Software Defect Isolation

Prathibha Tammana and Danny Faught
gondi@rsn.hp.com, faught@asqnet.org

"A problem well-stated is half-solved."

[STIMSON98]

1. Introduction

Dealing with defects is a necessary part of using or developing software. Dealing with them in the wrong way can waste days and weeks of your time and can delay defect repair by months or years. In our years of tenure in a software test group, we have dealt with hundreds of defects in internal tools, in software that ships to external customers, and in programs acquired from external vendors. In this paper we share some of the techniques that we have used on the job and will recount some examples of how we've applied these techniques.

We first discuss the defect isolation process then defect reporting. In the Defect Isolation section, we discuss intermittent failures and some techniques to track them down. We also talk about some of the debugging techniques we have used and describe them in detail. In the Defect Reporting section, we give some examples of bug reports and also a bug template we used to generate some internal bug reports. We conclude this paper with a brief discussion on defect tracking. Our hope is that both software developers and end users of software will benefit.

Herein, we consider "bug" a synonym of "defect".

2. Defect Isolation

In most cases, when you first see a symptom of a software defect, you don't yet have enough information to be able to identify what needs to be fixed. You must reproduce the failure more than once to gather more information. You can repeat some failures relatively easily, but others demand serious effort. To add to the complexity, you'll often come across "intermittent failures" that you won't see consistently every time you run the program.

2.1 Intermittent failures

It is important to note that even when you have intermittent failure, the software defect is always present. "There are no intermittent software errors. The problem may appear rarely, but each time the exact conditions occur, the behavior will repeat" [KANER93, p. 82]. The defect doesn't go away—only the failure is elusive.

It is these intermittent failures that cause testers and programmers headaches. In some cases, errors are so rare and dependent on real-time events that it is not possible to reproduce them with the limited equipment in the labs. In those cases, all you can do is introduce more "hooks" into the program to gather debug information and wait for the failure to appear again.

Intermittent failures are often caused by synchronization problems in parts of the code than run concurrently, which are also called "race conditions". Other intermittent failures may be caused by a complex set of conditions, requiring several specific things to happen at the same time. In [BEIZER96], the author compares software problems to FAA accident reports, where a crash is rarely attributed to only a single cause. Race conditions often appear in event-driven programs, such as programs that contain graphical interface. If you have not anticipated and dealt with the concurrency issues in your code, the program might fail [see KANER93, p. 421].

In the Defect Reporting section that follows, we mention a defect report template. The category "How to reproduce the problem" under "Detailed Description" section is of special importance when dealing with intermittent failures. Before you submit a defect report, make sure that the information you provide is sufficient for someone to use to reproduce the problem. If not, the developer might close it without ever working on it [SWTEST]. Kaner et al. [KANER93, p. 82] suggests a number of ways to make the defect more reproducible, such as looking for race conditions by slowing down the computer or testing on a slower machine, looking for user errors that may have triggered the defect (example: corrupt data file), recreating the conditions under which the defect appeared, etc.

If the intermittent failure might be caused by a synchronization problem, make sure you are using correct synchronization mechanisms between various parts of your program. A good example would be using sleep() system call instead of using signals to synchronize between two processes. If the timing dependency is not in the user program, it might be harder to trace where or why it is occurring. You might have to seek developer's help if you do not have the access to the source code or the expertise to steer your way through it.

Recreating conditions under which the defect had originally appeared may be difficult to achieve. At one time, we had reports of some hard-to-reproduce defects that were inherited from an older version of our operating system. The product that originally tripped over the defect was not supported on the current OS any more. To recreate the original conditions, the test group developed a stress test suite that mimicked the load that the original program subjected the system to. This suite had the additional benefit that it was designed to be much easier to debug than the original product and it was very successful in reproducing some of the defects.

2.2 Debugging Techniques

Even when a defect appears consistently, locating the problem may not be simple, especially when it involves thousands of lines of code and complex logic. In this section, we describe some of the techniques we have used successfully to simplify debugging.

First we must emphasize that you need an open mind. Acting on incorrect assumptions can cause you to waste vast amounts of time and resources. A training class on quality principles warns against slipping into "genius mode", where the solution to the problem seems to be blatantly "obvious". This is dangerous when you have not examined the problem in any detail and have not looked at all possible solutions. When dealing with software, you will often find that the symptom and the defect are far removed. In "Software *Is* Different" [BEIZER96], Boris Beizer presented this analogy to the defect locality problem: you have trouble tuning your car radio, and as a result your right rear tire blows out. This sounds ridiculous in terms of hardware, but the equivalent situation occurs on a regular basis with software. So be open to any possibility, no matter how unlikely.

Before digging into a problem, make sure you're taking advantage of the available tools. Scour the documentation to see if the program or application supports any kind of debug or verbose mode. Sometimes these modes are not documented, so check with the support staff, or developers if available, if you cannot find documentation for a debug mode. Make sure you are checking all the applicable log files. If you're working with a programming language, get a good symbolic debugger. An investment in obtaining and learning how to use a debugger will pay for itself many times over.

While debuggers and debug output are very useful, just having volumes of information available isn't enough. You have to zero in on the cause of the problem before you can single-step through a large program. When facing megabytes of debug output, you have to find the point where the problem first occurs, and trace it back to the source of the problem. You may have to conduct experiments with multiple runs of the software in slightly different scenarios, and compare the output for each of them. Be sure to save each output file along with the exact details of how you ran the program each time. By referring back to these details, you will be able to narrow down the possible causes with each experiment.

When you find a place where the problem is manifested, you then need to work backwards to find out how the program got into that state. Most debuggers do not allow you to roll back the clock to see previous program states. So you have to start the program over, this time with more knowledge of when to start looking for trouble and where to look for it. Keep careful notes on the results of each run. If you remember some details incorrectly, you may waste hours on a wild goose chase based on an incorrect assumption.

With further runs from the beginning, you'll be able to track the problem further back toward the source. As you find one place where everything is working, and another place not too much later in the program flow where the program state is incorrect, you will then be able to single-step though that portion of the program a few times and pinpoint the location of the defect.

Sometimes you have no idea where to start looking. Let us say you are getting a cryptic error message and you can't tell where in the program it is coming from. At least one debugger we know of, has a trace mode, sort of an automatic single-step. You can turn on this mode, and continue running the program. You will get a stream of output as each line of code is executed.

Unless you are in the unfortunate situation in which the error is not reproduced when you are using the debugger, the error message will appear in the output right by the offending line of code. You may have to use some other methods to further isolate the defect first, because you may get thousands of lines of output before the error shows up. It will be helpful to use some method, such as "script" utility in UNIX$^{(R)}$[1], to capture the output in a file so you can search through it later. Be careful if you use a capture mechanism that does not preserve the proper mixing of standard output and error output.

2.3 Binary Defect Search

There will probably be phases of the defect isolation process where a debugger will not be effective, or maybe you simply will not be given the resources to obtain a decent debugger. One technique that we have used is patterned after the algorithm for a binary search. We call it “Binary Defect Search”, for lack of a better name. Binary defect search is very powerful. It can be applied to any kind of program code, and if you stretch your imagination, even to data files such as start-up files or to word processing documents in some cases.

Here is a basic application of the binary defect search—first scatter a few statements fairly evenly throughout your code that print out a unique and easily identifiable message along with any information that is needed to help detect the problem. Use your imagination, or something like ***debug #1***, ***debug #2***, etc. Then run the program until you have reproduced the defect. Make a note of the last debug message you saw. You then know that the defect lies between the location of that message and the next message that the program would have encountered. At this point, you may remove the extra print statements that are outside the problem area, unless you think the failure may be manifested at different points during execution.

Now iterate the same process, this time putting the print statements more densely throughout the code section you have isolated. It should only take a handful of iterations of this process to locate the precise line of code where the problem first shows itself. As you work on gaining more information about the failure, you may need to print out more information about the program state, such as the value of a loop iteration variable, if the defect is contained in a loop.

Here is one successful example of a binary defect search, on a defect associated with a very non-obvious piece of data. We had a problem with a software installation tool which was incorrectly stating that the software we were wanting to install was not valid on the current operating system. At first we had a vague idea of what might be the problem, but did not make any progress until someone decided to use a wildcard in the field that defines the valid operating systems. This sufficed as a workaround, but still did not help much with isolating the problem. But then, on a hunch, we tried a more sophisticated "character class" wildcard, like "[a-m]" and "[0-9]". By narrowing down the character classes, we were able to do a binary search through the ASCII character set, and with only a handful of iterations we determined that the installation tool was wanting the string ":32" at the end of the operating system name. After that, a few queries around our group led us to the source of the problem very quickly. Without the very specific details on what data the program was looking for, it would have taken days or weeks to make any progress.

1. UNIX$^{(R)}$ is a registered trademark of The Open Group.

2.4 Simplification Techniques

Once you have a scenario for reproducing the defect, the next step is simplification. The simpler your scenario is, the easier it will be to find the source of the problem and fix it. Also, some of the things you used to reproduce the problem may be proprietary and you may not want to share them with anyone else. Whoever you report the defect to will appreciate a "one-liner" that demonstrates the problem rather than the thousand line monstrosity that it came from originally. It will also be helpful because simplification usually reduces the time it takes to reproduce a failure, sometimes dramatically. Reducing an eight-hour scenario to five seconds of execution that still reproduces the bug is not unusual.

Start by simplifying the environment that you're running in. If command line arguments are involved, try simplifying them. Remove optional arguments. Change the remaining ones to see if a particular option is needed in order to reproduce the problem. Look for other parts of the environment that could be simplified. For example, try removing each of the environment variables. You may find out that some of them do affect the program and should be present in order to reproduce the bug, and this is very important information to have.

2.4.1 Bottom-up approach

If you have a hunch about what the particulars of the defect may be, try a bottom-up approach first. Go ahead and create a very minimal environment and see if it reproduces the defect. If not, add a few portions of the original environment back in, and try again.

If you do not find the problem very quickly, you should try the top-down approach instead. Bottom-up is frustrating and non-productive beyond a few iterations. There may be a few very specific parts of the original scenario that are needed, and they will probably be thrown out in your quest to get a minimal scenario right away. In the rare case that your first hunch is correct, though, it can be a good shortcut.

2.4.2 Top-down approach

To use the top-down technique, start with the full scenario that you have defined so far. Remove chunks of code or data in order to simplify. (We will talk in terms of reducing a body of program code, but this can also apply to a set of data, environmental elements, or steps in a scenario.) This might be easier to achieve if the code is not very interdependent. Keep the binary defect search technique in mind as you organize your attack on the code. Work only with a copy, of course; do not touch the original code.

Extract the easy parts first, like all the code after the point where the failure occurs. Consider using non-traditional constructs such as "goto" or moving the code that triggers the problem earlier in the program. After each step of extracting code, verify the defect is still reproduced. If not, then the section of code you just extracted was the culprit and you are almost done—see which part of the extracted code is the crucial trigger (see that binary search here?). If the defect continues to appear, go back and extract more pieces of code.

While you use this technique, remember to save all the intermediate versions of code you generate, even if you only changed one character. Very subtle changes can affect the behavior of the program and you may not be able to remember all the changes you have made. At the least, save the last version of the code that reproduced the problem, and the last version that didn't. You'll always know that the problem lies somewhere in between. You can comment the code instead of deleting it in order to go back to previous versions more easily.

Consider that you might be dealing with more than one defect. Watch for subtle changes in behavior that indicate that you are skipping over one defect but are still hitting others.

2.4.3 Final stage simplification

Once you have isolated the defect to a few steps or a few lines of code, alter the order, even if order does not seem to matter. Try using similar or equivalent mechanisms to do a particular task. Continue to squeeze out parts of it and see if the behavior changes—but be wary of removing a part of the environment that must be there for the software to work. This final stage of simplification can be very valuable with only a small bit of extra effort. It will either give evidence that the problem is more prevalent than your example shows or it will provide more information towards narrowing down the location of the defect.

3. Defect Reporting

In many cases, when you find a defect, it is someone else's responsibility to fix it. Given that "The point of writing problem reports is to get bugs fixed" [KANER93, p. 65], you must create a defect report such that someone will have all the information and motivation they need to fix the defect. This section highlights a few important characteristics of a good defect report. Even if you will be fixing the defect yourself, if you do not start working on it immediately, you will need to record all the relevant information so you can come back to it later.

In high level summaries of defect reports that are often generated, you may have noticed that the entire problem description is not included. It is usually the first few lines of the description that make it to the summary. Therefore, it is important that you summarize the problem in a couple of lines at the very beginning of the problem description section in the defect report. Kaner et al. [KANER93, p. 69] actually suggests using two different fields—Problem Summary and Problem Description.

The following description extracted from a defect report would be considered "bad" because it not only fails to summarize the problem but also fails to report any details useful to reproduce the problem. The developer who gets assigned to work on the problem that this defect report refers to has to either contact the person who submitted it to gather additional details or he has to try and reproduce the problem himself.

While running the utility tests 16 way parallel the following test failed: sh/sh041

A better problem summary for the above report may have been: "Test sh/sh041 fails because of a race condition in the parent and child processes". You can save other details, such as when and where it occurs, for the problem description section. We also strongly recommend that you "explain why you think this is a problem, unless it is obvious" [KANER93, p. 70]. Just a statement of fact might lead to a "So, what?" response. Consider the following description from a defect report:

The FRAG_IN_USE flags in the buffer cache are not protected when modified.

The author of the above defect report failed to mention why this is a problem. Unless the consequence is visible, the person who reads through the report may not judge how important the problem is and how soon it needs to be fixed. The more serious consequence a defect has, the more attention it is likely to get. "Find the most serious consequence of the defect in order to boost everyone's interest in fixing it" [KANER93, p. 77]. You should also provide supporting information showing why the current behavior is incorrect, for example, the behavior does not match the documentation, or the product does not meet your requirements, etc.

The following is the template that we have used to report defects against test suites. You will notice that we have separated the Description section into categories so as to extract all the necessary details.

Problem: ***<State the problem in a couple of sentences>***

Impact: ***<Explain why you think it a problem>***

Detailed Description:

Testcase/suite/tool	**:*<Path to the test case/source file>***
OS version	**:*<OS you ran the test on>***
Utility version	**:*<Utility you ran the test against>***
State of the machine	**:*<Describe the state of the machine after test was run>***
How to reproduce the problem	**:*<If lengthy, please use comments section>***
Other details	**:*<Any other details that could be important to fix/analyze the failure>***

Proposed Solution **:*<Optional>***

Include as much information as possible in the details section. Use attachments if necessary. Sometimes we have seen defect reports that included the entire panic signature on the console or a two page long debug output. No detail would be considered minor in this section. Apart from being "understandable, reproducible, legible and non-judgemental" [KANER93, p. 74], a defect report should also be "simple", which means it should describe only one problem. Combining two or more defects in one report will often lead to just one of the defects getting fixed. Or at best, you will not know when any of the individual defects are fixed until all of them are fixed.

However, sometimes you may want to write higher level defect reports. Maybe one functional area of the software is badly broken and you know the development team is actively redesigning it. It might make sense to lump all of the problems in one report, saving yourself the effort of detailing all of the problems. You can refer back to this report later to make sure all of the problems were fixed, but because the functionality was redesigned, it is likely to have an entirely new set of problems.

Sometimes you cannot tell whether multiple symptoms are caused by a single defect or not. If there are just a few such symptoms, report them as separate defects. Be sure to mention in each report the existence of the others—the developers will greatly appreciate the ability to check progress on the other reports to see how closely they are really related. But if you have a long list of similar symptoms, it is too much effort to report all of them individually. If they turn out to have a common cause, you will lose credibility with the developers, as they will duplicate effort and eventually go through the hassle of merging them all together.

In all cases that more than one symptom is mentioned in a defect report, you have a greater responsibility to follow up to see whether the problems were all resolved adequately.

The template mentioned above has a "Proposed Solution" category. Kaner et al. recommends suggesting a fix if you have any ideas [KANER93, p70]. In general, this is a very good idea. But you need to judge how the suggestion will be received. There have been cases where the developer didn't like the suggestion, so he closed the defect report and ignored the problem entirely. Separate the description of the problem from the suggested solution, giving more visibility to the problem description. Consider who will be reading the report, and what result you want, when deciding to suggest a solution or not.

You may find some functionality that does not match the documentation. If you just want to be sure that the documentation matches the actual functionality, do not specify whether the documentation or the behavior should be brought in line with each other. However, if you really do not want the current behavior to change, be explicit about requesting that the documentation be changed, or, if you really want the functionality to be fixed, do not leave the possibility open for the documentation to be simply revised.

3.1 Case Studies

The psychological aspects of defect reporting are at least as important as the technical concerns. For example, we once found a defect in an incarnation of the "groups" utility in UNIX and duly reported it to the vendor. The response from the vendor was that maybe they would have it fixed in the next major release, unless we could prove that it violated some standard. We couldn't convince the support engineer to do anything further. Since several people we worked with were getting constantly frustrated by the defect, we decided to try to build a better case.

Fortunately, we had access to the source code of UNIX, SVr4, as well as the sources for the vendor's version. By comparing those sources and the observed behavior on BSD systems, we were able to make educated guesses about how the vendor's code had evolved, and exactly why it was broken. We wrote up our observations, including direct references to the vendor's source code for the groups utility, and showed exactly which line in the "login" utility needed to be fixed in order to make the groups utility work. A patch appeared within a few days.

Granted, this is an extreme case. Having access to the source code is a luxury that is often not available. The point is that the defect didn't change between the two different situations—the first case where we reported only a few symptoms, and the second case where we pointed to the culprit which was far removed from where the symptoms appeared. The response from the vendor was very different in each case. Each time you report a defect, keep in mind that you may have within your control a wide array of possible outcomes.

Another interesting example was a defect we found in a test harness developed by a group internal to the company. The test harness uses a declarative programming language. From thousands of lines of such code, we isolated the defect down to a minimal but rather bizarre sequence of just six lines. We turned on debugging output and gained more information about the cause of the defect. We made further unsuccessful attempts at isolation, and tried changing the order of some of the lines of code, and gained even more data. We did everything but search through the source code. We reported everything we knew about the bug, and it was fixed on the same day, with only minutes of effort on the part of the developer.

Did we go to the right level of detail? We had access to the source, but the developer knew more about the source and was able to track down the defect faster than we could have. Did we put too much effort into it? Maybe. The developer may have been able to track down the problem with less data, with less total combined time between his effort and ours. But the developer was very appreciative of the high level of detail in the defect report, and the result was that future defect reports we sent were treated seriously and dealt with promptly. Having a good defect reporting reputation is very important, while a bad reputation can be devastating. The level of effort you should put into your defect reporting depends on your relationship with the people you're reporting the defect to and how badly you want the problem fixed.

3.2 A Note on Enhancement Requests

If you are not reporting a broken piece of functionality, but rather asking for a new or better feature, you can submit "enhancement requests". Be careful when you are submitting an enhancement request, since they are often given very low priority. A request is generally considered an enhancement request if the current system is working as documented. A much higher priority is given to fixing things that are easily seen as broken. Enhancement requests may be identified in a defect database by giving them the lowest possible severity rating, or there may be a separate field identifying the record as an enhancement. In either case, they are often low priority and are even commonly left out of the defect metrics entirely.

You might consider a particular enhancement request as a higher priority than some of the defect reports. For example, you may submit a minor defect report against some feature that you rarely use, and you may also have an enhancement request open, asking for a new feature that you urgently need. For this reason, we recommend that development groups track enhancement requests along the same priority spectrum as defect reports. You can use a separate field to indicate an enhancement request, or since enhancement requests and defect reports are both requesting changes to the product, do not track them separately at all.

As an end user, you have no control over how enhancement request are tracked internally by the vendor. Keep this in mind as you frame your report. If it is important to you, submit it as a defect report, even if there is a way to identify it as an enhancement request. To you, since the software is not doing what you need it to do, it is a defect. Just as you do for defect reports, try to find the most serious consequence of the lack of this feature. For example, tell them if it may affect your future purchases and renewal of the support contract, and try to present evidence that lots of other customers also want the proposed feature. Of course, we do not advocate disguising enhancement requests maliciously—you may end up diverting resources away from other defects you have sent that you consider a higher priority, and you might damage your credibility.

3.3. Defect Tracking

We would like to conclude by stressing the importance of a good defect tracking system. Kaner's book [KANER93, chapter 6] deals with this topic in detail. Also, see the Problem Management Tools FAQ [see FAQ in References]. A defect tracking system that lets the user query the defect database is useful not only to generate summary reports, but also to track the status and the progress of a project that's underway. A good defect tracking system should also have a mechanism to capture the work that has already been done on a particular defect. This feature is especially useful when there is a team of people working together on a particular defect report or if the work has changed hands.

When you are reporting defects to others, judge how much you trust their defect tracking system. It is often wise to keep your own records. We have dealt with vendors who never followed up on defects we reported, and when we called to determine the status, even the reports had expired from their database. So we recommend keeping careful records of everything you report to someone else, along with the date and name of each contact you have, and any status and decisions that were made along the way.

If you're in a situation where you simply can't get enough information to report a defect, or you just can't find where the defect is in the code, there is one last ditch effort. Embark on a general quality improvement effort. Use new tools and unused options on your existing tools to find potential problems in your code. You'll find defects you didn't know you had, and perhaps you'll catch that one defect that's been bugging you along the way. For that matter, you shouldn't wait for a last ditch effort to put this practice in place.

We hope that you and the vendors you deal with are hard at work to institute processes to prevent software defects. But to the extent that you have to battle against software problems, hopefully the information in this paper helped to make the process less painful.

4. References

BEIZER96	Boris Beizer, "Software *Is* Different" Conference Proceedings, 9th International Software Quality Week, San Francisco, CA, 1996.
FAQ	Dave Eaton, ed., *Problem Management Tools* FAQ. Posted to the comp.software.config-mgmt newsgroup and available on the web at http://www.iac.honeywell.com/Pub/Tech/CM/index.html
KANER93	Cem Kaner, Jack Falk, Hung Quoc Nguyen, *Testing Computer Software*, 2nd edition.

	Boston: International Thomson Computer Press, 1993. ISBN 1-85032-847-1.
STIMSON98	Carl Stimson, "The Quality Improvement Story", presentation to the Dallas section of the American Society for Quality, February 26, 1998.
SWTEST	Discussion on the swtest-discuss mailing list on February 17, 1998, subject "Better testing/diagnosing", various authors. Message IDs: 34E94E03.3A4EF320 34E977A9.3BEA60A9 3.0.3.32.19980217045605.03608dbc 21903CE71B04D1118798006097CF2339255931 3.0.32.19980217084310.00f7762c 4060E0842A0AD111B20D0060970672A308FEFD (To subscribe and access the archives, send mail to swtest-discuss-request@rstcorp.com with "subscribe swtest-discuss" in the body of the message.)