Programming Techniques for Zero-Defect Software

Bruce Toback
Infotek Systems

Introduction

All non-trivial software has bugs.
-- Anon.

Genesis: This oft-quoted saying was quoted once too often to the author by none other than his Hewlett-Packard SE, referring to MPE. Inasmuch as the I am a daily user of this non-trivial software, it occurred to me that perhaps I should examine the underlying foundations of that which is presently the *art* of programming. After all, part of *my* job is to write non-trivial software.

This saying is as much a bias on the part of most programmers and system designers as it is a fact of the computing world today. Is there a cause-and-effect relationship?

It'll be fixed in the next revision.
-- Original author unknown.

Perhaps because software is so "easy" to change, this saying has become almost the watchword of the software industry. Mass-produced hardware products may go through two revisions in a year, and these trivial. Software products may go through two revisions a month, some of them major. This flexibility engenders the "next revision" syndrome: It's OK to leave a problem in the code this time around 'cuz we'll get it next time. And "next time" is only a few weeks away, anyway. Enhancements work this way, of course: enhancements are essentially fixes to problems in the initial system definition. Barring some mutation in the human genotype permitting clairvoyance and/or precognition, the need for enhancements will always exist; there will likely also exist some doubt as to whether a particular change constitutes a "fix" ("it doesn't perform the way I said it should") or an "enhancement" ("it doesn't perform the way I really need it to [now]"). (Anybody who has had their "Problem Report" come back marked "Enhancement Request" will understand this dichotomy.)

Get it right the first time. --
Everybody at one time or another

I have yet to encounter a professional - *any* professional - who does not quote this saying frequently. It tends to become less frequently quoted as deadlines approach, particularly software release deadlines. Strict adherence to this principle need not mean missing deadlines: one need only bend the definition of "right" slightly to insure that this does not happen. "Enhancements" change something over which the programmer has little control: the definition of a defect. (Actually, politically savvy programmers can get quite a bit of control over the definition of a defect, but that is a subject for another paper.) "Fixes" change *only* those things over which the programmer has control. Fixes should always be right the first time.

The phrase "first time," as used in this paper, needs some clarification. An iterative process is inherent in almost any creative endeavor. (Programming, for better or worse, is still a creative endeavor.) "First time," as used here, means the first time anyone except the programmer is asked to use or evaluate the product. ("Programmer," as used here, means not only the individual actually doing the writing, but managerial support, librarians, and others whose primary task is to create *and support* the software products that others will use.)

Zero defect software requires not only good program construction, but good problem definition, testing, and **repair** facilities as well. Systems should be capable of tolerating program errors in one module without causing a "defect" visible to the user. In addition, with the use of redundant design and checking, systems can and should be capable of repairing problems caused by some program(ming) errors. (The era of the "no-flat" program is at hand.)

A Word about Modules

Definition

Many definitions of a software "module" have been used in the past, with the definition usually set up to suit the point being made. For this paper, a "module" will be a piece of code with a single well-defined function, and a set of well-defined input and output variables. Input and output variables are not limited to function arguments, of course: they include data set and file records, user-input parameters, forms, and other entities that the program can manipulate. In FORTRAN, PASCAL, and SPL, input and output variables include global or COMMON area values that are used or modified.

This paper will not specify the *form* of the module: it could be an entire program, a lexically defined unit such as a procedure or subroutine, or simply a particular group of program lines.

Understandable function

Relative to writing zero-defect software, the *primary* purpose of a module is to provide a well-defined function that can be tested thoroughly. It is almost impossible to test a piece of a program, regardless of its lexical structure, whose function is poorly defined. By doing this, once a piece of code has been determined to perform its specified function, it can be used forever as a "black box:" its purpose *and its effects outside its defined purpose* will be dependable and well-understood.

In-line testing

Testing is important because no technique published to date results in perfect code the first time. (Programmers, like engineers, build and perfect prototypes of their designs before (one hopes) committing them to production. A successful zero-defect methodology will recognize this and not rely on first-time perfection.)

One function of a module is to provide a suitable unit for *in-line testing*. In-line testing of a large, straight-line program is very difficult, since a the function of any lexical entity larger than a statement is hidden in the oversized structure of the program. (Which part of the spider web is *this* thread holding together?) Some programmers, when faced with an oversized structure, determine that in-line testing is useless, and fall back on the technique of repeatedly removing program lines and rerunning the program until they come up with an understandable result, then replacing lines until they understand the function of each removed line and find the faulty one (or ones). Carried to its extreme, this technique leads to rewriting the entire "module." If the same structure is used in rewriting as was used in the original writing, a career loop results.

Interpretation of Test Results

If a piece of a program performs an incomprehensible function, or has too many inputs and outputs, interpretation of test results becomes nearly impossible. This is because one technique of testing may be to vary one of the inputs and watch the results on the output. If 53 outputs must be watched, and their movements interpreted in light of the single input change, one may get the feeling that a certain amount of random variation is involved. In such situations, the tester tends to get the feeling that "this output *looks* right; it must be OK" and the test session ends. If inputs and outputs are ill-defined, this latter test method is often the only usable one.

Complete Testing

A piece of code with an ill-defined function tends to inhibit complete testing of itself, since it is difficult to know exactly what constitutes "complete" testing of something whose function is unknown. In addition, a set of input or output variables that is too large inhibits complete testing, simply because testing all possible combinations of inputs (including those which the definition of "right" precludes as illegal) will be too time-consuming. (The assumption is that some deadline eventually needs to be met.) With a small enough set of input and output variables, and a sufficiently well-defined function, not only can all possible combinations of inputs be tested, but all possible values as well. (In practice, modules for which all possible input values can be tested tend to be trivial, since the testing program needs to be at least as complex as the module under test in order to know all the right answers!)

BITE

BITE is an acronym from the ever-fertile acronym fields of defense hardware. It stands for Built-In Test Equipment, and provides a way of testing equipment without special additional equipment. For hardware, the built-in test equipment generally consists of some generalized testing tools such as multimeters, and some specialized testing tools designed to measure critical operating parameters of the equipment being used.

The same concept can be applied to software: built-in test "equipment" can be included in production programs, to be activated by a programmer, a tester, a support person, or (perish the thought) a customer under the direction of a support person. The test function can be used in three ways. First, of course, it can be used to debug a prototype module before it is released to production. Second, it can be used during system integration to determine the location of a problem, both to the module involved, and to the portion of the module which may be in error. Finally, it can be used to diagnose production programs which have failed in some fashion, either by producing "incorrect" answers, or by unexpectedly aborting. (More on "expected" aborts shortly.) BITE can come in either of two varieties: monitor and diagnostic.

Mutually Suspicion

The concept of mutually-suspicious modules is a general rule in operating system design, where reliability and security are of paramount importance. (This is an example of a self-fulfilling expectation: everybody *expects* operating systems to be absolutely reliable and secure. Why not all application systems as well?) Mutual suspicion means that modules do not rely on their callers for input checking (is the date being sent to the date conversion routine valid?), nor on the modules they call for output checking (does the record you [called routine] said I [calling routine] wanted really exist?). Monitor-type test "equipment" is used to implement this kind of mutual suspicion.

Monitors

In general, pieces of any system should attempt to monitor other parts of the same system. Monitors can be divided into two rough categories, cheap and expensive. The activities being monitored can also be divided into two rough categories: critical and non-critical.

Cheap monitors are those which cost almost nothing to use. (They may take considerable effort to implement, although this is unusual for a **really** cheap monitor.) An example of this kind of monitor is a statement which checks to see if the program is about to divide by zero. Division by zero will (or should) almost always result in an unexpected program abort because it is mathematically undefined. In most cases, the real-world operation being represented by the division is also undefined if its divisor is zero. (What is the cost per item if you don't have any items?) A monitor will check for this and perform any appropriate action. Remember that it is not enough to assume that the system definition (the definition of "right") precludes this: the program in this case should check the system definition. Cheap monitors monitor an operation *every* time it is performed.

Expensive monitors are those which can cost a lot to use. Examples of this kind of monitor are those which check to insure that a summary field in a master record matches the total of the individual detail records. Naturally, the module which takes care of the detail records *always* takes care of the summary. Doesn't it? One way to check this is to call another module which performs the *very* well-defined operation of adding all the details and comparing it to the summary. But since this may involve going through a large number of database operations, it is an expensive monitor: not one which you want an interactive (and impatient) user to wait through every time he or she enters a purchase order item, for example. Expensive monitors therefore need to be "triggered" by some event, either from within the system or outside it. Sometimes the monitor may be outside the software, as when the computer operator instructed to run a data base integrity check every day just before backup. Indeed, "expensive" monitors should always be triggerable manually, just as a check on automatic triggers.

Monitors can be triggered by the system itself in a number of ways. One way is to trigger the monitor every time an infrequent operation is performed. For example, in the MPM/3000 manufacturing system, a check of inventory summaries for a part is triggered every time the user does a physical count on that part. The trigger may be periodic, i.e., every *n* times an operation is performed, the monitor is triggered. The monitor may be probabilistic, i.e., triggered 5% of the time a particular operation is performed. Finally, the monitor may be time-triggered: run every hour, for example.

Very expensive monitors generally need to be controlled by some of the same techniques used to control diagnostic test routines, in addition to their normal triggers.

Damage reporting

What happens if a monitor determines that an error has occurred? This usually depends on the severity of the error and its likely effect on the remainder of the system.

In all cases, the failure should be logged. This can be done using the MPE logging system, or the circular file (as distinguished from the more usual "round file") facility provided by MPE and some other operating systems. The implication is that even if an error is determined to be correctable, its existance must be logged: there *is* a defect in the software, even if it didn't cause a defect in overall system operation.

Obviously, correcting the error is the most desirable option. This implies that one part of the system is deemed by the system designer to be more trustworthy than others (this situation rarely occurs), or is defined by others. Examples of the latter include recording the total amount of a purchase order: the total field is *always* defined by the sum of the amounts in each line item, regardless of whether the amount in each line item is correct. (If this were not so, correcting the line item error would cause a problem with the summary.) Correcting the error should be done *only* if determination of the correct value is not more complex (or identical in method or complexity) than the process which originally generated the error!

Sometimes correcting the error is not possible. This is usually the case with "cheap" monitors: the monitor would not have caught the error if the software which supplied the parameters to the monitor had been working. In this case, the designer, and sometimes the software, must make a determination as to the consequences of continued operation with the data error in place. Sometimes, a default value can be substituted (i.e., a divide-by-zero caught in the act can be made to have a result of zero) that will be as acceptable as the original data. Other times, part or all of the system must shut itself down (gracefully), let the user know what has happened, save as much context as possible for later analysis, and then terminate. This is where a problem detected by the monitor is declared critical or non-critical. A non-critical problem may simply shut down a single function of an overall system, e.g., it may prevent further entry of purchase orders. A critical problem will require shutdown of the entire system. Note that at this point, the zero-defect methodology is no longer involved with preventing defects; it is concerned now with damage control.

Diagnostics

The other type of built-in test "equipment" is the diagnostic. In some sense, the compilers and the operating system already provide diagnostic tools: when a program aborts, MPE prints the location within the program at which the abort occurred, and if asked, will display the data in the program's stack at the time of the abort.

Several problems arise when using the "default" diagnostics. First, they are *very* unfriendly: the "stack display" is recognized throughout the HP3000 world as a sign that the programmer blew it. Second, they are not very informative: they can be interpreted *only* by the programmer, if at all, and present the death of the program as a singular event, without even an invitation to autopsy. Finally, they are useful *only* in the event of a program abort; they cannot be used to diagnose a logic error.

If you are one of the few users that program in FORTRAN or SPL, HP provides a subsystem called TRACE/3000, invoked through the use of the $TRACE compiler command. If you know about this subsystem, you are one of a select few: many HP SE's are not even aware of its existence. TRACE/3000 was created in the days when the HP3000 was to be an engineering number-crunching machine, and FORTRAN was to be the primary language. Things have not quite turned out that way. TRACE/3000 is documented in the TRACE/3000 manual. It is not terribly useful, with an obtuse command language and an uneducated support staff. It does not operate with COBOL, COBOLII, RPG, BASIC, APL, PASCAL, or TRANSACT.

Program tracing outside of TRACE/3000, however, is a common and useful technique. Typically, though, there are a number of disadvantages to its use. First, trace messages can typically be activated and deactivated only at compile time. In some languages, the only way to do this is to comment out or physically remove the statements. Commenting is not a well-controlled process, and physical removal of the trace logic is not conducive to retaining the same set of trace messages the next time the module is tested. A better way is to use conditional compilation (for any language except BASIC) to add and remove tracing statements (and other diagnostics as well). HP's compilers have two commands that implement conditional compilation. (Remember that a compiler command tells the compiler to perform some action itself, and does not directly affect the operation of your program. Examples of compiler commands are $TITLE (tells the compiler to title your listing), $PAGE (tells the compiler to skip a page and perhaps add a new title), and $CONTROL (tells the compiler to perform various other actions.)

The compilers provide ten *switches* which can be set and tested by the compiler. These are typically used not only for controlling diagnostics, but for including or not including features in a software product. Compiler switches are named X*n*, where *n* is a digit from zero to nine. They are set with the $SET compiler command:

```
$SET X1 = ON
```

or

```
$SET X2 = OFF, X5 = ON
```

These settings can then be tested (by the compiler, remember) with the $IF command:

```
$SET X1 = ON
  .
  .
$IF X1 = ON
        DISPLAY "ADD-ITEM: TOTAL-COST is now ",
                TOTAL-COST.
$IF
```

In COBOL, this may be redundant, because the PROCESS DEBUG statement may be used. The method works well for other languages, however, as well as for COBOL implementations in which PROCESS DEBUG is not included.

The problem with this type of tracing is that it is available only after recompilation of the system. This is fine *if* there will be an opportunity to duplicate the problem with a different program. During testing, this method of debugging is not particularly desirable, since either two versions of object code (one with and one without the trace messages) must be maintained, or alternatively the program must be recompiled (possibly after changing the compile switches) in order to trace its activity. In addition, in some cases the trace messages themselves may have an effect on the execution of the program.

A method through which diagnostics may be selectively enabled or disabled at run-time seems a better alternative. (You might still want to eliminate the diagnostics in the "production" version to save object code space, but this may be of limited value.) One way to do this is with a debug flag that is set by the INFO or PARM parameters on the RUN command. The program can then test the debug flag to determine whether or not to print trace messages:

```
77  TRACE-FLAG  PIC S9(4) USAGE COMP.
  .
  .
  CALL "GETPARM" USING TRACE-FLAG.
  .
  .
  IF TRACE-FLAG IS NOT ZERO
     DISPLAY "ADD-ITEM: TOTAL-AMOUNT now ",
             TOTAL-AMOUNT.
```

GETPARM is a library routine which obtains the run PARM for a program. (It is available in the contributed library.)

This can be extended to provide multiple levels of tracing:

```
IF TRACE-FLAG IS NOT ZERO
   DISPLAY "ADD-ITEM: TOTAL-AMOUNT now",
           TOTAL-AMOUNT.
IF TRACE-FLAG IS GREATER THAN 3
   DISPLAY "ADD-ITEM: Item amount was ",
   ITEM-AMOUNT.
```

Thus, the programmer (or the tester) can select a particular level of trace detail. (The programmer must, of course, be fairly astute in determining what "appropriate levels" should be.) This greatly facilitates the use of tracing during the "prototyping" and system test phases of program development.

(Hewlett-Packard uses this method for V/3000 and some other products: try running FORMSPEC with PARM=1 sometime.)

One problem with this method of doing things is that all modules are traced at a given level, when in fact we are usually (one hopes always) concerned with tracing a specific module. We may in fact wish to use trace messages from more than one module in order to check inputs to and outputs from the module under test. One way to do this is by having a separate trace flag for every module. The problem, of course, is assigning values to all these trace flags when the program is run. Other than simply asking the user/tester about it from within the module under test, is there a way to do this?

By using JCW's (job control words), we can set up *symbolically* a separate trace flag for each module independently. For those modules we don't want to trace, no job control word is set up:

```
:SETJCW ADDITEM=1
:SETJCW DISPLAYORDER=4
:RUN POSYS
```

In the program, then, ADD-ITEM and DISPLAY-ORDER can interrogate job control words ADDITEM and DISPLAYORDER, respectively, to see if they should be traced. This is done with the FINDJCW intrinsic. (The default, if no JCW is set, will usually be "don't trace.")

```
ADD-ITEM.
      MOVE "ADDITEM " TO JCW-NAME.
      CALL "FINDJCW" USING JCW-NAME,
           TRACE-FLAG,
           JCW-STATUS.
      IF JCW-STATUS IS NOT ZERO
           MOVE 0 TO TRACE-FLAG.
```

A similar set of statements at the beginning of each module can be used to define the trace level for that module. (The example, by the way, is contrary to the principles of Zero-Defect Programming. Can you see why?)

By using this method, then, the programmer, the tester, or (perish the thought) the user can selectively activate various levels of tracing - of diagnostics - to determine what the program is doing at any given time.

Symbolic Debugging

Symbolic debugging techniques such as tracing can be extended to include other concepts. Using a control-Y trap, it is possible to allow the programmer or tester to execute a special purpose debugging subsystem. Such a subsystem may be command-driven and may be used to print various record values, to display the status of open files, or to run various "expensive" monitors on demand. For large systems, this technique is invaluable and well worth the amount of work it takes to implement. Typically, such a subsystem will require between five and 10 percent of the code required for the entire system.

Conclusion

Program bugs need not be a part of every non-trivial software system. With proper care in construction of the software, and with the assistance of some simple programming and testing assists, bug-free software can be delivered to the end user every time. (I recognize that some of these methods are impractical for use in debugging interrupt-driven I/O drivers, but very few applications include these!)

To summarize the techniques presented here:

1. Divide your programs into well-defined modules - logically if not lexically. Make sure that each has a name.
2. Include "cheap" monitors to check inputs to each module every time it is used.
3. Include "expensive" monitors to check system data base integrity at intervals as frequent as the user will tolerate.
4. Provide for a method for logging errors discovered by your monitors. Use the softest failure mode you can without jeopardizing system integrity.
5. Provide a method for repairing as well as logging all repairable errors.
6. Provide a trace facility that can be used by the programmer, the tester, the support person, or (perish the thought) the user. Try to make the facility usable "on demand," rather than waiting for specific action by a programmer or librarian.
7. If your system is large enough, provide a debugging subsystem to print values of various records, to run your "expensive" monitors on demand, and to print information about execution of the program. Make this subsystem accessable on demand if possible.
8. Don't be satisfied with fixing it in the next release!