THE ART, SCIENCE, AND PSYCHOLOGY (ASP) OF DEBUGGING

Russell E. McMahon[1],

Abstract *-- As any computer language teacher knows it doesn't take long before a student produces his/her first bug. Therefore, it is not only important to teach students good programming techniques, but also, how to debug. In a lecture it is important that debugging is addressed early and in a lab situation it is imperative that an instructor can quickly debug a student's program. A student who has no concept of debugging will likely have a difficult time completely homework assignments on time. A student's failure to successfully debug a program can also results in the failure of the concept being adequately learned and a lot of frustration on both the learner's part and the teacher's part. Students need to understand that their own perception of their code is as important as knowing how to code. Ideas on how to conduct lectures, labs, homework, and tests will be given.*

Index Terms -- computer programming, debugging, problem solving, programming

1. Introduction

Ever since Grace Hopper "debugged" the Mark II in the 1940s this term has been an integral part of computer programming. Debugging skills are important and its mastery is necessary for anyone who plans on a career in the Information Technology (IT) field. Debugging is a form of problem solving that has three aspects to it: an art, a science, and a psychology. Grace Hopper popularized the term "debugging" in the computing field on that fateful day in 1945 when Howard Aiken (her boss) asked her what she was doing. Her bug was a real one, a moth to be exact[1].

If you want your students to both enjoy coding and be successful at programming, it is important that they learn how to debug quickly and thus, be able to concentrate on the concept that is currently being taught. Often times, it is not the concept that is the problem, but those "darn bugs." I even put together a bemusing PowerPoint slide show entitled "Bugs come in all shapes and sizes" to try and get students to think about what kinds of bugs they create and how to find them.

In many ways you are a coach preparing your students for that day when they will indeed have to debug a system that a client needs and all eyes are focused on them. This comes only with practice, patience, and a lot of hard work on the teacher's part and the students' part.

2. Types of Bugs

Basically, there are four kinds of programming bugs: syntactic, run-time, logic, and design. The first three need to be taught in detail to beginning programming students. Design bugs come later during a systems analysis and design course.

Syntactic errors are failures to use the correct grammar of your language and are caught immediately by the compiler (or interpreter) or even beforehand by a "background" syntactic checker of many modern Integrated Development Environments (IDEs). They are usually easy to fix and in general present the least amount of problems for students (provided they are paying attention).

Run-time errors can be either syntactic errors not identified by the compiler ahead of time or errors that result in the program crashing. Performing an "illegal" operation like division by zero or moving string data into a numeric field are example of these. The compiler is unable to catch these kinds of errors because the values of the variables are unknown until run-time. These errors cause your program to throw an exception, which a programmer may want to handle programmatically.

Most beginning students spend their time here when it comes to debugging. Part of the problem is that students tend to write too much code before they test it especially, if it compiles clean. Therefore, the instructor should also spend more time teaching how to deal with these kinds of bugs.

Logic errors can be from problems with the design of the application to assumptions being made about what is acceptable output. The algorithm used may be the wrong one or something as simple as not testing for all the possible values of a variable. These kinds of bugs required a good testing plan to be put in place and are the hardest to detect and find. Often times these are created as a result of poor design.

Logic errors can lie dormant for a long time before they are detected and if they are not discovered soon enough, their output becomes they may become the accepted answer. This is why companies need to spend a lot of time testing an application before it goes to production (and retesting it after it goes to production).

Logic errors are hard for many students because they don't take the time to verifying the results. As with the run-time error mentality, if students get a program to run and it looks pretty good then it is OK to turn in for credit. The best

[1] Russell E. McMaon, University of Cincinnati, ML 0103, 2220 Victory Parkway, Cincinnati, Ohio 45206, russ.mcmahon@uc.edu

way to teach this is to give students a completed program and have them develop and implement a testing plan.

Design errors occur at the start of the development cycle and a poorly designed program is doomed to failure. However, this topic is more suited for a systems analysis course, advanced programming techniques course, or software engineering course. A good design does not mean logic errors won't exist. Testing is still a very important aspect of enterprise development. In a beginning programming course, design errors are not as big since most programs written are fairly short. As a student progresses, both testing and good design become a key aspects of the system development life cycle.

3. Research

Recent studies indicate that programmers spend between 50 – 80% of their time on debugging. If these numbers are only half right this would still reflect an enormous amount of time and energy being spent on something that is obviously a major problem with application development[2]-[3]. In a recent study by the National Institute for Standards and Technology (NIST) it was found that software errors cost the US economy roughly $59.9 USD annually. This report estimates that more than half the bugs are not found until well into the development cycle or after the product has been sold or put into production[4].

In his book Code Complete, Steve McConnell points out that the industry average for code production is 8-20 lines of correct code per day. He further points out that there are 15-50 errors per 1000 lines of delivered code. Mr. McConnell recommends that programmers learn how to code more defensively[5]. Since most applications go into the millions lines of code these bugs can become an enormous drain on the programmers supporting them and the companies using them

Marc Eisenstadt of the Open University wrote a paper entitled "My hairiest bug war stories." He collected antidotal information from programmers who related their worse bug nightmares. What is especially interesting is the number of programmers who inherited code from someone else and then were expected to complete the project. Many of these programmers complained that the bugs were there when they got the code.[6]

In a paper entitled "The Debugging Scandal and What to Do About It", Henry Lieberman from MIT states "debugging is still, as it was thirty years ago, largely a matter of trial and error." Part of the problem, Lieberman contents is the "lack of attention to improving the tools for debugging programs."[7] There is also a lack of attention on the instructors' part of teaching students how to use a state-of-the art debugging tool. Debugging is a skill that is not normally taught instead, many students learn it on their own (through "osmosis").

There is currently a lot of excitement about Agile or Extreme Programming with its approach to producing nearly bugless production code. Emphasis is placed on shorter development cycles and lots of ongoing testing and debugging as the system progresses not just after the system is finished[9].

The concepts of debugging and testing need to be taught. Testing is not the same as debugging, but it can show the presence of errors in the code and both skills go hand-in-hand. Knowing one has a bug in his/her program is a start, but it still remains for the programmer to find the bug and correct it. At the beginning level of programming courses, students need to be taught how test and debug their programs and they need to understand that this is an art, a science, and most importantly a frame of mind (psychology) in terms of what one thinks the problem is and how one views their own code.

A special kind of logic error is called the "Heisenbug" name after the Heisenberg Uncertainty Principle. This kind of bug appears in an actual test run of the program, but disappears when run within the debugger. Sometimes it seems to magically appear and then just as quickly disappears. In general, if you don't know what causes, it you have a possible Heisenbug.

4. Art

Debugging is described as a black art or a secret art. It is more of a creative art, which requires the practitioner to use that creative, non-logical part of the brain to track down the bug. The more creative part of our mind is sometimes needed to know where to look for the bug and the willingness to look in the exact opposite place. This form of debugging is where hunches are sometimes the way a programmer finally finds the bug[8].

Debugging will always be an art because of the constant changes in computer languages prevents someone from thoroughly knowing a language. (Many programmers joke that if you know the language intimately, then you are three releases behind.) When switching from one language to a new one there is a new syntax to learn, a new set of compiler rules, a new set of bugs, and sometimes a new paradigm about which to learn (like going from procedure programming to object-oriented programming). All computer languages are different and all have a different set of problems associated with them. Debugging SQL is different from debugging HTML which is different from debugging C# but, it is possible to use all three in the same application.

Debugging is a creative art because even the best programmer knows that sometimes what appears as the obvious location of the bug is really just where it manifests itself and the real bug lays somewhere else. It may even be in the design of the application itself.

Debugging is a learned art. This means learning how to use available tools to see what's going on in your code. This leads to the science of debugging.

5. Science

Debugging is a science. There is a set of rules for general debugging that follows the scientific method. A hypothesis is formed on what the caused the error and is then tested. If the assertion is true then the bug can be fixed quickly and the student is able to move forward with the program. Otherwise, the student must reformulate his/her hypothesis and test again.

There is a methodology that can be applied in tracking down the bug and fixing it. Back in the older days of programming (where punched cards and GOTOs ruled the world) there were no real easy ways to debug programs except to use some sort of "print" method to display the values of the variables you were interested in seeing and doing a lot of desk-checking. Desk-checking is still a good method to use when all else fails as it requires you to "play" computer and really think about what it is your code is doing.

Debugging requires good analytical skills. Students who are good in math and science tend to be good in debugging. Knowing where to start debugging is the challenge. This can, however, be taught and modeled by the teacher. Code isolation and verification is a good technique to use when presented with a large program that does not work. Comment out sections of code until you are able to narrow the bug down to the offending line of code.

Teach program develop through the use of an IDE. They are very rich in tools that make it quicker and easier to debug. Plus, it is very likely that students would be using such a tool on the job. You can set breakpoints, step through your code, watch the values of your variables, and switch over to the assembled code if you need a closer look at what is going on.

Finally, the science of debugging requires students to read the documentation on any errors they receive. The documentation does not always help in explaining what the programmer did wrong but, it can serve as a guide by telling you what areas of your code you should check. If this is the first time a student of mine has seen this particular error, I expect him/her to use the help facility.

6. Psychology

Sometimes you convince yourself that what was written cannot be wrong and therefore, the error is either elsewhere in the code or in the system outside of your code. The error is staring you right in the face, but you refuse to accept it. Hence the saying: "Programmer know thy self and your compiler." Students need to understand that sometimes they will make mistakes and not be able to "see" them because they have convinced themselves that the code in question is correct. Compilers are constantly being updated and sometimes code that compiled cleanly under an older version fails to recompile in the newer one.

There are three basic questions all coders should ask themselves when they encounter a bug. What sort of errors have I made in the past? What kinds of habits have I picked up that lead to this bug? How can I change these habits? Sometimes the only difference between my students and myself is that I know what kinds of errors I can make and the habits that lead up to this error and they don't.

Students need to understand the psychology of debugging because the mind can be the roadblock in the way of successfully finishing an assigned program in a timely manner or frustrating oneself for hours on end. Getting students to catch their errors sooner instead of later is important. Most of my students agree that some of the errors they have made are because they convinced themselves that the line of code in question or algorithm is correct and therefore the bug must be somewhere else in the code or that the OS, network, IDE, compiler, or some other fluke of nature is the cause of their problem.

One needs to adjust his/her mindset when debugging. First and foremost all programmers need to accept that they will create bugs. This sounds simple enough, but many people have trouble with this. Second, one must feel comfortable enough in his/her programming skills such that bugs can and will be found and appropriately handled. Instilling confidence in your students is very important. They must feel that the bugs they encounter are not insurmountable and that with a little more persistence and work they will solve the problem. Students can become so convinced that something else was the cause of the error that even when it was shown, they still wanted to defend their invalid code. (This has happened to me a number of times.)

Remember what's in the book is not always correct. Most programming books are one release behind the actual product and often times the sample code has not been thoroughly tested. What may have worked before may not work with a newer release of the product. Teach your students to treat all information with caution. This is difficult for beginning programming students because they expect the author and instructor to be infallible experts and they really have no way of knowing what is correct or incorrect.

Another problem is switching to new version of a language or to a whole new language (such as going from Visual Basic 6 to Visual Basic.NET). Although the same errors like dividing by zero will generate an error, the error messages do change (sometimes for the better). (In C# it is possible to execute unchecked code that would normally cause an overflow error. This does not stop the code from executing or from returning results back.)

7. Lecture

In a lecture it is important that good coding techniques be taught. Code that is well designed, written, and structured is less prone to bugs to begin with and is easier to debug later. Students learn early on that I will not help them with a

programming problem if their code does not follow good coding standards. Debugging is addressed early in my programming courses with students being introduced to the more common errors of the language they are learning.

Introduce bugs in small doses and give students the opportunity to create their own bugs so they become familiar with the error messages and what they really mean. Teach students to fix bugs before continuing and to test their code every step of the way. Good programming habits will carry a student far. This includes showing them how to use the available help facilities.

Start with an explanation on debugging by asking students if they all know what ASSUME means. Most know what is meant here (don't make an ASS out of yoU and ME). Next tell your students to stop making preconceived ideas about what the caused the error. Many times it is this notion that the error can't be where it appears that causes them problems.

Teach students how to write stub program code and then how to enhance the various stubs as they go alone checking each step before proceeding to the next. Have students identify the important parts of the program and start out small. Teach student how to desk-check. Although, this is not as necessary as in the past, it still is useful and teaches the students how to "think like a computer."

Writing clean code is not so much a problem when students have plenty of time to design/write/test/debug their code as much as it is during a test. Many students believe that writing pseudo-code on a test is better than nothing at all. By pseudo-code I mean the program did not compile, but instead of fixing the problem (because the student could not do so) the student continues to write code pass the problem area. (Damn the torpedoes, full speed ahead) In some cases students do not even compile their code until they are all done with the problem. These habits can be hard to break, but if the rule of "no compile -- no points" is applied, it tends to stop most students from handing in incomplete code.

Although an IDE usually has a debugger built in it and will help a student find the errors, this does no good if the student fails to understand how the program is functioning in the first place. This goes back to the psychology of debugging. The data given back by a debugger is interpreted by a student who may not really understand what the true meaning of the information returned. It is critical that students learn how to decipher the meaning correctly or else they could easily spend more time than needed to solve the problem.

Before sophisticated IDEs and debuggers the standard way of checking what a piece of code does was to insert "print" statements into the code and watch the values as the program ran. Nowadays you can insert a breakpoint in your code and a variable "watch" window will show the variables you are interested in.

8. Lab

Most of our computer-related courses have an associated lab. We feel that a lab is vital to our students' success. Students need practice debugging under a control environment. They need to be shown how programming is done in lecture and practice it in lab. This is the time where I assess a student's ability level. A student who is not very successful in lab is likely to have a hard time with any homework and subsequent tests.

Just as thought of eating an insect makes most people squirm. I like to give my students a set of programs with bugs and watch them "squirm." I have built up a library of "interesting" code for use in my labs. I give them a document with code on it and they have to find the bugs first via desk-checking and later verify their results on the computer. Other times I have students work as teams and create bugs in a program and then have another team debug it. A little competition goes a long way.

I also like to have another student to debug someone else's code. This helps both students. Many times I have seen my errors as soon as a colleague of mine came to my cubicle to help me debug. If those two students can't solve the problem, I may assign a third or fourth student to the bug before I get involved.

During lab time I walk around observing students' work on the computer. Occasionally, I observe a bug that a student has created. However, I do not immediately rush to aid the student instead, I choose to let him/her find it and fix it. If that student later needs my help, I am often times able to look like a magician for the speed in which I found their bug. Of course, this doesn't happen like this all the time so my students know I am fallible.

A programming course without a formal lab makes teaching a language difficult but, not impossible. Treat your labs like a typical science lab. This is the time for students to learn, interact with one another, and discover new things. Students are expected to be there and do the work assigned during that time period. Also, use the lab time to work with individual students. At the University of Cincinnati, we use Blackboard as a means in which students can communicate with their classmates and post their answers.

At the College of Applied Science we have a Programming Learning Center (PLC) where students can go for additional help. This program was started in January 2002 and has a goal of allowing our more advanced students to work with our any of our students who are having problems. Our PLC techs are paid so there is an incentive for them to delve into the problems given to them. The PLC techs can be link with a struggling student for more one-on-one help.

In a lab situation it is imperative that an instructor can quickly debug a student's program. In a beginning course a student can become too frustrated and is then unable to move forward. Even in an advanced programming course it is important that the instructor can debug the program or at

least point the student in the right direction. Failure to do this can result in the failure of the concept being adequately learned and a lot of frustration on the student's part.

9. Homework

A professor has less control in this situation so it becomes even more imperative that students learn how to design, write, debug, and test their code. A student hitting a roadblock here can be bogged down for a long time and frustration sets in sooner or later. Keep the list of programming assignments reasonable and add one or two that are very easy to write. Spend time going over possible solutions and buggy solutions.

There are students who are very good programmers and some who can hack their way through most any problem you give them. By in large however, most students are not natural born programmers and need a good foundation before they can successfully build large applications. Both homework and labs are the perfect time for students to practice what they have learned.

Homework should also give the students the opportunity to learn something new about the capabilities of the language they are using. C# for example has extension date/time functions, sort capability, data structures and other functionality already built-into its extensive class libraries. Give assignments that will allow students to learn more about these and expand beyond what you teach or what is in the book.

10. Testing

I generally give three tests a quarter (10-week, 3-quarter system). The first two have a debugging part to it. I test less on debugging as the quarter progresses, but students realize they are responsible for being able to debug their programs. You may need to occasionally help a student debug his/her program during a test. Do this when the bug is something that may not have encountered before or is too obscure to let it impede a student's progress. When you assist a student, speak loud enough so other students are made aware of this problem and can adjust accordingly.

11. Conclusion

There are plenty of articles and books on debugging in a particular computer language and these make excellent resources for learning language specific debugging techniques. Debugging is a form of problem-solving. I enjoy debugging and always have learned something new when presented by what sometimes looks like the simplest bug and it's this life-long learning, that I want to impart to my students.

Teach students how to limit their bugs by testing as they go along. Do not reward students for coding additional code beyond the initial bug. Give students programs with bugs for lab and homework. Not only discuss debugging and how it can be best achieved, but also discuss the psychology of debugging. Students will miss bugs because they are convinced that the area in question was correct and therefore the bug has to exist elsewhere. Be patience with your students. Once they become successful at debugging many of the advance programming concepts become easier to teach because students are now able to concentrate on them and they no longer have to worry about the incidental bug.

12. References

[1] *Annals of the History of Computing*, Vol. 3, No. 3 (July 1981), pp. 285-286.

[2] University of Cambridge Department of Engineering http://www-h.eng.cam.ac.uk/help/tpl/languages/debug/node1.html

[3] Thermo Galactic, "Debugging Array Basic Programs", http://www.galactic.com/Programming/articles/debugging.htm

[4] National Institute of Standards and Technology, June 28, 2002, http://www.nist.gov/public_affairs/releases/n02-10.htm

[5] McConnell, S., *Code Complete* , Microsoft Press, 1993

[6] Eisenstadt, M., "My Hairiest Bug War Stories", *Communications of the ACM*, Vol 40, No 4, April, 1997, pp 30-37

[7] Lieberman, H., "The Debugging Scandal and What to Do About It", *Communications of the ACM*, Vol 40, No 4, April, 1997, pp 30-37

[8] McMahon, R., "The Art of Debugging", *Mantis Memo*, Issue 4, 1988, Cincom Systems

[9] Extreme Programming: A Gentle Introduction, http://www.extremeprogramming.org/