**General Testing and Debugging Guidelines**

**Testing**

**The Testing Attitude**

As with any form of human behavior, software testing is dominated by the attitude of the tester toward the testing activity. This attitude toward testing can be summarized in a statement of the primary goals of testing. Some of the prevailing goals of testing are stated as follows:

* To demonstrate that errors are not present
* To show that intended functions are present
* To gain confidence in the software's ability to do what it is required to do

While these are admirable goals of testing, that of demonstrating the correctness of the software, the underlying attitude tends to permeate the complete testing activities resulting in test case selection and execution which tend to show that no errors are present.

One problem with these goals is that *it is virtually impossible to remove all of the errors in a non-trivial software program.* Hence the goals are unrealistic to start with. Another problem is that although a program may perform all of its intended functions, it may still contain errors in that it also performs unintended functions.

*When the prevailing testing attitude is to show that no errors are present,  
the likelihood of finding an error is greatly decreased.*

A much more productive goal of testing is the following:

*To discover what errors are present in the software*

This goal assumes that errors are present in the software, an assumption which is true for virtually all software and one which exhibits a much more productive attitude towards software testing, that of stressing the software to the fullest, with the goal of finding the errors. Since this goal is much more conducive to finding errors, it is also much more likely to *increase the reliability of the software.*

One way of detecting the prevailing attitude toward testing is to see how the words "successful" and "unsuccessful" are used in describing test case results. If a test case which uncovers no errors is considered to be successful, this is a sign that an unproductive attitude exists. Such a test case adds no reliability to the software and is hence largely a waste of time and energy. *A successful test case should be one that uncovers errors.* In fact, the more errors uncovered, the better the test case.

**Black-Box Testing**

Two alternate and complimentary approaches to testing are called black-box and white-box testing. Black-box testing is also called data-driven (or input/output-driven) testing. In using this approach, the tester views the program as a black box and is not concerned about the internal behavior and structure of the program. The tester is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived solely from the specifications (i.e., without taking advantage of knowledge of the internal structure of the program).

If one wishes to find all errors in the program, using this approach, the criterion is exhaustive input testing. Exhaustive input testing is the use of every possible input condition as a test case. Since this is usually impossible or impractical from an economic view point, exhaustive input testing is rarely used. In order to maximize the yield on the testing investment (i.e., maximize the number of errors found by a finite number of test cases), the white-box approach is also used.

**White-Box Testing**

Another testing approach, white-box or logic-driven structural testing, permits one to examine the internal structure of the program. In using this strategy, the tester derives test data from an examination of the program's logic and structure.

The analog to exhaustive input testing of the black-box approach is usually considered to be exhaustive path testing. That is, if one executes (via test cases) all possible paths of control flow through the program, then possibly the program can be said to be completely tested.

There are two flaws in this statement, however. One is that the number of unique logic paths through a program is astronomically large. The second flaw in the statement that exhaustive path testing means a complete test is that the path in a program could be tested, yet the program might still be loaded with errors. There are three explanations for this. The first is that an exhaustive path test in no way guarantees that a program matches its specification. Second, a program may be incorrect because of missing paths. Exhaustive path testing, of course, would not detect the absence of necessary paths. Third, an exhaustive path test might not uncover data-sensitivity errors.

Although exhaustive input testing is superior to exhaustive path testing, neither prove to be useful strategies because both are infeasible. Some way of combining elements of both black-box and white-box testing to derive reasonable, but not air-tight, testing strategy is desirable.

**Testing Guidelines**

The following set of testing guidelines are suggested by Myers [1979]. They are interesting in that most of them appear to be intuitively obvious, yet they are often overlooked.

*A necessary part of a test case is a definition of the expected output or result.*

If the expected result of a test case has not been predefined, chances are that a plausible, but erroneous, result will be interpreted as a correct result because there is a subconscious desire to see the correct result. One way of combating this is to encourage a detailed examination of all output by precisely spelling out, in advance, the expected output of the program.

*A programmer should avoid attempting to test his or her own program.*

It is extremely difficult, after a programmer has been constructive while designing and coding a program, to suddenly, overnight, change his or her perspective and attempt to form a completely destructive frame of mind toward the program. In addition, the program may contain errors due to the programmer's misunderstanding of the problem statement or specification. If this is the case, it is likely that the programmer will have the same misunderstanding when attempting to test his or her own program. This does not mean that it is impossible for a programmer to test his or her own program, because, of course, programmers have had some success in testing their programs. Rather, it implies that testing is more effective and successful if performed by another party. Note that this argument does not apply to debugging (correcting known errors); debugging is more efficiently performed by the original programmer.

*A programming organization should not test its own programs.*

This is particularly true in the latter stages of testing where the program is verified against its objective. In most environments, a programming organization or a project manager is largely measured on the ability to produce a program by a given date and for a certain cost. One reason for this is that it is easy to measure time and cost objectives, but it is extremely difficult to quantify the reliability of a program. Therefore it is difficult for a programming organization to be objective in testing its own program, because the testing process, while increasing the reliability of the program, may be viewed as decreasing the probability of meeting the schedule and cost objectives.

*Thoroughly inspect the results of each test.*

This is probably the most obvious principle, but again, it is something that is often overlooked. A significant percentage of errors that are eventually found were actually made visible by earlier test cases, but slipped by because of the failure to carefully inspect the results of those earlier test cases.

*Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions.*

There is a natural tendency, when testing a program, to concentrate on the valid and expected input conditions, at the neglect of the invalid and unexpected conditions. Hence many errors are suddenly discovered in production programs when the program is used in some new or unexpected way. Test cases representing unexpected and invalid input conditions seem to have a higher error-detection yield than do test cases for valid input conditions.

*A program must be tested to see if it does something it is not supposed to do.*

Examining a program to see if it does not do what it is supposed to do is only half the battle. The other half is seeing whether the program does what it is not supposed to do. This is simply a corollary to the previous principle. It also implies that programs must be examined for unwanted side effects.

*Avoid throw-away test cases unless the program is truly a throw-away program.*

This problem is seen most often in the use of interactive systems to test programs. A common practice is to sit at a terminal, invent test cases on the fly, and then send these test cases through the program. The major problem is that test cases represent a valuable investment that, in this environment, disappears after the testing has been completed. Whenever the program has to be tested again (e.g., after correcting an error or making an improvement), the test cases have to be reinvented. More often than not, since this reinvention requires a considerable amount of work, people tend to avoid it. Therefore, the retest of the program is rarely as rigorous as the original test, meaning that if the modification causes a previously functional part of the program to fail, this error often goes undetected.

*Do not plan a testing effort under the tacit assumption that no errors will be found.*

This is a mistake often made by project managers and is a sign of the use of the incorrect definition of testing, that is, the assumption that testing is the process of showing that the program functions correctly.

*The probability of the existence of more errors in a section of a program  
is proportional to the number of errors already found in that section.*

This counter-intuitive phenomenon at first glance makes little sense, but it is a phenomenon that has been observed in many programs. Errors seem to come in clusters, and in the typical program, some sections seem to be much more error prone than other sections. This phenomenon gives us insight or feedback in the testing process. If a particular section of a program seems to be much more error prone than other sections, then in terms of yield on our testing investment, additional testing efforts are best focused against this error-prone section.

*Testing is an extremely creative and intellectually challenging task.*

It is probably true that the creativity required in testing a large program exceeds the creativity required in designing that program, since it is impossible to test a program such that the absence of all errors can be guaranteed.

**Debugging**

**Definition of Debugging**

Debugging is that activity which is performed after executing a successful test case. Debugging consists of *determining the exact nature and location of the suspected error* and *fixing the error.*

Debugging is probably the most difficult activity in software development from a psychological point of view for the following reasons:

* Debugging is done by the person who developed the software, and it is hard for that person to acknowledge that an error was made.
* Of all the software-development activities, debugging is the most mentally taxing because of the way in which most programs are designed and because of the nature of most programming languages (i.e., the location of any error is potentially any statement in the program).
* Debugging is usually performed under a tremendous amount of pressure to fix the suspected error as quickly as possible.
* Compared to the other software-development activities, comparatively little research, literature, and formal instruction exist on the process of debugging.

Of the two aspects of debugging, *locating the error represents about 95% of the activity.* Hence, the rest of this section concentrates on the process of finding the location of an error, given a suspicion that an error exists, based on the results of a successful test case.

**Debugging by Brute Force**

The most common and least effective method of program debugging is by "brute force". It requires little thought and is the least mentally taxing of all the methods. The brute-force methods are characterized by either debugging with a memory dump; scattering print statements throughout the program, or debugging with automated debugging tools.

Using a memory dump to try to find errors suffers from the following drawbacks:

* Establishing the correspondence between storage locations and the variables in the source program is difficult.
* Massive amounts of data, most of which is irrelevant, must be dealt with.
* A dump shows only the static state of the program at only one instant in time. The dynamics of the program (i.e., state changes over time) are needed to find most errors.
* The dump is rarely produced at the exact-time of the error. Hence the dump does not show the program's state at the time of the error.
* No formal procedure exists for finding the cause of an error analyzing a storage dump.

Scattering print statements throughout the program, although often superior to the use of a dump in that it displays the dynamics of a program and allows one to examine information that is easier to read, is not much better and exhibits the following shortcomings:

* It is still largely a hit-or-miss method.
* It often results in massive amounts of data to be analyzed.
* It requires changing the program, which can mask the error, alter critical timing or introduce new errors.
* It is often too costly or even infeasible for real-time software. Debugging with automated tools also exhibits the shortcomings of hit-or-miss and massive amounts of data which mist be analyzed. The problem of changing the program however is circumvented by the use of the automated debugging tool.

The biggest problem with the brute-force methods is that they ignore the most powerful debugging tool in existence, a well trained and disciplined human brain. Myers suggests that experimental evidence, both from students and experienced programmers, shows:

* Debugging aids do not assist the debugging processes.
* In terms of the speed and accuracy of finding the error, people who use their brains rather than a set of "aids" seem to exhibit superior performance.

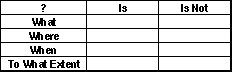
Hence, the use of brute-force methods is recommended only when all other methods fail or as a supplement to (not a substitute for) the thought processes described in the subsequent sections.

**Debugging by Induction**

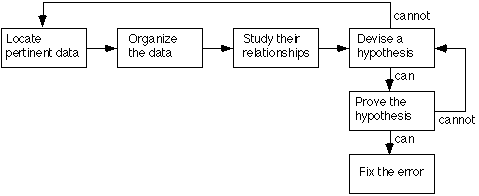
Many errors can be found by using a disciplined thought process without ever going near the computer. One such thought process is induction, where one proceeds from the particulars to the whole. By starting with the symptoms of the error, possibly in the result of one or more test cases, and looking for relationships among the symptoms, the error is often uncovered.

The induction process is illustrated in Figure 1 and described by Myers as follows:

* **Locate the pertinent data.** A major mistake made when debugging a program is failing to take account of all available data or symptoms about the problems. The first step is the enumeration of all that is known about what the program did correctly, and what it did incorrectly (i.e., the symptoms that led one to believe that an error exists). Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.
* **Organize the data.** Remembering that induction implies that one is progressing from the particulars to the general, the second step is the structuring of the pertinent data to allow one to observe patterns, of particular importance is the search for contradictions (i.e., "the errors occurs only when the pilot perform a left turn while climbing"). A particularly useful organizational technique that can be used to structure the available data is shown in the following table. The "What" boxes list the general symptoms, the "Where" boxes describe where the symptoms were observed, the "When" boxes list anything that is known about the times that the symptoms occur, and the "To What Extent" boxes describes the scope and magnitude of the symptoms. Notice the "Is" and "Is Not" columns. They describe the contradictions that may eventually lead to a hypothesis about the error.



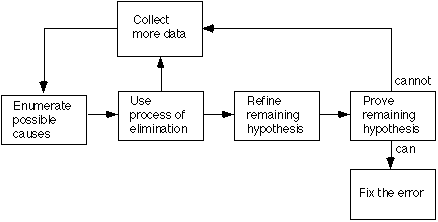
* **Devise a hypothesis.** The next steps are to study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If one cannot devise a theory, more data are necessary, possibly obtained by devising and executing additional test cases. If multiple theories seem possible, the most probable one is selected first.
* **Prove the hypothesis.** A major mistake at this point, given the pressures under which debugging is usually performed, is skipping this step by jumping to conclusions and attempting to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before proceeding. A failure to do this often results in the fixing of only a symptom of the problem, or only a portion of the problem. The hypothesis is proved by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues. If it does not, either the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present.

  
Figure 1. Inductive Debugging Process

**Debugging By Deduction**

An alternate thought process, that of deduction, is a process of proceeding from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion. This process is illustrated in Figure 2 and also described by Myers as follows:

* **Enumerate the possible causes or hypotheses.** The first step is to develop a list of all conceivable causes of the error. They need not be complete explanations; they are merely theories through which one can structure and analyze the available data.
* **Use the data to eliminate possible causes.** By a careful analysis of the data, particularly by looking for contradictions (the previous table could be used here), one attempts to eliminate all but one of the possible causes. If all are eliminated, additional data are needed (e.g., by devising additional test cases) to devise new theories. If more than one possible cause remains, the most probable cause (the prime hypothesis) is selected first.
* **Refine the remaining hypothesis.** The possible cause at this point might be correct, but it is unlikely to he specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory to something more specific.
* **Prove the remaining hypothesis.** This vital step is identical to the fourth step in the induction method.

  
Figure 2. Deductive Debugging Process

**Debugging by Backtracking**

For small programs, the method of backtracking is often used effectively in locating errors. To use this method, start at the place in the program where an incorrect result was produced and go backwards in the program one step at a time, mentally executing the program in reverse order, to derive the state (or values of all variables) of the program at the previous step. Continuing in this fashion, the error is localized between the point where the state of the program was what was expected and the first point where the state was not what was expected.

**Debugging by Testing**

The use of additional test cases is another very powerful debugging method which is often used in conjunction with the induction method to obtain information needed to generate a hypothesis and/or to prove a hypothesis and with the deduction method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis.

The test cases for debugging differ from those used for integration and testing in that they are more specific and are designed to explore a particular input domain or internal state of the program. Test cases for integration and testing tend to cover many conditions in one test, whereas test cases for debugging tend to cover only one or a very few conditions. The former are designed to detect the error in the most efficient manner whereas the latter are designed to isolate the error most efficiently.

**Debugging Guidelines (Error Locating)**

As was the case for the testing guidelines, many of these debugging guidelines are intuitively obvious, yet they often forgotten or overlooked. The following guidelines are suggested by Myers to assist in locating errors.

*Think.*

Debugging is a problem solving process. The most effective method of debugging is a mental analysis of the information associated with the error's symptoms. In efficient program debugger should be able to pinpoint most errors without going near a computer.

*If you reach an impasse, sleep on it.*

The human subconscious is a potent problem-solver. What we often refer to as inspiration is simply the subconscious mind working on a problem when the conscious mind is working on something else, such as eating, walking, or watching a movie. If you cannot locate an error in a reasonable amount of time (perhaps 30 minutes for a small program, a few hours for a large one), drop it and work on something else, since your thinking efficiency is about to collapse anyway. After "forgetting" about the problem for a while, either your subconscious mind will have solved the problem, or your conscious mind will be clear for a fresh examination of the symptoms.

*If you reach an impasse, describe the problem to someone else.*

By doing so, you will probably discover something new. In fact, it is often the case that by simply describing the problem to a good listener, you will suddenly see the solution without any assistance from the listener.

*Use debugging tools only as a second resort.*

And then, use them as an adjunct to, rather than as a substitute for, thinking. 15 noted earlier in this section, debugging tools, such as dumps and traces, represent a haphazard approach to debugging. Experiments show that people who shun such tools, even when they are debugging problems that are unfamiliar to them, tend to be more successful than people who use the tools.

*Avoid experimentation.*

Use it only as a last resort. The most common mistake made by novice debuggers is attempting to solve a problem by making experimental changes to the program. This totally haphazard approach cannot even be considered debugging; it represents an act of blind hope. Not only does it have a miniscule chance of success, but it often compounds the problem by adding new errors to the program.

**Debugging Guidelines (Error Repairing)**

The following guidelines for fixing or repairing the program after the error is located are also suggested by Myers.

*Where there is one bug, there is likely to be another.*

When one finds an error in a section of a program, the probability of the existence of another error in that section is higher. When repairing an error, examine its immediate vicinity for anything else that looks suspicious.

*Fix the error, not just a symptom of it.*

Another common failing is repairing the symptoms of the error, or just one instance of the error, rather than the error itself. If the proposed correction does not match all the clues about the error, one may be fixing only a part of the error.

*The probability of the fix being correct is not 100%.*

Tell this to someone, and of course he would agree, but tell it to someone in the process of correcting an error, and one often gets a different reaction (e.g., "Yes, in most cases, but this correction is so minor that it just has to work"). Code that is added to a program to fix an error can never be assumed correct. Statement for statement, corrections are much more error prone than the original code in the program. One implication is that error corrections must be tested, perhaps more rigorously than the original program.

*The probability of the fix being correct drops as the size of the program increases.*

Experience has shown that the ratio of errors due to incorrect fixes versus original errors increases in large programs. In one widely used large program, one of every six new errors discovered was an error in a prior correction to the program.

*Beware of the possibility that an error correction creates a new error.*

Not only does one have to worry about incorrect corrections, but one has to worry about a seemingly valid correction having an undesirable side effect, thus introducing a new error. Not only is there a probability that a fix will be invalid, but there is also a real probability that a fix will introduce a new error. One implication is that not only does the error situation have to be tested after the correction is make, but one must also perform regression testing to determine if a new error has been introduced.

*The process of error repair should put one back temporarily in the design phase.*

One should realize that error correction is a form of program design. Given the error-prone nature of corrections, common sense says that whatever procedures, methodologies, and formalism were used in the design process should also apply to the error-correction process. For instance, if the project rationalized that code inspections were desirable, then it must be doubly important that they be used after correcting an error.

*Change the source code, not the object code.*

When debugging large systems, particularly a system written in an assembly language, occasionally there is the tendency to correct an error by making an immediate change to the object code, with the intention of changing the source program later. Two problems associated with this approach are (l) it is usually a sign that "debugging by experimentation" is being practiced, and (2) the object code and source program are now out of synchronization, meaning that the error could easily surface again when the program is recompiled or reassembled.

**Error Collection and Analysis**

During each phase of software development, it is very important to categorize and collect information about software errors. Then, later on, this information can be analyzed to provide valuable feedback in terms of improving future design and testing processes.

In addition to the simple summarization of the errors and calculations of what percentage of the total errors are represented by a certain type, a more detailed analysis is needed to answer the following very important questions (also suggested by Myers):

* **When was the error made?** This question is the most difficult one to answer, because it requires a backward search through the documentation and history of the project, but it is also the most valuable question. It requires one to pinpoint the original source and time of the error. For example, the original source of the error might be discovered to be an ambiguous statement in a specification, a correction to a prior error, or a misunderstanding of an end-user requirement.
* **Who made the error?** Wouldn't it be useful to discover that 60% of the design errors were created by l of the 10 analysts, or that one programmer makes three times as many mistakes as the other programmers?
* **What was done incorrectly?** It is not sufficient to determine when and by whom each error was made; the missing link is a determination of exactly why the error occurred. Was it caused by someone's inability to write clearly? Someone's lack of education in the programming language? A typing or keypunching mistake? An invalid assumption? A failure to consider invalid input?
* **How could the error have been prevented?** What can be done differently in the next project to prevent this type of error?
* **Why wasn't the error detected earlier?** If the error is detected during a test phase, one should study why the error was not detected during earlier testing phases, code inspections, and design reviews.
* **How could the error have been detected earlier?** The answer to this is another piece of valuable feedback. How can the review and testing processes be improved to find this type of error earlier in future projects?

**Test Completion Criteria**

If one is to formalize any type of activity, such as we are trying to do with software testing, the criteria for completing the activity must be defined. This is particularly important in software testing since, except for small programs, there is virtually no way to tell when the last remaining error has been detected.

Two commonly used criteria for the end of software testing are the following:

* When the time scheduled for testing is expended.
* When all the test cases execute without errors.

The first criterion is useless because it can be satisfied by doing nothing. The second criterion is also useless because it is independent of the quality of the test cases. It also encourages one to write test cases that have a low probability of detecting errors.

Three much more useful criteria for ending software testing are discussed in the following paragraphs:

* Specific test case design
* Detection of a specified number of errors
* Use of error detection rate charts.

The best criterion is probably a combination of the three.

**Specific Test Case Design**

The first criterion is the use of specific test case design procedures. For example, module testing might be completed when the test cases which are derived from satisfying the multi-condition coverage criterion and a boundary-value analysis of the module interface execute without errors.

On the other hand, function testing might be completed when the test cases which are derived from cause-effect graphing, boundary-value analysis, and error guessing, and all resultant test cases are eventually unsuccessful.

This criterion is better then the two mentioned earlier, however it is not helpful in a test phase in which specific methodologies are not available, such as the system test phase. Also it is a subjective measurement, since there is no way to guarantee that a person has used a particular methodology (e.g., boundary-value analysis) properly and rigorously.

**Detecting a Specified Number of Errors**

The second criterion, is to state the test completion requirements in terms of the detection of some specified number of errors. For example, the completion criteria for a performance test might be defined to be the detection of 90 errors or an elapsed time of 3 months, whichever comes later.

Using this criterion requires one to estimate:

* How many errors are in a program?
* What percentage of these errors can be found?
* In what phase do the errors occur?
* In what phase are they likely to be detected?

In order to estimate the number of errors in a program, one can search for an error model based upon historical date for similar programs. Myers suggests that the number of errors that exist in typical programs at the time that coding is complete (before a code walkthrough or inspection is employed) is approximately 4-8 errors per 100 program statements.

An estimate of the percentage of errors that can be found is somewhat arbitrary and depends upon the impact of the error.

Estimating when errors are likely to occur and be detected is even mere difficult. However once this goal is established, historical date can be collected and used to help predict the time of occurrence and detection of the errors.

The real advantage to this criterion is the emphasis on detecting errors by establishing a goal and partitioning it into the phases of testing, as opposed to emphasis on the running of test cases.

**Using Error Detection Rate Charts**

Use of this criterion requires one to plot the number of errors detected as a function of time for each phase of the program. Then by looking at the shape of the error detection rate curve, one can decide whether or not to continue with one phase or go on to the next phase. The main idea is to continue a phase so long as the error detection rate is high or is increasing. When the error detection rate is declining, however, more efficiency in detection of errors may be obtained by proceeding to the next phase, where the error detection rate will again start to increase.

Graphs shown in Figure 3 (from Myers) show first an increasing rate where the phase should be continued and secondly, a decreasing rate where the phase should probably have been terminated 10% earlier.

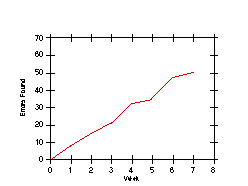
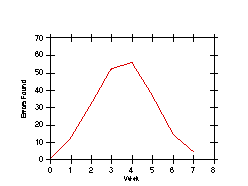
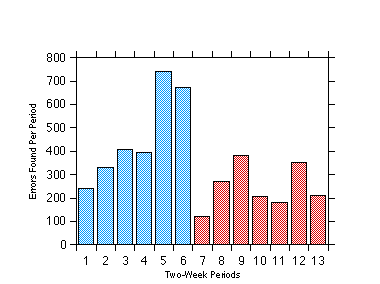
   
Figure 3. Estimating Completion by Plotting Errors Detected Per Unit Time

Figure 4 is an illustration of what happens when one fails to plot the number of errors being detected. The graph represents three testing phases of an extremely large software system; it was drawn as part of a postmortem study of the project. An obvious conclusion is that the project should not have switched to a different testing phase after period 6. During period 6, the error-detection rate was good (to a tester, the higher the rate, the better), but switching to a second phase at this point caused the error-detection rate to drop significantly.

Using the error detection rate charts in conjunction with either of the other two criteria for test completion is highly recommended.

  
Figure 4. Post-Mortem Study of the Testing Processes of a Large Project