increasing product quality compound the high cost of testing and reworking source code, it is crucial for the 5ESS Switching Development organization to find ways to prevent faults from being introduced into the software in the first place. The 5ESS Switch Coding Fault Prevention Team was assembled to find methods to prevent the most frequent faults from being injected into a product during coding. The Coding Fault Prevention Guidelines, *developed by the team for use in various Lucent switching development organizations, lists the most frequent errors made during coding. It also provides coders with information that will help reduce the risk of introducing faults into the software. This paper describes the most common preventable faults and the technical guidelines developed to overcome them. It also explains the metrics used to evaluate the results achieved.*

Introduction

Lucent's advanced 5ESS® switching system is complex, making the cost of finding, fixing, and retesting a software fault expensive. The current 5ESS switching software system contains several million lines of source code that provide many complicated real-time switching features. Today, hundreds of scientists and engineers are working continually on the 5ESS switch software to develop and enhance features that will advance cellular, voice, and data communications technologies.

Customer demands for ever-increasing product quality and the high cost of testing and reworking source code made it crucial for the Lucent 5ESS Switching Development organization to find ways to prevent faults from being introduced into the software in the first place.[1-4] The goal of fault prevention is to help programmers avoid injecting the most frequent faults into a product. C language, the predominant programming language used in the 5ESS switch development environment, strongly influences fault prevention efforts.

The 5ESS Switch Coding Fault Prevention Team, led by the author, began an effort in this area in 1993. The team's work resulted in significant achievements, leading to its selection by the Switching Systems Business Unit to represent the organization in receiving the "1994 AT&T Network Systems Quality Team Excellence Award." This paper introduces the coding fault prevention process and the technical guidelines used to prevent coding faults. It also explains the results achieved and the metrics used to measure them.

Development Process Analysis

The team completed a study of the 5ESS switch software fault removal effectiveness and fault flow by conducting interviews with developers on the nature of each of more than 600 faults. Performing an exten-

Where faults were introduced (baseline)		
Phase	Injected fault density*	Total (%)
Requirement	1.2	12.2
Data design	0.7	6.6
High-level design	1.9	18.5
Low-level design	1.5	15.3
Coding	4.7	47.4
Total	10.0	100.0

KNCSL: Thousand noncommentary source lines
*Injected fault density: Number of injected faults per KNCSL (normalized)

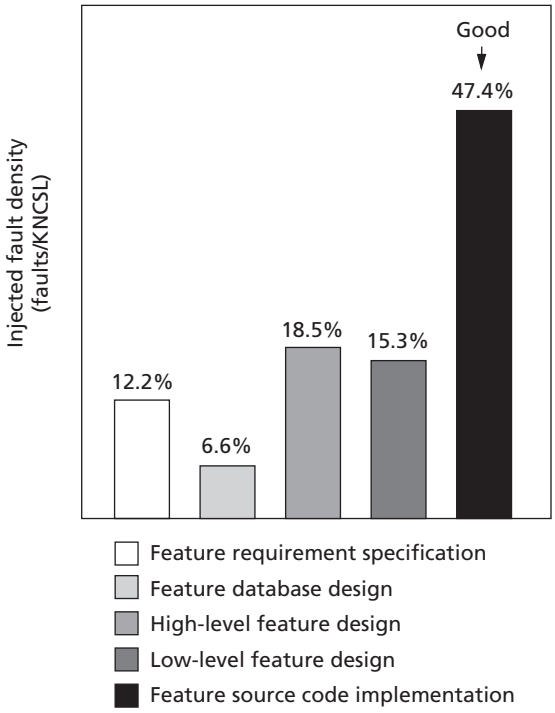


Figure 1.
***Baseline results of release* T.**

sive root cause analysis of the faults found and fixed in the past 5ESS switch software releases enabled the team to establish a baseline (called release *T*) of the software faults' life cycle—that is, how the faults were injected. **Figure 1** shows the baseline results established from the study.

A crucial finding determined from the analysis was that nearly half the faults were coding faults, and the majority of them could have been prevented. The team closely examined these faults to understand the types of coding problems that existed. **Table I** shows a list of the major coding faults found. The results of the analysis also showed that three types of faults—logic, interface, and maintainability—account for more than 50% of the total coding faults.[5]

The faults discussed in this paper were actual faults found in a very large software project environment. The average programming ability and experience of developers connected with the project were high compared to the norm in the industry. The team extensively interviewed developers who produced the code and the faults found in it to validate the faults and their causes. To enumerate the most significant faults for the *Guidelines*, the team conducted a statistical categorization and analysis on the fault data.

Table I. Major coding faults.

Type of coding defect	Total (%)
Logic	19.8
Standards	19.8
Maintainability	17.9
Interface	14.3
Performance	8.4
Functionality	4.0
Human factors	3.5
Consistency	2.9
Data	2.4
Syntax	0.5
Other	6.6
Total	100

Using a structure developed from software quality perspectives to describe various categories of coding faults, the team developed the faults in each

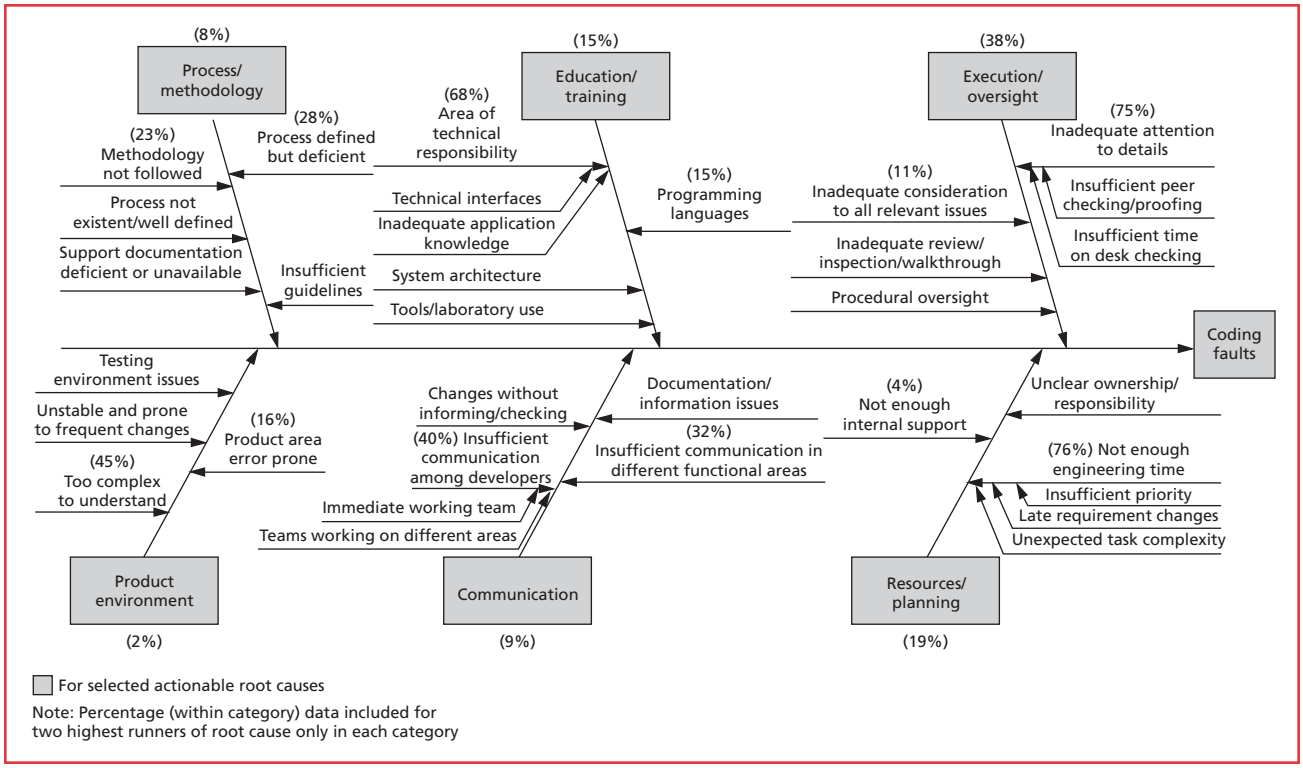


Figure 2.
Fishbone analysis used to identify root causes of the coding faults.

category based on actual error information and considerations of how program components were used. The team's approach to classifying faults is extensive and comprehensive.

The faults have been implemented in a software tool within the production environment. For each fault fixed, developers choose a fault type and one or more root causes. Based on the fault data collected from developers, 90% of the total actual faults were covered.

Faults reported in the literature are sometimes generalized to a higher level with brief descriptions, making the information difficult for developers to fully understand and apply in a software development environment. From a fault prevention point of view, developers must be encouraged to think about how to detect coding faults automatically. Some faults—such as functions without return value checks, function declarations and calls with incorrect numbers or types of arguments, variables declared but not used, and unreachable program statements—can be automatically detected using the "lint" tool. This tool detects features of C program files that are likely to be bugs, nonportable, or wasteful and then issues error warning messages.

Root Cause Analysis

The team used a fishbone analysis to identify the root causes of the coding faults. The top three root causes of the faults were execution/oversight (38%), resource/planning (19%), and education and training (15%).

In addition, for each of the three major root causes, the top two actionable detailed causes that had the greatest effect were:

- *Execution/oversight*. Inadequate attention to details (75%) and inadequate considerations to all relevant issues (11%);
- *Resource/planning*. Not enough engineer time (76%) and not enough internal support (4%); and
- *Education and training*. Area of technical responsibility (68%) and programming language usage (15%).

The complete fishbone diagram shown in **Figure 2**

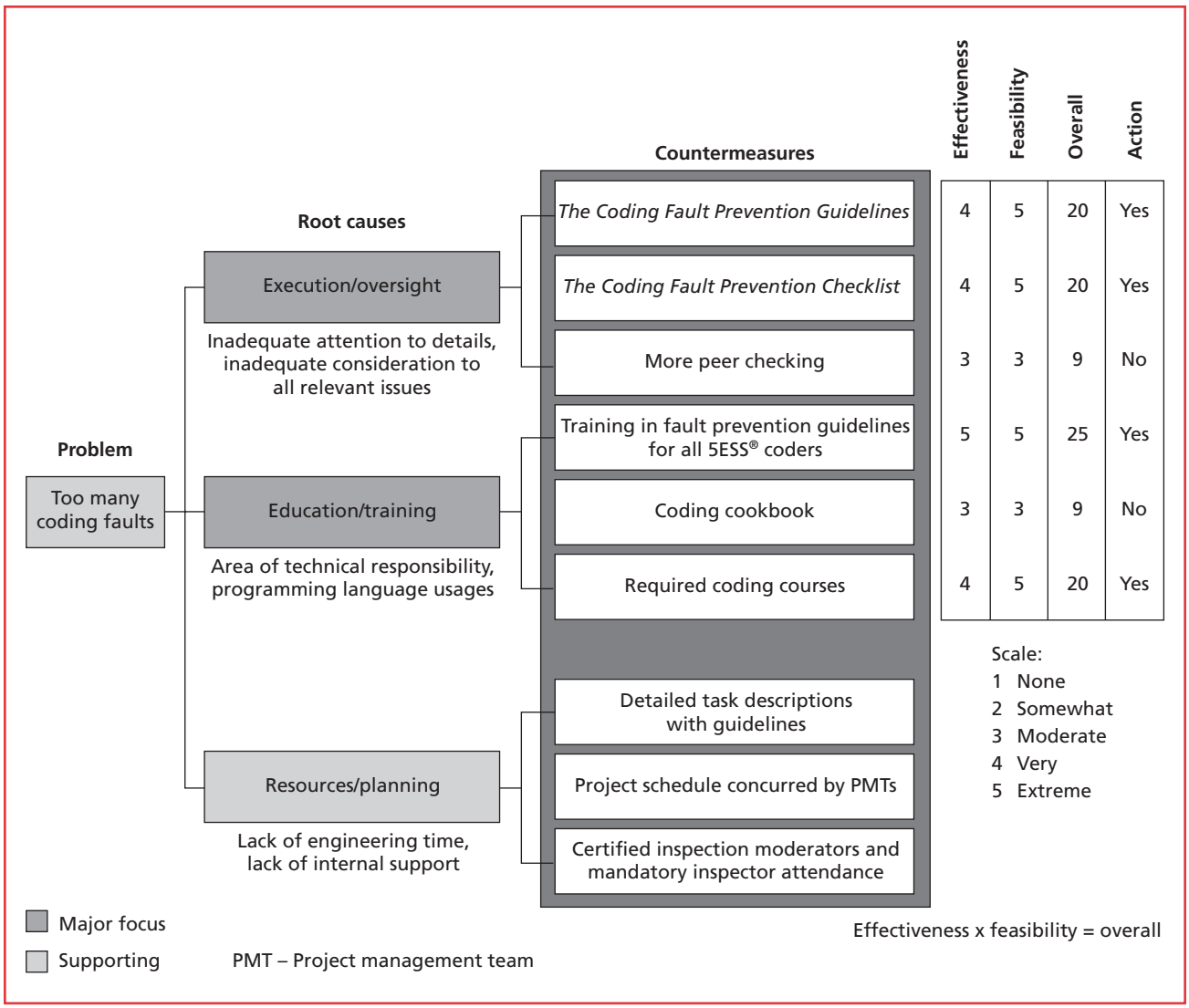


Figure 3.
Root causes and countermeasures for preventing coding faults.

illustrates the root causes identified and the percentages of distribution of their corresponding actionable causes.

Countermeasures for Improvement

Based on these root causes, the team identified several solutions, or countermeasures, and then rated each countermeasure numerically using two process characteristics: effectiveness and feasibility. The overall score for each countermeasure was calculated as the product of the values of effectiveness and feasibility. **Figure 3** shows a systematic diagram used to derive the most effective and feasible countermeasures.

To deal with the identified major root causes—oversight and lack of education and training—the team developed the *Coding Fault Prevention Guidelines* and a checklist for 5ESS switch developers. (Because the team could not control resource/planning, its root cause was addressed by project management.) The *Guidelines* describe, in detail, the faults frequently introduced by engineers, along with side-by-side examples of actual errors and the corrected code. They also provide engineers with information that helps reduce the risk of making those frequent faults. Engineers were asked to understand the *Guidelines* before they began their coding tasks.

In addition, to enhance the code inspection effectiveness the team developed *The Coding Fault Inspection Checklist*. Both the *Guidelines* and the *Checklist* were made a formal part of the coding process. The team then trained hundreds of engineers responsible for coding tasks in their use. All the countermeasures have been standardized in the development organizations.

The Coding Fault Prevention Guidelines

The *Guidelines* provide detailed explanations of each kind of error, accompanied by examples of correct and incorrect code. Coders and code inspectors then use these guidelines during the coding phase, preferably before coding begins. Periodically, the team analyzes root causes for new frequent coding errors. Because the most frequently encountered coding faults change over time, the team uses the results to update and maintain the *Guidelines*.

Logic Faults

Logic errors arise when a fault occurs in the correctness and consistency of computational and control logic of the software programs in C. This type of error relates to logical decisions or flows and branches within a program. To further reduce the testing effort, the logical correctness of the code needs to be carefully verified during coding.

List of Major Logic Faults

The analysis gave the team a better understanding of the types of logic faults the 5ESS Switching Development engineers made easily and frequently. The team used a tag number to uniquely identify each type of logic fault. **Table II** lists the tag numbers for these faults, along with the corrections for them. The data on the logic faults has helped the team focus on the most "critical" ones.

Samples of Logical Faults

The sections that follow describe each logic fault listed in *The Coding Fault Prevention Guidelines*, accompanied by samples of correct and incorrect code.

C operator associativity and precedence. Each C operator has its own order of precedence in an expression. It is the programmer's responsibility to be familiar with both the order of precedence of C operators and their order of evaluation (or associativity) when multiple operations are performed in a single expression or on a single line of code. Each of the following *incorrect* examples has occurred in the 5ESS switch code.

Table II. Logic faults.

Tag number	Fault correction
L1	Initialize all variables before use
L2	Control flow of `break` and `continue` statements
L3	Check C operator associativity and precedence for correct usage
L4	Ensure loop boundaries are correct
L5	Do not over-index arrays
L6	Ensure value of variables is not truncated
L7	Reference pointer variables correctly
L8	Check pointer increments/decrements
L9	Ensure logical OR and AND tests are correct
L10	Use all assignment and equal operators as intended
L11	Ensure bit field data types are either `unsigned` or `enum`
L12	Use logical AND and mask operators as intended
L13	Check preprocessor conditionals
L14	Check comment delimiters
L15	Test unsigned variables for `== 0` or `!= 0` only
L16	Use 5ESS® switch-defined variables in the correct context
L17	Use cast cautiously

A. The intent of this test was to mask `blkptr->rpthead.fltdesc` with `HWMFLTCLAS` and then compare the results of that masking operation with `HWMATEFLT`.

Incorrect:

```
if (blkptr->rpthead.fltdesc &
  HWMFLTCLAS == HWMATEFLT)
```

Correct:

```
if ( (blkptr->rpthead.fltdesc &
  HWMFLTCLAS) == HWMATEFLT)
```

However, since the == operator has higher precedence than the `&` operator, `HWMFLTCLAS` is compared with `HWMATEFLT` first and then the resulting true/false (1 or 0) is bit-wise masked with `blkptr->rpthead.fltdesc`.

B. Another common precedence error occurs when a pointer is passed to a function and the called function attempts to increment or decrement the contents of what that pointer is indicating. In this example, `numretry` is a pointer to a variable that has been passed to this function as an argument.

Incorrect:

```
*numretry++;
```

Correct:

```
(*numretry)++;
```

Here, even though both ++ and * operators have the same precedence, they associate right to left. First the increment is applied to the pointer and then the * operator is applied. This procedure makes no sense, because the contents of that variable have not been assigned. To fix this bug, a set of `()` are needed to force the increment to take place on the contents indicated by the pointer, as shown above.

C. Another common precedence error occurs when an attempt is made to call a function, save its return value, and test it from within a conditional expression.

Incorrect:

```
if( (rtc = _ims_open(NPRD_CH) !=
  _SUCCESS) ) {
```

Correct:

```
if( (rtc = _ims_open(NPRD_CH)) !=
  _SUCCESS ) {
```

Since the `!=` operator has higher precedence than the = operator, the return value of `_ims_open()` is compared with `_SUCCESS` first, and then the resulting true/false value is assigned to the `rtc` variable. Any subsequent tests made on `rtc` to identify the specific error code returned by `_ims_open()` proved fruitless, because the actual return value was lost and `rtc` only contained a true/false value. These errors are common when complex expressions are used.

D. In this example, the order of the evaluation of the `idx` variable is unclear and confusing at best.

Incorrect:

```
for (idx = 0; idx < 40;
  dispstring[idx] =
  COTsuccess[idx++]);
```

Correct:

```
for (idx = 0; idx < 40;
  idx++) {dispstring[idx] =
  COTsuccess[idx];
  }
```

Because some compilers evaluate from right to left and others evaluate from left to right, `idx` could be off by one value, depending on the application. And at any rate, the code is confusing; it would be clearer if it were rewritten to allow only the increment of `idx++` to take place in the `for` statement, with the assignment taking place within the body of the `for` loop itself, as illustrated above.

E. Although this example may look like a perfect piece of code, a problem stems from a faulty definition of the `GLCLRSHORT()` macro, which violates C coding standards.

Incorrect:

```
GLCLRSHORT (worklist[indx].unblocked,
  picb^1 );
```

This macro, which is defined as:

```
#define GLCLRSHORT(map_name, bitno) \
  (map_name[bitno >> GLL2SHORT]
  &= (1 << (bitno & (GLSZSHORT-1))))
```

expands in the code to:

```
( worklist[indx].unblocked[ picb^1
  >> GLL2SHORT ] &= ( 1 << (picb^1 &
  (GLSZSHORT-1)) ) );
```

Two orders of precedence errors are introduced when an expression containing the exclusive OR operator ^ is passed for the second argument of the macro. Because both the >> and the & operators have higher precedence than the ^ operator, the logic of the macro is altered.

Kernighan and Ritchie[6] give an excellent table that shows the order of precedence and the associativity of each C operator. Programmers can refer to this table during programming to determine how the operators interact with each other.

Variable initialization. Programmers should verify that all variables, both global and local, are initialized before they are used. Members of globally allocated structures and all pointers must be properly initialized before use.

Rules governing the initialization of variables are as follows:

- Local and global constants must be initialized where they are defined.
- Nonconstant, nonstatic local variables must be initialized in the code segment rather than where they are defined.
- Process-specific global variables must be initialized by specific functions designed for their initialization whenever those processes are recreated.

Loop boundaries. Variables are sometimes incremented beyond their designed capabilities, especially in such practices as indexing an array by more values than that array is designed to hold. This practice can cause wild writes on other program text and stack space. It is advantageous, therefore, to verify that the bounds or limits of each loop statement (`for` and `while`) are correct.

A. In this example, although the correct number of elements was allocated to the local array `proname[ ]`, it was eventually over-indexed because the loop limit was set too high.

Incorrect:

```
char proname[ 10];
          .
          .
          .
☞   for(i=0;i<14;i++)
  {
        proname[ i]  = *argv[ 0];
          .
          .
          .
```

B. Another error occurs when loop boundary tests continue too far. In this example, the variable `featent.curpos` should only be tested for less than, never equal to, `NBISTRMFTENT`.

Incorrect:

```
for( ; (featent.curpos <=
  NBISTRMFTENT &&...
```

Correct:

```
for( ; (featent.curpos <
  NBISTRMFTENT && ...
```

To avoid such subtle errors, all loop boundary tests should be carefully examined.

Variable truncation. If a value larger than itself is assigned to a variable, that value becomes truncated. Although not detrimental in some cases, this practice is devastating in others, and should be avoided. Special care must be taken when bits are being assigned to fields, because it is very easy to assign a value that uses more bits than a field can accommodate. This is especially true if the bit fields exist in different data structures, where their respective structures may be defined for different sizes.

A. This example is taken from a field bug that required a software update (that is, a software patch/change) to rectify:

```
pcccap->cindex = (cindx + 1) & 0x7f;
```

Here, the value being assigned is masked with `0x7f`, which is 7 bits. Because `cindex` was only allocated 6 bits of storage in its data structure, the data being assigned was truncated, causing the critical field problem.

B. This example has also required a software update to rectify it.

Incorrect:

```
char time;
time = SMLI2PDEL;
OSWAIT(OSTIMEOUT, time);
```

Correct:

```
long time;
time = SMLI2PDEL;
OSWAIT(OSTIMEOUT, time);
```

Here, `SMLI2PDEL` was `#define` to 10000 and then passed as the second argument to `OSWAIT()` to wait for 10 seconds. However, since `time` was only allocated as a signed `char`, the assigned value of `SMLI2PDEL` was truncated to 16 decimals. Thus, this code was only waiting for 16 milliseconds, instead of the required 10 seconds.

Hints: Never take any assigned variables for granted. Always check that the value being assigned can never be larger than the variable or data structure can accommodate.

Pointer increment rules. Using pointers provides notational convenience and program efficiency, resulting in code that uses less memory and executes faster. Pointer increments and decrements are scaled to the size of the data type to which the pointer is pointing.

When incrementing and decrementing pointers, do not exceed the boundary of the data type. The example below illustrates a pointer being decremented and incremented to out-of-bounds values:

Incorrect:

```
short value[ EXMAXVAL] ;
short *value_pointer;

value_pointer = value;
☞      --value_pointer;

value_pointer = value;
☞      value_pointer += (EXMAXVAL + 1);
```

Logical OR and AND tests. Any logical OR of two or more "not equal" (!=) tests of the same variable will always produce true results. Likewise, any logical AND of two or more "equality" (==) tests of the same variable will always produce false results.

A. In the incorrect example below, the tests of `CRMTCTYP(cid)` will always produce true results.

   Incorrect:

   ```
   if ((call.dn[ i] == CRINIT) &&
       ((CRMTCTYP(cid) != CRINWATS) ||
       (CRMTCTYP(cid) != CRTINWATS)))
   ```

   Correct:

   ```
   if ((call.dn[ i] == CRINIT) &&
       ((CRMTCTYP(cid) != CRINWATS) &&
       (CRMTCTYP(cid) != CRTINWATS)))
   ```

B. This incorrect example will always produce false results, regardless of the value of `rc`.

   Incorrect:

   ```
   if (rc == GLFAIL && rc == DBSYS_ERR)
   ```

   Correct:

   ```
   if (rc == GLFAIL || rc == DBSYS_ERR)
   ```

Logical AND and mask operators. One method of giving up control to allow other processes a chance to run while executing within a loop is by setting a mask value and checking it against a loop counter with each iteration of a loop. However, if a double `&` is used, no mask occurs; instead, the program performs a true/false operation, and the loop may run for a much longer time period before it takes a break. This is an example taken from within the body of a `for` loop.

Incorrect:

```
if (( i && 0xff) == 0) OSSUSPEND();
```

Correct:

```
if (( i & 0xff) == 0) OSSUSPEND();
```

Here, the variable `i` was declared to be a `short`. The programmer intended to have the loop give up control every 256th iteration, that is, whenever the lower 8 bits of `i` were set. However, because the programmer used `&&` instead of `&`, this statement would only produce true results once every 65,536 times.

Preprocessor conditionals. The `#if` and `#ifdef` preprocessor statements are used for conditional compilation of programs. The `#if` statement tests whether a constant expression produces a non-zero value, and the `#ifdef` statement evaluates whether a single identifier has been defined previously (and has not been subsequently undefined by a `#undef` statement), either by using a `#define` statement or being passed as a `-Didentifier` option to the compiler.

Note: If an identifier is not currently defined, the `#ifndef` statement is used to evaluate it.

A. The only valid use of a `#ifdef`, `#ifndef`, or `#undef` statement is for checking a single identifier:

   ```
   #ifdef  M68000  #ifndef EES  #undef SPTAG
           .           .
           .           .
           .           .
   #endif          #endif
   ```

   The `#undef` statement does not require a `#endif` statement. In addition, it is illegal syntax for a `#ifdef` or a `#ifndef` statement to have parentheses—such as `#ifdef (PAM)`—surrounding the identifier.

B. Since a `#if` statement is used to test a constant expression, it can test for more than a single identifier, such as any of the following:

   ```
   #if PAM      ☞ This is equivalent to a #ifdef PAM
   #if (defined PAM)
   #if (defined PAM && (!defined EES))
   ```

Flow of `break` and `continue` statements. Sometimes, `break` and `continue` statements lead to confusion about exactly what has happened, that is, where control has been passed. It is advantageous, therefore, to verify the proper use of these control-changing statements.

A `break` statement exits from within a loop statement (`do`, `for`, `while`) or from within a `switch` statement. Confusion sometimes arises when one or more loops and/or `switch` statements are nested. If a `break` is executed from within a set of nested loops, only the innermost loop in which the `break` statement occurs is terminated. Likewise, if a `break` is executed from within a set of nested `switch` statements, only the innermost `switch` statement is exited. When both loops and `switch` statements are nested, be sure the action executed by the `break` statement is what is desired.

The `continue` statement skips remaining code in the innermost loop from which the `continue` is executed and continues executing from the end of that loop. This ensures that any increments, such as the third expression of a `for` statement, will take place. If a `continue` is executed from within a `switch` that is itself within a loop, the `switch` is exited, passing control to the end of that loop and skipping any statements in between. This is in contrast to a `break` statement, which would only exit that particular `switch` statement and continue executing the statements that follow the end of that `switch` in the loop.

Whenever a `break` or `continue` statement is used within a loop statement, the programmer must clearly annotate how the change will affect the program flow. In this example the `continue` passes control to line 20, where `value` is incremented (as expression 3 of the `for` statement) and then tested (as expression 2 of the `for` statement). When the `break` statement is executed, control is passed to line 19.

```
11: for (value = 0; value < EXMXVAL; value++ ) {
12:       switch (get_value(value)) {
13:       case EXNOTUSED:
14:            continue;
15:       default:
16:            doit();
17:            break;
18:       }
19:       funcA();
20: }
21: other_work();
```

Interface Faults

Just what is an interface? Among the many definitions offered are the following:

- A shared boundary across which information is passed;
- A hardware or software component that connects two or more other components for the purpose of passing information from one to the other;
- A connection between two or more components for the purpose of passing information from one to the other; and
- A connecting or connected component, as in the second item above.

Therefore, interface problems/errors result when a mismatch occurs in a data/control transfer between a function and its surrounding environment, other functions, global/local variables, or data variables/structures, including:

- A variable/symbol name mismatch, such as a reference to a nonexistent variable;
- A structure, type, or configuration mismatch, such as:
  - An argument type mismatch between calling statement and called function,
  - One or more required arguments missing, or
  - Extraneous arguments passed;
- A value or variable range mismatch, such as a wrong value substitution for an argument; or
- A data transfer or control procedure mismatch, such as a failure to set the program state.

List of Major Interface Faults

It is sometimes difficult for engineers to avoid creating interface faults, because they are easily overlooked during code inspection and testing. Understanding what the major interface faults are and designing and writing code that excludes them is the most effective way to prevent them. **Table III** lists each fault type and describes its correction.

Samples of Interface Faults Listed in *The Coding Fault Prevention Guidelines*

The sections that follow describe some of the interface faults and how to avoid them.

Function arguments and return types. Developers should analyze every function call to determine whether the needs of the called and calling functions have been met, based on the arguments

Table III. Major interface faults.

Fault type	Fault correction
I1	Verify that the arguments and return types of all function calls are consistent with those declared in the function.
I2	Complete function and global variable definitions and declarations.
I3	Use a reasonable number of function arguments.
I4	Verify macros' use (for example, full context).
I5	Follow naming standards for function arguments and global variables.
I6	Use defensive programming practices.
I7	Ensure system process entry points are correct.
I8	Use function return values or cast to `void`.
I9	Verify parentheses are placed on function calls.
I10	Use subsystem provided initialization functions where possible.
I11	Use library functions or pre-existing functions where possible.

passed and the return types used, and what the function call will provide for their programs. Questions such as the following may stimulate thinking along these lines.

- Is this function providing the necessary functionality?
- Will the parameters passed provide sufficient data to achieve that functionality?
- Would it be more efficient to pass a pointer rather than a group of variables or, if those values are to be modified, a data structure?
- Will the return value provide information that the calling function can readily use?

For every function call made, developers need to verify that the arguments' types and order match those of the called function. They should also verify that the return type is correct. If the return value is not used, it should be cast to `void`. If a function does not have a return value, define the function to return `void`; otherwise it will default to returning `integer`. Function arguments of the wrong type should be cast to the expected type when appropriate.

A. A common error that occurs in many function calls is passing a copy of a variable when the function definition expects a pointer, or passing a pointer when the function definition expects a value. In the following example, `MLtisfail()` is defined to accept a pointer to the data being passed as argument one, but no `&` was coded.

   Incorrect:

```
return(MLtisfail(mltmsg->cmd.req.req2,
  tmrc));
```

   Correct:

```
return(MLtisfail(&mltmsg->cmd.req.req2,
  tmrc));
```

   It is helpful to use pointers as arguments when:

   – Large amounts of data or large data structures are being passed, and
   – A called function must modify the calling function's data.

B. Often one level of indirection is missing, such as the asterisk in the example below. The argument should have been cast as a pointer to a pointer.

   Incorrect:

```
DXal_fndnxt( ..., (DXALMDATA *)
  &data_ptr );
```

   Correct:

```
DXal_fndnxt( ..., (DXALMDATA **)
  &data_ptr );
```

   It is helpful to use a pointer to a pointer when it is necessary for a called function to modify the pointer to the calling function's data (that is, to modify the address where the data is stored or point to new data).

C. A common error occurs when the data types of arguments are not verified before calling the function. In this example the first parameter expected by `CCbinasc()` is an `unsigned`

long, but loop_ptr->annc_rtidx is an unsigned short. The parameters that CCbinasc() receives will be wrong because the difference in data types causes a skew. CCbinasc() will take the first four bytes of data as its first parameter. Only two bytes of data will be taken from loop_ptr->annc_rtidx; the next two bytes will be taken from the next argument.

Incorrect:

```
(void) CCbinasc( loop_ptr->
 annc_rtidx, ... );
```

Correct:

```
(void) CCbinasc( (DMUNLONG) loop_ptr->
 annc_rtidx, ... );
```

D. Another common error occurs when an additional argument is defined for a function but not every occurrence of the call is modified. (An error of this type can also arise when a function is called but not all its arguments are passed.) This problem becomes more severe when a new argument is added at the beginning or the middle of the existing argument list. When this occurs, all the previously existing arguments from that point on are "skewed." The example below illustrates just such a problem.

Function called with three arguments

```
☞ DBnswch_mem(&omsg, &smtimr,
      grwsize);
```

but defined with four arguments

```
☞ DBnswch_mem(OSPID, *omsg, *timr,
      grwsize)
```

E. Another common error arises from misunderstandings between the DMPORT and the DMGPORT data types. The former is an unsigned 16-bit quantity, but the latter is a typedef struct containing both a member and a module. In this example newtup_ptr->port is pointing to a DMGPORT. What is needed is to pass the member of the port structure being pointed to by newtup_ptr.

Incorrect:

```
RTdslprmpt(newtup_ptr->port);
```

Correct:

```
RTdslprmpt(newtup_ptr->port.member);
```

Other similar, but not identical, errors occur for other data structures.

F. Another group of errors that could be classified under this subheading results from an improper number of arguments being passed to printf() and its related function types. These problems arise when more arguments are passed to be printed than there are conversion characters, or when there are more conversion characters than arguments passed.

Incorrect:

```
sprintf(infile,
  "/rclog/tmp.evl%d.5E31");
```

Incorrect:

```
sprintf(corclog, "/log/cpcorc", 0);
```

G. Many errors result from confusion between fprintf() and printf(), because the former requires a stream pointer for argument number one and the latter does not.

Incorrect:

```
fprintf("system error, 'cdbcom' not
  found in relation table\en");
```

Incorrect:

```
printf(stdout,"%s VPATH:\en",Prompt);
```

Function and global variable definitions and declarations. When a program calls a function or accesses global data, it needs a declaration of the function's return type and the global data type. Usually, these declarations are part of header files and must be defined according to the scope of the function calls and/or data accessed. Declarations of functions must be provided as function prototypes, also known as function templates. A *function prototype* lists the types of each function's arguments, as well as the type of argument the function returns. The use of these function prototypes and header files for data definitions ensures the exact definition is given for the called function and/or the data being accessed.

If the called function and/or data accessed is defined and used only in a specific module, no special header files are needed. If the called function and/or data accessed is defined in a specific module and used in more than one module within that subsystem, then the declarations of those types (as well as the definitions of the types used) must be made in the subsystem's local header files.

If the called function and/or data accessed is used by subsystems other than the one where they are defined, then the declarations of those types (as well as the definitions of the types used) must be made in global header files.

Function arguments. In interfaces, simplicity is a desirable property, because the number of arguments within a function affects a program's interface complexity. A large number of arguments makes the interface unmanageable, and the result can be difficult to read and understand. The maintainability of the interface also suffers, because it becomes easy to insert errors, such as argument type mismatches, when modifying an argument. The "rule of seven" may be used as a reasonable maximum number of arguments in a function call.

Some interface measures help manage the use of function arguments, such as:

- The number of variable items per function (in + out), and
- The number of functions to which it is connected (called and calling).

Passing data in structures (`struct` or `typedef struct`) that contain all or most items required by the called function is a powerful method of simplifying an interface.

This example illustrates a function with too many arguments. It can be rewritten to pass a structure and/or pointers.

Original:

```
SSfunction (code, value, type, DBYES,
  msg.numb, msg.size, msg.unit.side,
  msg.unit.mod, msg.reg_make, DBNO);
```

Pass pointer:

```
SSfunction (code, value, type, DBYES,
  &msg, DBNO);
```

Pass structure:

```
SSfunction (code, value, type, DBYES,
  msg, DBNO);
```

If a pointer to a structure is passed, then the called function can change the values of that structure's elements. If an entire structure is passed, and it contains more elements than the summation of the individual arguments passed, then the programmer must avoid introducing stack overflow problems.

Function returns. Although C language does not require explicit definitions or declarations of function return types, functions that do not return values should be defined and declared as type `void`. When a function is defined or declared without an explicit return type, the compiler defaults the return type to `int`. In this case, however, it becomes difficult to determine whether the function will return a value. To avoid this confusion, the function's return type should always be defined and declared.

Using the `void` data type informs the compiler that the function will not return a value, preventing any further ambiguous use of the function and improving the program's maintainability. Because the compiler does not generate object code for returning a value, it also saves stack space and helps system performance. A function defined or declared to be type `void` cannot be used in an expression.

Items returned from functions should either be the same type as the function or they should be cast to the function type. If a function returns a value that does not need to be checked or saved, the function's return can be cast to `(void)`, which also saves stack space.

A. Functions defined without an explicit type default to `int`:

Incorrect:

```
☞
SSfunction()
{
        return;
}
```

Correct:

```
void
SSfunction()
{
        return;
}
```

B. In this example, although the function `SSmodify()` is declared to return type `void`, its return value is checked:

Incorrect:

```
void
SSfunction()
{
      void SSmodify();
☞     if (SSmodify() == GLSUCCESS)
                .
                .
                .
```

C. When a function that returns a value is called, but there is no need to save or check its return value, it should be cast to type `void`:

Incorrect:

```
RET_VAL
SSfunction()
{
     strncpy(cost, Glctcst, 4);
                .
                .
                .
```

Correct:

```
RET_VAL
SSfunction()
{
     (void)strncpy(cost, Glctcst, 4);
                .
                .
                .
```

D. If a function defined to return `void` instead returns a value, it will not compile.

Incorrect:

```
void
SSbad_func()
{
☞      return(0);
}
```

Defensive programming practices. When a function is called, the program must verify that the arguments passed are within the expected ranges. This practice is helpful for preventing indiscriminate use of pointers whose values may not be guaranteed to be sane. The level of defensive checking must be balanced with real-time constraints. For example, it is not absolutely necessary to check and recheck the same arguments as they are passed down to lower functions. If the arguments are checked in the first function, there will probably be no need to recheck them at each function called unless they have been modified in some way.

A. Before using global pointers, programmers should verify the sanity of the pointers.

Correct:

```
    /* front and rear pointer range checks */
☞   if (CRrptr < CRQPTRESET || CRrptr > CRMAX
      ||
☞     || CRfptr < CRQFPTINIT CRfptr > CRMAX) {
      CRqrecover ();  /* recover the CRA queue
        */
      AUASRTA (AUFALSE, CRA_4_ASRT);
      return (CRERROR);
      }
```

B. When a function is entered, the program must verify the sanity of the arguments.

Correct:

```
Short
DBopgparm( parid, new_buf, pcrid,
  operation)
DMSYSID parid;
char *new_buf;
unsigned char pcrid;
DMDBMODE operation; /* Global
  parameter operation */
{
                .
                .
                .
        <<< local declarations >>>
                .
                .
                .
       /* range check parameter id */
☞    if ((parid <= 0) || (parid
         >= dbparmax)) {
             return(DBSYS_ERR);
       }
```

Maintainability Faults

The *maintainability* of a software program represents the ease with which it is possible to maintain the software on that system. Basically, maintainability measures how simple it is to correct and change software programs. Maintainability can be defined as:

- The ease with which software programs can be maintained,
- The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements, and
- The ease with which a software error can be located and fixed within a specified time period.

Table IV. Major maintainability faults.

Fault number	Fault correction
M1	Only use macros to make code easier to read and/or more flexible.
M2	Give macros containing arguments descriptive argument names.
M3	Do not hide important details, significant operations, or side effects in macros.
M4	Avoid code with program knots, such as inappropriate use of `goto` or `break` in loops.
M5	Use structure bit fields to avoid shift/mask instructions to access bit fields.
M6	Include any definitions used in all global/local header files (`#include`).
M7	Use `if-else` series instead of a series of `if` statements when deciding between a series of mutually exclusive possibilities.
M8	Use `#define` constants instead of hard-coded numbers.
M9	Verify that parentheses are used to clarify and ensure correct precedence.
M10	Enclose all bodies of flow-control statements in braces.
M11	Always use previously defined `struct/union/typedef` types for arguments.
M12	Use existing `#feature` statements when possible.
M13	Follow the standard layout for source files and functions.
M14	Use library functions rather than rewriting code.
M15	Use enumerations instead of constants when appropriate.
M16	Provide clear and meaningful comments.

Maintainability can have several components, including modularity, simplicity, self-descriptiveness, and verifiability.

The *modularity* of a software program is defined by a structure of highly cohesive components with optimum coupling. In a modular program, a change to one component has minimal impact on other components. *Simplicity* is the ability to make definitions and implementations of software functions noncomplex and understandable. A software program is *self descriptive* if it contains enough information for others to determine its objective, assumptions, constraints, inputs, outputs, components, and status. *Verifiability* is the ease with which the operation and performance of a specified software program can be checked.

List of Major Maintainability Faults

Maintainability faults affect how efficiently engineers can modify source programs. **Table IV** lists the major maintainability faults.

Samples of Maintainability Faults

The sections that follow describe corrections for some of the maintainability faults.

Flow-control statements. A software update may introduce a bug if it adds a line of code that does not enclose the body of an `if` statement in braces, thus inadvertently changing the logical progression of the code. Therefore, the bodies of all `if`, `else`, `for`, `while`, `do-while`, and `switch` constructs should always be defined within braces and indented a full 8-character tab stop. (All flow control statements must follow one of the permitted coding standards styles to prevent problems with later code changes and non-standard macro expansion that may contain multiple statements or `else-less-if` statements.)

A. In this example, an `if` statement body is defined both without and with braces:

Original code

Incorrect:

```
if (condition == TRUE)
  flag = DBYES;
```

Correct:

```
if (condition == TRUE) {
  flag = DBYES;
}
```

B. The programmer then inserts a software update (SU), adding a call to `ASfunction()` immediately following the setting of the flag.

Incorrect:

```
if (condition == TRUE)
  flag = DBYES;
SU added ☞  ASfunction();
```

Correct:

```
if (condition == TRUE) {
  flag = DBYES;
SU added ☞  ASfunction();
 }
```

As this example illustrates, when braces are not used in the original software, lines added to the program can easily disrupt its logical execution. The incorrect example above contains no braces, either existing or added. Instead of calling the `ASfunction()` in addition to setting the `flag`, the program calls the `ASfunction()` regardless of the condition being tested, with the `flag` only being set if the condition is true.

Explicit variable comparisons. All logic control statements must test for an explicit comparison. The programmer cannot assume that if the expression evaluates to a zero it will be taken as true, and anything else is assumed to be true. The example below illustrates the principle of explicitness:

Incorrect:

```
if (strlen(Name)) {
  printf("Emp Name %s\n", Name);
}
```

Correct:

```
if (strlen(Name) != 0) {
  printf("Emp Name %s\n", Name);
}
```

Embedded assignments and multiple statements. Only one variable or function may be declared per line. This allows the programmer to place a comment for each declaration.

For example:

Incorrect:

```
DBxyz()
 {
     short AMicrtmrs(), rc;
     short code, status = 0;
```

Correct:

```
DBxyz()
 {
     short AMicrtmrs();  /* set ICR timers */
     short rc;  /* return code */
     short code;  /* access code */
     short status = 0;  /* process status */
```

Preprocessor conditionals. Proper use of preprocessor conditionals can contribute greatly to the readability of programs, as shown in the following four examples.

A. Preprocessor conditionals used within function definitions should only be used for minor differences based on the definition of a flag. Major differences should be handled by separate functions.

B. Preprocessor conditional changes within a function should be localized as much as possible, not scattered throughout the function (that is, any decisions that depend on a conditional expression should be made in one place if possible.)

C. A preprocessor conditional must not split or interrupt the flow of a complete C statement.

Incorrect:

```
 if ((cond == GLRUN)
#ifdef FEAT_A
  || (cond == GLWAIT)
#endif
  ) {
```

Correct:

```
#ifdef FEAT_A
  if (cond == GLRUN || cond == GLWAIT)
#else
  if (cond == GLRUN)
#endif
{
```

D. Preprocessor conditional code can often be made more readable by isolating it within macro definitions.

Incorrect:

```
  #ifdef EXPORT
    for ( i = 0; i < MSXRSMMAX; i++)
  #else
    for ( i = 0; i < MSRSMMAX; i++)
  #endif
```

Correct:

```
MSmax.h:
  #ifdef EXPORT
  #define MSMAXRSM MSXRSMMAX
  #else
  #define MSMAXRSM MSRSMMAX
  #endif
  Source File:
    for ( i = 0; i < MSMAXRSM; i++)
```

As this example illustrates, placing the preprocessor conditional in a header file and defining a `#define` constant according to whether that particular conditional is true makes the source code much more readable. This is especially true when the `#define` constant is used repeatedly in one or more source files.

Macros. A *macro* is a short piece of text, or text template, that can be expanded into a longer text. Often, problems in macro definitions do not appear where the macro is defined; instead, they cause errors much further down in the program. Using macros can simplify code, but they can also obscure important details or significant operations. Careful attention is needed to prevent unwanted and/or unexpected side effects.

A. For example:

```
#define SQUARE(x)   ((x) * (x))
                .
                .
                .
w = SQUARE (++value);
```

The invocation of the `SQUARE` macro will cause `value` to be incremented twice, because this statement will be expanded by the C preprocessor to:

```
w = ((++value) * (++value));
```

B. Defining macros to change flow control is considered a bad programming practice. This example obscures the basic control flow of the program.

Incorrect:

```
#define FOR_ALL for(i = 0; i < SIZE; i++)
                        .
                        .
                        .
/* Clear the c array */
FOR_ALL {
        c[ i]  = 0;
}
```

Null statement forms for `for` and `while` loops. The null statement forms of the `for` and `while` loops are emphasized by placing a comment on the line following the left brace. The comment should also state that this is a `/* Null Body Loop */`.

The examples below show incorrect and correct null loop statements.

Incorrect:

```
while (*s++ = *t++);
```

Correct:

```
while ( *s++ = *t++) {
        /* Null Body Loop */
}
```

Program knots. A *program knot* is a measure of unavoidable intersections of program control flow arcs, as well as a measure of the lack of structure in the linear program flow graphs. Inappropriate use of `goto`, `break`, `continue`, or nested loops can cause extra logical complexity in a program, which can be reflected by the number of "program knots." If the statements of the program can be rearranged to reduce the number of counts of "program knots," the structure of the program logic will be easier to understand and maintain.

A. A `break` statement in the `for` loop of this example has caused one "knot."

Incorrect:

```
for ( i = 0; i < n; ++i ) {
  eof_flag = fscanf (in_file, "%d;",
    &x[ i]);
  if ( eof_flag == EOF ) {
knot ☞      break;
  }
  sum += x[ i];
}
```

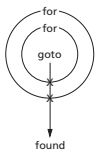
This program can be rewritten without any "knot," as follows:

Correct:

```
  eof_flag = 0;
     for ( i = 0; i < n && eof_flag !=
       EOF; ++i ) {
       eof_flag = fscanf (in_file,
         "%d", &x[ i]);
       if ( eof_flag != EOF ) {
               sum += x[ i];
       }
  }
```

B. The goto statement in the inner for loop causes a program knot of 2.

```
for (y = 0; y < Y_LIMIT; y++) {
  for (x = 0; x < X_LIMIT; x++) {
    if (data [ x] [ y] == 0) {
      goto found;
    }
  }
}
(void) printf ("Not found\n");
exit(8);
found: (void) printf ("Found at %d,
          %d\n", x,y);
```



Some similarities exist between the concept of "program knots," which reduce the complexity of programs, and "irreducible flow graphs," used to analyze signal flow graphs. A flow graph is irreducible if all nontrivial proper subflow graphs have been eliminated. An irreducible flow graph is one in which nodes and edges cannot be reduced further to maintain the functions of the original flow graph.

Each source program statement can be represented as a node in a flow graph, and each explicit control transfer between statements is represented as a direct connecting edge (or called branch) between nodes. The number of intersections is a measure of a program's "structuredness." It can also be used as a measure of program control complexity. The higher the number of program knots in a program, the greater the cost of maintaining the program and the higher the risk that the program's complexity will produce faults.

A flow graph[7] containing a limited number of nodes with edge cross intersections is more structured and more helpful for system analysis or general study. The concept of "irreducible flow graphs" emphasizes the property of minimum numbers of nodes and edges. Systematic methods are available to reduce any signal flow graph to an essential flow graph, which contains only essential nodes (in addition to sinks and sources) and eliminates the nodes with edge cross intersections. The irreducible flow graph concept can be viewed as a generalized scheme of the program knots concept.

Pointers in array function arguments. An array name used without a subscript is a pointer to that array. Specifying the array name without a subscript produces a pointer to the first element of the array. To declare a function parameter an array, use its name rather than a pointer to the array, as shown below.

Incorrect:

```
void
FCgetvalues( arrayname )
DMLONG *arrayname;
{
            .
            .
            .
```

Correct:

```
void
FCgetvalues( arrayname )
DMLONG arrayname [];
{
            .
            .
            .
```

Measurement Metrics

Metrics measure the results of preventing software faults using three criteria: the number of faults, costs of developing accurate code versus fixing faulty code, and the number of coding faults found by customers after delivery divided by the source code size.

Faults

The team defined the metric *injected coding fault density* as the number of faults injected into the software during coding divided by the source code size of the software in thousands of noncommentary source code lines (KNCSL).[8] The metric showed a 34.5% coding fault reduction in the release *T*+1. The 5ESS Switching Development organization exceeded its original 10% target value for improvement by 250%.

The metric *testing cost per source code line* is defined as the total testing cost of the product (in dollars) divided by the source code size of the software (in KNCSL). The metric showed an 18.3% reduction in the release *T*+1.

Costs

Each of the three metrics used to measure costs showed a reduction, as follows:

- The *software fixing cost per fault*, defined as the average software fixing cost (in dollars) per fault, was derived from the project's historical results. Based on the 34.5% coding fault reduction, a total of US$7M was saved in product rework and testing. The result was validated by the actual cost data collected.
- The *engineer cost per hour*, defined as the average cost (in dollars) per engineer per hour, was derived from the project's historical results.[8] Based on the total number of hours expended by engineers in implementing the chosen countermeasures, the total engineering cost was US$100K.
- The *cost per source code line*, defined as the total actual development cost of the software (in dollars) divided by the source code size (in KNCSL), showed a decrease for the release *T*+1.

Coding Faults Found by Customers

The *customer-found coding fault density*, defined as the number of coding faults found after delivery to external customers divided by the software's source code size (in KNCSL), showed a 35.2% reduction in the release *T*+1.

Results

The results of implementing the countermeasures were significant. The team compared the results quantitatively with the baseline release *T* data below:

- The number of coding faults injected for the *T*+1 software release was reduced by 34.5%. **Figure 4** shows these results exceeded the target for improvement in *T*+1 by 250%. The average total testing cost (including the engineering effort spent fixing and testing) was reduced by 18.3%.
  - Reductions in injected coding faults shortened the engineers' work interval. As a result, engineers were able to start new work earlier, thereby improving their productivity.
  - The reduction in coding faults gave product management the confidence to discontinue customer site testing, a 5ESS switch development phase, thereby eliminating a burden on external customers.
  - The result also contributed to the on-time delivery goal of the customer business unit (CBU). The development interval of the release was shorter than the release *T*, enabling the CBU to exceed its goal by 8.3%.
- A total saving of $7M in product rework and testing versus the implementation cost US$100K produced a benefit-to-cost ratio of 70:1. For the *T*+1 release, the average development cost was lower than that of the baseline release *T*. The implementation also contributed to the CBU's financial goal for development costs by surpassing its goal of a 17% decrease and achieving a 26% decrease.

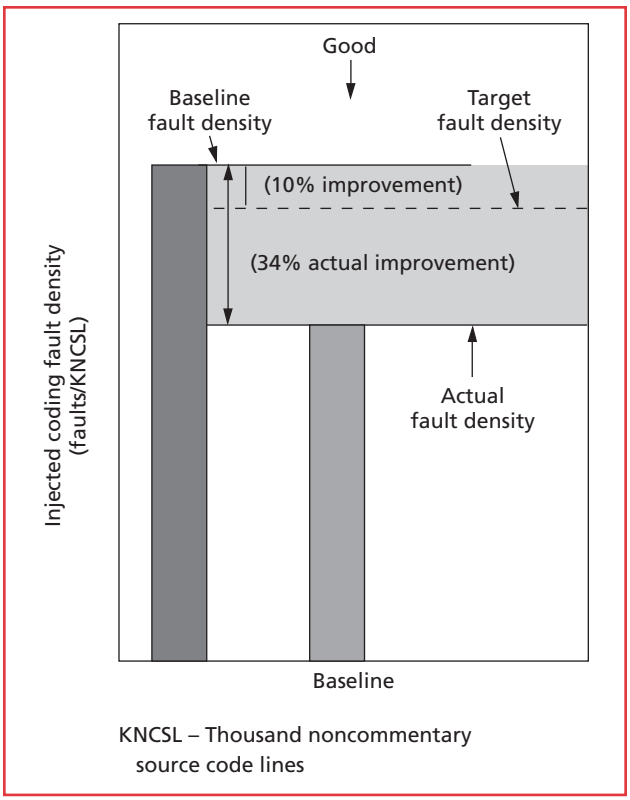


Figure 4.
Results of implementing the countermeasures.

Conclusions

The software release issued after the fault prevention approach and technical guidelines reported in this paper were instituted was one of the best produced by

the 5ESS Switching Development organization. The number of coding faults delivered to external customers was reduced by 35.2%, lowering the impact of the major selected actionable root causes by 60%. External customers continue to be satisfied with the superior performance of the 5ESS switch—the best in its class, according to the Federal Communications Commission's official report.[9]

Despite the fact that implementing coding fault prevention was new to the organization, the results achieved helped the organization shift the software development process from fault detection to fault prevention, a more advanced development paradigm. Engineers became more knowledgeable and skilled in preventing coding faults from the outset—before they reached the customer.

Acknowledgments

The author gratefully acknowledges all the efforts of the 5ESS switch engineers who participated in this project, especially G. M. Gehi, A. Gupta, J. Hendry, D. A. Reimer, B. Verner, D. G. Raj-Karne, and P. Y. Chan. For their support, thanks go to J. M. Perpich, C. E. W. Ward, C. P. Huang, S. T. Huang, S. M. Kania, M. L. Zajac, and P. V. Lessek of Lucent Technologies.

References

1. Watts S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.
2. Capers Jones, *Applied Software Measurement*, McGraw-Hill, New York, 1991.
3. Richard E. Fairley, *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
4. Weider D. Yu, "Verifying Software Requirements: A Requirement Tracing Methodology and Its Software Tool," *IEEE J. Select. Areas Commun.*, Vol. 12, No. 2, Feb. 1994, pp. 234–240.
5. Weider D. Yu, Alvin Barshefsky, and Steel T. Huang, "An Empirical Study of Software Faults Preventable at a Personal Level in a Very Large Software Development Environment," *Bell Labs Tech. J.*, Vol. 2, No. 3, Summer 1997, pp. 221–232.
6. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1988, p. 49.
7. Wai-Kai Chen, *Theory of Nets: Flows in Networks*, John Wiley, New York, 1990.
8. Weider D. Yu, D. Paul Smith, and Steel T. Huang, "Software Productivity Measurements," *AT&T Tech. J.*, Vol. 69, No. 3, May/June 1990, pp. 110–120.
9. *Service Quality Report,* Automatic Reporting and Management Information System (ARMIS), Federal Communications Commission, 1998. See http://www.fcc.gov/ccb/armis

(Manuscript approved May 1998)

WEIDER D. YU received a B.S. in mathematics from Fu-Jen Catholic University in Taiwan; he also earned an M.S. from the State University of New York at Albany and a Ph.D. from Northwestern University in Evanston, Illinois, both in computer science. As a distinguished member of technical staff in the Systems Management and Quality Architecture Department at Lucent's Network Systems in Naperville, Illinois, Dr. Yu is responsible for 5ESS®-2000 interval metrics and measurement, software quality analysis and measurement, defect prevention and root cause analysis, software productivity, requirement specification and traceability, software fault flow and removal effectiveness, and software estimation and cost modeling. Dr. Yu is also an adjunct associate professor in the Electrical Engineering and Computer Science (EECS) Department at the University of Illinois in Chicago, the chair of the Chicago chapter of the IEEE Communications Society, and president of the Chinese Academic and Professional Association of Mid-America (CAPAMA). ◆