
QAC Clinic

Volume 1.1

1998

MANAGERS

PROGRAMMERS

Static Testing

What is it? When programmers and software managers hear the word testing, the usual thought is the process that follows coding. In this type of testing the software program is actually running and its performance is tested. This form of testing is called dynamic testing. The opposite of dynamic is static, thus any testing done while the program is not running, is called static testing. Why are so many people unfamiliar with static testing? It may be that a different terminology is more often used. The term "Inspection" is the more common term used, for example "code inspections" or "design inspection". Another term very often heard is "review", an example would be "code review" or "design review". For the purpose of this text we are going to focus on static testing as it relates to code inspections. What are code inspections? They are just as the name implies, an inspection of the source code either by human or an automated tool, looking for various types of possible errors. The levels of sophistication of a code inspection can vary greatly. The simplest being a programmer reviewing his own code to a very formal code inspection with a review group including a moderator as outlined by M. E. Fagan¹, formerly of IBM. The automated tools used range from simple brace checkers to comprehensive deep flow static analyzers like QAC from Programming Research Limited.

continued on page 2



We are behind schedule. . . Get QAC now!

Japan's Top Five Coding Errors

How often have you wondered how your coding ability compares with others in the industry? What types of mistakes are common, or how does Japan compare with the United States and Europe with respect to the C language?

Before we get into details, let us see where the data comes from. For Japan, the data comes from Toyo Technica's Code Analysis Center. In the past two years the Code Analysis Center has done more than 70 code audits and the total number of code lines now exceeds 2,000,000. Data for the USA and

Europe comes PRL, the manufacturers of QAC.

Let us now compare some specific issues between Japan and USA/Europe. The first is the use of function prototypes.

continued on page 3

Inside This Issue

3 Tools are Available *Dr. Les Hatton*

5 Code Complexity

3 Trusting Compilers

6 The Misspelled Bug

7 Code Analysis Center

Static Testing

continued from page 1

There is a very important point to be made at this time. Properly conducted code inspections are widely acknowledged to be the most important and effective tools for defect-free software²

Figure 1 is a plot of Return on Investment (ROI) for Inspections and Testing. This data is taken from a study done by HP.

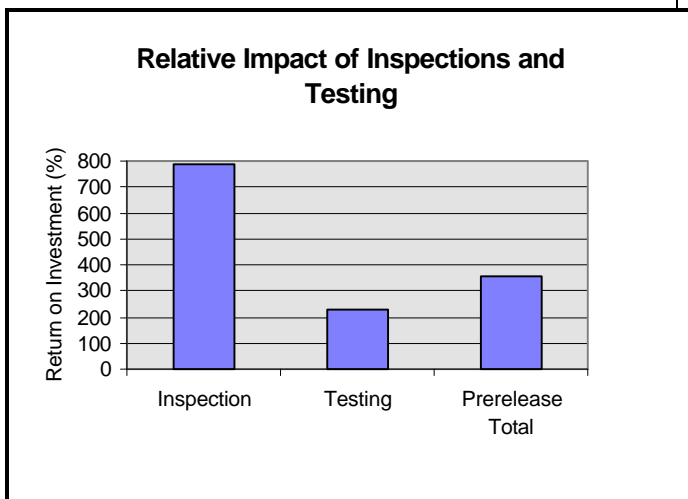


Figure 1

The data shows that inspections have resulted in more than three times the ROI of testing

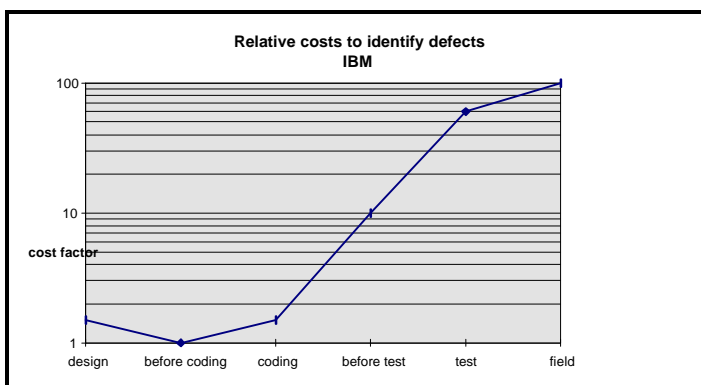


Figure 2

Figure 2 is data from IBM. It basically indicates a relative cost factor for identifying defects over the lifecycle of a software project. As can be seen the cost factor rises very quickly at the test process. This data would appear to support the HP data that the most cost effective location to locate defects is during coding and prior to it.

Conceptually, this is not a hard principle to understand, the sooner a mistake is discovered, the easier and less costly it is to repair. As previously mentioned, code inspections are widely acknowledged to be the most effective tools for defect-free software; but the fact is they are not widely used. There are generally three reasons associated with this lack of use.

1. Poor results, not a significant number of defects identified.
2. High resource costs, both personnel and project schedule
3. Managers have never heard of or seriously studied the merit of code inspections.

There are also many cases due to reasons 1 and 2 where code inspections become very easy targets for deletion by managers who find their projects behind schedule!

During a code inspection, what are the important things to look for?

1. Language syntax and constraint violations
2. Company programming standard violations.
3. Source code layout standard violations.
4. Code complexity within testing limits
5. Correct Data structures used?
6. Appropriate algorithms used?
7. Design criteria met?

Items 1 through 4 involve an extremely large amount of information and when these items are inspected manually by programmers, it would probably be very rare if they found more than 1% of the total problems. It is not that the programmers or possibly dedicated inspectors may not be knowledgeable, it is simply this task is not suited for the human mind!

This portion of the inspection process has to be automated, not only for reasons of accuracy and consistency, but also to complete the job in a reasonable length of time. When this portion of the inspection is not automated, the problem of high cost and lack of significant defect detection is almost guaranteed.

3

Static Testing cont'd

When a deep flow static analyzer such as QAC is used, inspectors can then concentrate on items 5 through 7, the areas where humans are best suited to do the work.

At this point, let us review some of the important points made:

- Static testing, in contrast to dynamic testing is done while the program is not running.
- Code inspections are a type of static testing.
- Code inspections are widely acknowledged to be the most effective in reducing software defects.
- Automated static code analysis is a part of a comprehensive code inspection.

Some of you reading this may have had experiences with static test tools in the past and there are probably many of you who say that they are not experiences you wish to repeat! Why is this? Perhaps one of the best known static test tools is lint, which is or was distributed as part of the UNIX system. Lint is a classic example of a potentially useful tool whose usefulness is limited by the obscurity of its user interface and the frequency of spurious warnings.

Static test tools have come along way since the early days of "lint". QAC, a deep flow static analyzer, while not strictly a lint like tool, does check the same and more items than lint does (800+ vs 100). Another job of a static analyzer is to provide software metrics. These measurements of the code provide objective view of the source. For example, lines in a function, number of branch statements, static path count and depth of nesting are to name a few. A primary use of metrics is to provide insight into the testability and maintainability of the source code.

Another large advantage QAC has over the older lint is that the message content and whether a message is issued are configurable. This allows users to reduce the messages or place the message emphasis in certain areas. It is also possible for users to incorporate their company programming standards into QAC so it is automatically checked. Many managers would probably agree, if standards are left to be checked manually, it is never done!

In the past ten years the use of application software and the amount of embedded software in products has increased dramatically. It touches our lives in untold ways every day. In the event of software failure, the

QAC Clinic

result could mean loss of life or large monetary loss. It is simply not acceptable not to use every means available to reduce the rate of software defects and static testing has been ignored for far too long.

Tools Are Available

Dr. Les Hatton, author of *Safer C*

"It has been known for many years that software engineering is particularly susceptible to classes of repetitive failure that statistical process control in the manufacturing industries has systematically eliminated in the last 20-30 years. Of these repetitive failures, analysis reveals that many of them could have been detected using techniques we already know how to do. They are not detected because of an absence of tool support. Code inspections supported by automated static fault detection using tools such as QAC have been proven to be very effective in controlling repetitive software failure, and help close the quality gulf between the current state of the art in software development and its much more mature cousin, hardware development."

The C language is like a knife: simple, sharp, and extremely useful in skilled hands. Like any sharp tool, C can injure people who don't know how to handle it.

Japan's Top Five

continued from page 1

Since the ISO C Standard became effective in 1990, the use of function prototyping has been allowed and very strongly encouraged. The reason is that without prototyping, the compiler is unable to check argument types when calling a function. Although it has been 7 years since the Standard was signed, the use of prototypes in the USA/Europe is only about 60%. . . but Japan is only 20%, even worse. Another item to compare is the implicit declaration ratio, in other words, how many functions were called with no declaration of any type. The data type of a function with no declaration in C defaults to an integer. The ideal figure for the implicit declaration ratio is zero, meaning all functions are correctly declared. The figure for USA/Europe is 9% and Japan is 15%. A more general comparison of wider

4

scope can also be made. QAC's warnings are placed in groups for greater user control, for example, portability, maintenance and such. There is one group called "Major" and this is where all warnings of a serious nature are placed. The USA/Europe average of "Major" warnings per 1000 lines is 12 (12/KLOC). Japan's average is 65 (65/KLOC).

Since the ISO Standard and the C language is the same the world over, I'll leave it to the readers to debate why Japan's numbers are worse than the USA and Europe!

Now let us look at the top five problems that are seen by Toyo Technica's Code Analysis Center.

The largest problem both in quantity and potential seriousness is the implicit data conversion to a smaller type. For example, integers assigned to character types, integer assigned to shorts or long data types assigned to integers.

```
int foo(int a)
{
    char c;
    c = a;
    c++;
    return c;
}
```

In the above example, integer **a** is assigned to character **c**, thus if the data in **a** is large there is a possibility that data could be lost

The average rate for code audited to date is 1 implicit conversion per 30 lines (1:30) it should be remembered that a static test warning of this type simply indicates this condition exists, it really requires a dynamic test with real data to verify whether a fault exists or not. On the other hand, if this static condition did not exist, a dynamic test would not be required would it! This problem is probably more common in Japan than in the USA and Europe because of the large number of embedded systems applications in Japan and the programmers tendency to use small data types to save memory.

Another major problem is the possible use of unset variables.

The average rate is once per 250 lines (1:250) in Japan. The average in the USA/Europe is once per 840 lines. This problem can not be attributed to any type of application, this is simply a programmer problem. It should be noted that the frequency of this problem increases quite rapidly when there are many long

QAC Clinic

functions with deeply nested branch statements in the source code. The reason would appear to be that when programmers are writing code within the "true" state of a branch statement, that is where they concentrate their thinking. The consequences of the "false" paths are not given equal weight.

```
int foo(int a)
{
    int b;
    if (a) {
        b = a;
    }
    return (b) ? 1 : 0;
}
```

As can be seen, **b** receives data within a branch statement thus if **a** were false, there would be a chance of using **b** with unknown data.

The third problem often seen is improper casting of types. Casting itself is really a double edged sword in that it can indicate when a proper type conversion is taking place and it can also mask an improper type conversion. The term masking here means it hides the problem. Compilers and static test tools will cease to give a warning for an improper type conversion if they encounter a cast. They assume the programmer knows what her or she is doing! The following code fragment will illustrate a very common casting error.

```
char c;
int a = 10;

(char)c = a;
```

The above problem is not only incorrect casting, it is a syntax error because it is illegal to cast an lvalue. Now why do programmers do this? One reason is that they have probably never really studied what is proper casting and the second reason is most compilers will never issue a warning message, they simply ignore the cast. In fact in one compiler, the above is an extension of the C language specific to that compiler!

Casting incompatible pointers is also seen quite often. It is suspected that in these cases, the compiler has actually issued a warning about the assignment of incompatible pointers, but the programmer has added the cast to quiet

5

the compiler. This practice is of course very dangerous and can lead to some very hard to find bugs.

Number four on the top five list is the improper creation and use of macros. Macros, as in the previously mentioned problems, can lead to very hard to find bugs. The problem here is the programmer does not actually see the expanded code after macro expansion has taken place, thus visual inspection and debugger use are very limited in their effectiveness. Let us look at a very interesting problem.

```
#define AAA 2
#define BBB 4;
#define CCC 8

int foo(void)
{
    int i;
    int idx = BBB;
    for (i = 0; i < BBB; i++) {
        printf("the number is %d", idx);
        idx++;
    }
}
```

In the assignment of `idx` the actual expanded line is:

```
int idx = 4;;
```

notice the second semicolon, now the expansion for the next line:

```
for (i = 0; i < 4;; i++) {
```

The extra semicolon creates a situation where `i++` is never executed, thus an endless loop has been created. The most common problem with macros is that programmers do not fully parenthesize the macro and its arguments

```
incorrect
#define cube(x) x * x * x
```

```
correct
#define cube(x) ((x) * (x) * (x))
```

To complete our list of the top five coding problems, we include the confusing issue of “order of evaluation” and “precedence”. This is not only a problem in Japan, but also in the USA and Europe. The fact is that programmers very often confuse the two and also think that they are inter-related. Consider these code fragments:

```
a = foo() * boo() + coo();
```

QAC Clinic

Now most programmers would agree that the multiplication would occur first and then the addition, but what would your answer be if the question was which function is called first! There are a great many programmers who would say an expression is always evaluated left to right, thus `foo()` is called first. What would happen if we re-wrote the code adding parenthesis, now which function is now called first?

```
a = foo() * ( boo() + coo());
```

The answer is we do not know, the ISO Standard does not specify the order. The order is determined by the compiler and it may be different depending upon the context. By adding parenthesis to the previous code, we have changed the precedence of the calculation so that the addition is done first. If the contents of the three functions were absolutely independent of one another, it may not make any difference which one was called first, but what if they all accessed and changed the same global variable? The result of course, could be a very hard to locate bug.

Precedence refers to operators and is clearly specified within the ISO C Standard. The order of evaluation refers to operands and is unspecified. Any programmer who writes code that depends on the order of evaluation for correct operation is simply creating headaches for himself, other programmers and the project manager!



With the exception of some instances of unset variables, a compiler will never warn a programmer of any of the previously mentioned problems. So how do programmers and project leaders protect themselves, they use a good static analyzer tool such as QAC.

Code Complexity

What is code complexity? Why should we worry about it? How, if at all, does it impact our software projects? First, let us look at what a dictionary says about the word complexity:

6

“complexity is the state of having many different parts, which are connected or related to each other in a way that may be difficult to understand or deal with”.

From the above we can assume that the greater the complexity, the greater the difficulty in understanding. What the dictionary does not tell us is at what level of complexity does a human lose control!

Now why should we worry about code complexity? The reason is that it has a direct bearing on whether the code is testable and maintainable.

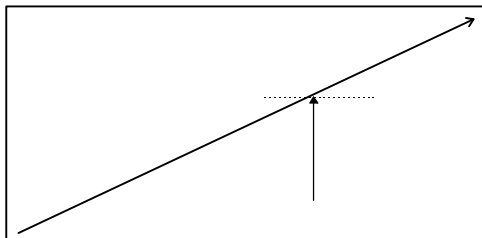
Here is what Thomas McCabe said:

“There is a critical question facing software engineering today: How to modularize a software system so the resulting modules are both testable and maintainable”³

Interestingly enough, this statement was made in 1976, and software managers and programmers are still facing this critical question 22 years later!

Using some simple logic, if module of a program is too complex to understand, how can someone maintain it? If a module is too complex to understand, how is a test engineer going to write the comprehensive test cases for the code? There is also another issue, quantity, what about the thousands if not millions of test cases needed to thoroughly test this complex code? The magnitude of the numbers would lead one to assume there is a lot of untested and buggy software being used today!

I think everyone would agree that when programming begins, coding complexity is zero. As programming progresses over the months, there will be various modules whose complexity finally exceeds understanding or the test case quantity has exceeded the time and money resources allotted for the project.



How do we know when we reach this point? This is a very important question, because at the end of coding cycle, it is too late to find that the code is not testable or nobody can understand it.

QAC Clinic

The need to measure of software complexity is a long understood problem and over the years many methods have been developed. Most of these measures are syntactic in nature and frequently involve counting one or more textual properties of the source code. Taken as a group, these measures are generally referred to a “software metrics”. A technical description of all the different “metrics” is beyond the scope of this article, but a couple will be described to illustrate the concept of complexity. Thomas McCabe published a paper in 1976 stating that a program’s complexity is defined by its control flow and is measured by counting the number of “decision points” in a routine. In C, we do this by starting with 1 for the straight path through the function and then adding 1 for each of the following keywords: **if**, **while**, **for** and **case**.

McCabe’s metric is commonly called Cyclomatic Complexity today. McCabe stated that when the Cyclomatic Complexity exceeded 10, the routine was probably too complex for human understanding and therefore the defect rate was likely to increase rapidly. The reader should be careful here not to assume the metric - complexity relationship is rigid and inflexible.

As an example, the complexity of 10 nested **if** statements is undoubtedly higher than 10 **case** statements. McCabe’s metric also does not take into account the added complexity of the logical operators within a branch statement as shown below.

```
if (a) {  
}  
  
if (( a && b && c ) || d) {  
}
```

A man by the name of Myers later added an extension to cyclomatic complexity by counting the number of logical operators within a branch statement. The metric is presently called Myers Interval or Extended Cyclomatic Complexity. McCabe’s metric in conjunction with Myers gives a much more realistic view of the complexity of conditional branch statements, especially considering that complex logical decisions are very prone to fault.[Moller 1993]⁴

Another useful metric is the “Static Path Count”. This is a measure of the testability of a program and simply counts all paths through a program. In essence, parallel paths add and serial paths multiply. Various studies have given different numbers, but generally, suggested maximum numbers would be in the range of 200-1000. An interesting note, QAC’s maximum figure for static

path counts is 500,000,000 and this number is seen very often in customer code. If a software project requires 100% of the paths to be tested, this metric would give a very good indication of the number of tests required!

The last metric to be explained is “code lines”. This metric can be very elusive because there are many ways of counting code lines, i.e. what is counted and what is not. QAC has four different line count metrics, but one of the most useful is “executable code lines” within a function. This metric thus allows the user to see where the “long” functions, it is easy to monitor and it is also the basis for the COCOMO models.

An obvious question that may be in your minds now is what is an appropriate line limit to a function? The numbers that are available in books and research papers vary to quite a degree. A number many are familiar with is “50 lines per function” this can be found in “C Programming Guidelines” by Thomas Plum⁵. Some years ago the reasoning was that “it fits well on a screen”! There has also been some research done by Dr. Les Hatton that shows the lowest defect density occurs when the module size is about 200 lines, irrespective of the coding language used! As a compromise between McCabe’s control complexity and Dr. Hatton’s research, an effort to keep function lengths under 150 lines may be a good starting point.

As was said earlier, finding that a completed software project has high complexity and can not be tested is a bad situation to be in. A better approach would be to monitor the complexity on a daily basis and when modules start to approach some preset limits, stop and examine the code for the reason. Very often a design problem is the reason for “growing” functions.

The only viable method for daily monitoring is to use an automated tool such as QAC, which besides testing for C or C++ language errors, will also generate a full suite of software metrics. Aside from creating an ASCII text file containing the metric information, QAC will also generate a comma separated file (CSV) which can be placed in a spreadsheet for further processing.

Function	Cyclomatic Complexity	Myers Interval	Nesting	Code Lines	Static Path
Wnd_message	103	35	11	589	500000000
Wnd_create	46	13	7	270	264241200
Wnd_move	24	5	4	96	165888
Wnd_destroy	27	6	4	146	93696
Msg_handle	23	15	3	61	86016
Msg_receive	21	6	3	93	77760
Msg_transmit	37	1	3	36	20480
Wnd_zoom	20	4	3	45	13056
Msg_key	10	0	2	18	96
Msg_mouse	13	6	6	65	80
Msg_queue	9	3	2	51	72

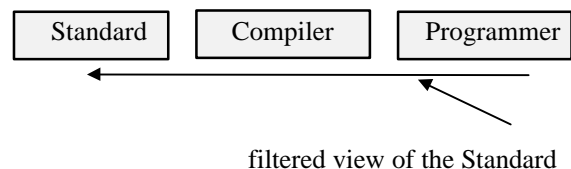
QAC Clinic

Shown is a sample of typical metric data from QAC in spreadsheet format and the information has been sorted on the “Static Path” data. A programmer or manager can very easily see that the top two functions “Wnd_message” and “Wnd_create” have very high metric numbers in at least four categories. In fact, these numbers indicate these two functions are not only totally untestable, but also probably not understood by the programmer who wrote them! The result is that unless these two functions are corrected, the entire project will suffer.

This ability for programmers and particularly managers to monitor and pinpoint areas of a program that are in trouble early in the coding cycle is invaluable in keeping a project on schedule and free of defects.

Trusting Compilers

I think many programmers would agree with the following situation, when coding, they compile at least once a day. The question is why? It certainly can not be to get an executable program! I think most programmers would also agree, they are testing the compilability of the code, i.e. looking for mistakes! Probably to clear up minor syntax and typing mistakes and also testing the compiler’s response to some syntax they are unsure of. For example, dereferencing a pointer to a pointer or maybe accessing an element in a multidimensional array. In these situations the programmer is relying on the “compiler’s knowledge of C”. In these cases, it would probably be safe to say the programmer is getting a rather “filtered” view of what is really “correct” C.



A question, how many times have you seen a compiler maker advertise they sell a “strictly conforming” ANSI C compiler? Or even a “conforming” compiler? Probably none, the reason is that some of the extensions that have been added to the compiler actually place them in the “non-conforming” group. Obviously this wouldn’t

sell products so what you may see is the claim “ANSI Compliant”. The Standard does not actually define the word compliant and a dictionary would reveal that “conforming” and “compliant” have nearly the same meanings. Thus if the claim is “ANSI Compliant”, the customer will likely accept it and purchase the compiler, but what does this actually mean? To what extent is it compliant?

It may be a surprise to some programmers, but a compiler is not a static testing tool and was never intended as such. To understand and appreciate this situation a little more clearly, we have to regress a bit and discuss the ISO C Programming Standard and what it is. You do have access to a copy don't you? The C Standard has two basic purposes. One, it defines the C language and second, it defines the responsibilities of the programmer and the responsibilities of the compiler.

Successful compilation of a program would of course be fine if reliability, portability and maintainability issues were enforced by a compiler, but unfortunately, comparatively few demands are placed on even a validated ANSI C compiler and many surprisingly dubious code fragments can compile successfully, leading to unexpected behavior at run time. In C, problems can arise from the following sources.

- a) Syntax errors
- b) Constraint violations
- c) Formally unspecified behavior (22 items)
- d) Formally undefined behavior (97 items)
- e) Formally implementation-defined behavior (76 items)
- f) Formally locale-specific behavior (6 items)
- g) Accidentally undefined behavior, (things the committee forget to define)
- h) Empirically determined misbehavior, (items which are perfectly well defined, but programmers still get wrong)

Of the above list, a validated compiler is required only to detect items in groups a) and b), in other words, those are its areas of responsibility.

Diagnostic Messages

This is what the ANSI Standard says about “Diagnostics” i.e. the messages a compiler is required to give.

“A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit (i.e. file) that contains a violation of any syntax rule or constraint. Diagnostic messages need not be produced in other circumstances.”

So as can be seen, compiler makers have a very wide latitude in how, when, where and how many warning or error messages they generate. The number of messages and the exact wording is of course going to be different between various compiler manufacturers.

Validated Compilers

Although there is now a C Standard available, this does not mean that implementations (compilers) adhere to it, or if they try to, to what extent. In order to provide this guarantee, validation suites are used to insure that an implementation is indeed conformant. There are two commonly used validation suites, one in the USA and one in Europe. The National Institute of Standards and Technology (NIST) uses the Perennial validation suite. In Europe, the British Standards Institute (BSI) chose the Plum-Hall validation suite. It should be noted that the validation suites are quite difficult and not very many compilers have been validated. A list of compilers that have passed the Perennial suites are available from the NIST in its “Validated Products List”

Another area of concern is the treatment of floating-point arithmetic. The C Standard does not define floating-point manipulations although they are under discussion by the Numerical Extensions Group.

There are some exercise programs available to assess the quality of the support for floating-point computations on any particular compiler/machine combination. Probably the best known of these is paranoia. It is available as a C program by sending the message `send paranoia.c` from `paranoia` to the Internet address `netlib@research.att.com`. The resulting source code can then be compiled and run as required. The following issues are examples of what is uncovered.

- Adequacy of guard digits for multiplication, division and subtraction
- Whether arithmetic is chopped, correctly rounded or something else for multiplication, division, subtraction, addition and square root.
- Whether a sticky bit is used correctly for rounding.
- Whether underflow is abrupt, gradual or fuzzy.
- Whether infinity is represented.

9

- Whether comparisons are consistent with subtraction.

In summary, the compiler's task is to compile source code, not to warn you, the programmer that you are making a mistake. Put a quality measurement tool such as QAC in your toolbox!

The Misspelled Bug

While writing source code, if a programmer mis-types something it will very often be caught by the compiler. For example, if a keyword is misspelled, a semicolon forgotten, the compiler will complain with some type of syntax error. There are some very subtle typing errors that the compiler will ignore. Consider the following code: (ignore the poor style)

```
void foo(int n)
{
    int m;
    switch (n)
    {
        case 1: m = 2; break;
        case 2: m = 4; break;
        case 3: m = 8; break;
        default: m = 0; break;
    }
    return m;
}
```

Can you locate the error? Unfortunately, a compiler would be of no help here, it would be perfectly happy with the above code. When the code here is scanned with QAC, the following message would result:

"WARNING (3202) The label 'default' is not used in this function and could be removed"

As can be seen, the "default" was misspelled, but why was it not detected by the compiler? The reason is that the "case" and "default" keywords within a switch statement are considered labels, the same as "goto" labels. Since it is legal to embed a goto label within a switch statement, the misspelled "default" is simply considered an unused label by the compiler and it is ignored! A very dangerous bug and also one that is extremely hard to catch by eye.

QAC Clinic

There are other classic examples of simple typing mistakes that results in code that "appears" correct to the compiler.

```
if (a == b) {
}
```

```
if (a = b) {
}
```

Which of the above branch statements is a typing error?

The next error is actually seen quite often in customer code.

`i == j;` where `i = j;` was intended!

As said earlier, the compiler will remain very silent, whereas QAC will issue warnings in all three cases.

Code Analysis Center

¹ M. E. Fagan, “Design and Code Inspections to Reduce Errors in Program Development” *IBM System Journal*, Vol.15, no3, 1976, pp 182-211

² Louis A. Franz, “Estimating the Value of Inspections and Early Testing for Software Projects”, *Hewlett-Packard Journal*, Dec. 1994

³ Thomas J. McCabe, “A complexity Measure”, *IEEE Transactions on Software Engineering*, Vol, SE-2 No.4 Dec 1976

⁴ Moller, K.H. “An empirical investigation of software fault distribution”, CSR '93, Chapman & Hall.

⁵ Thomas Plum, C Programming Guidelines, ISBN 0-911537-07-4 page 100