Finding Bugs in your Projects

David Kieras, EECS Dept., University of Michigan

Revised 1/9/2015

Debugging vs. Testing

A bug is a defect in your program. Debugging is the art of understanding the defect well enough to be able to fix it. You find the exact defect in the code, and you change it to the correct code. But you can't fix a bug unless you know it is there.

You will normally encounter a lot of bugs while you develop your program. Such bugs are usually obvious, and so you know to fix them. But it is a mistake to believe that you will encounter all of the bugs during development. You can be fooled into thinking that your program is correct just because there are no obvious bugs crawling around on it. But at this point your code is almost always a nasty mass of ugly bugs waiting to squirm out when the conditions are right. Ugh! But to get rid of them, you first have to find them!

Testing is the art of finding the bugs hiding in your code. You make them reveal themselves so that you can fix them. Instead of waiting for your program to fail when it is put to use (or run by the grading system), you test your program in advance to flush out the bugs and make them visible.

Have the Right Attitude

When you are writing your program, you are naturally pleased and happy when it appears to work. As programmer, it is natural to be proud of your creation, and to assume that your brain-child is both beautiful and perfect. But when you test your program, you must play a different role. You try as hard as you can to get your code to fail! You are delighted when you make it fall flat on its face! As program tester, you have fulfilled your mission, your destiny, when you find a bug in your code!

Think of testing like a shoot-em-up sort of video game. Your goal is to be destructive! Can you zap your program? Can you shoot it down? Can you make it crash, give bogus output, or go crazy? You score when the program fails!

This isn't silly. Many times when buggy software is delivered, it is because the developers were too willing to believe that their product was perfect and wonderful. They didn't see the Dark Side; they couldn't stand the thought that horrible, disgusting bugs were lurking in their code, and so they didn't try to find them. But closing your eyes is no way to get the cockroaches out of the kitchen.

So the first step in testing is getting the right attitude: Your program almost certainly has bugs. It is far better if you find them and fix them, than let your customer (or the grading system) find them and penalize you. You are the winner when you find the bugs!

Code Inspection Comes First

Is your code even *trying* to do the right thing? Many bugs in code are due to just a simple oversight — you forgot part of the specifications while coding some function, so of course it won't work correctly. However, it can take a long time to flush out such errors by testing — you have to think of a test for part of the spec that you forgot about!

A better approach is to *inspect* your code and *compare* it against the specification. If the specification says that the function should do X, Y, and then Z, then look at your code — is it trying to do X, Y, and then Z? Of course, if it is, it could still be wrong, but an amazing number of bugs are due to simply leaving something out. For example, your code tries to do X and then Z, but Y got left out! Why do a lot of testing when running your eyeballs over the code and the specifications can find these problems quickly? In fact, this method is so efficient that you should try it first, before doing any form of run testing.

Look for common problems in the code. Ace bug-hunters realize that they can often catch bugs by looking at the source code itself the right way, another type of code inspection. For example, if the code is repetitious, you can figure out the pattern that should be in the code, and see if it is there every place it should be. For example, the projects in this course do error-checking of user commands following a certain pattern stated in the specifications. By checking that this pattern is present everywhere it is supposed to be, you can find and correct a lot of bugs quickly that would only be found by very thorough run-testing.

Also, you can "mentally execute" what the code would do at the extreme cases, and see if there is a possible problem there. Whenever there is any kind of iteration, you can check for whether the iteration will terminate when it should.

A good thing to check especially carefully is whether you have either initialized, or stored values in, every variable (especially pointers) before your code uses the value. This can be done by eye quickly.

Quality code is easier to inspect! Code inspection is much easier if your code is well-structured and well-written, which makes it easy to tell what the code is trying to do. Funny how quality code wins in so many ways!

How to Find Bugs with Run Testing

General Approach

Be systematic. The key is to be systematic and methodical, keeping track of what you need to check, and what you have checked, and what works, and what does not. If you just fool around at random with your program for awhile, you haven't really tested it. Most bugs are too clever for a casual check-out to find.

Be right. How do you know your program is producing the correct behavior? Don't just assume it is! Obviously, you can find a lot more bugs by checking to see if the output is really correct or not. This is especially important when numerical computations are involved. Run a check by hand or with a calculator. Don't rely on your memory to retain the details involved in a complex situation — makes notes on paper!

Be clever. You catch bugs by trying every possible relevant input, and see if the program produces the correct output for the input. If not, hurrah! You've found a bug! Of course, testing all possible inputs is not practical, so this is why you try each relevant input. What's a relevant input? One that is cleverly chosen to flush out a hiding bug, based on the specifications, the structure of the program, and

knowledge of typical coding errors. By being clever, you can find almost all of the bugs in a reasonable amount of time.

Before you Start

Prepare some kind of check list. To find and eliminate the bugs, you have to look under every stone. In other words, you have to be sure you have tested and tried every component of the program, and every type of behavior the program is supposed to perform. It is way too easy to lose track in all this detail, and thus do only a sloppy and incomplete job of bug-hunting. An easy way to make a check list is to simply mark up a hard copy of the project specifications with a highlighter pen. For example, for each program command, highlight every specific thing that the command is supposed to do. Then check each highlighted item. When you have given up trying to find a bug there, check it off and go on to another one. Make sure you've checked everything off before submitting your program.

Where to Look for Bugs

Catching bugs in small programs is easiest. Catching bugs is especially fast and easy if you test the code as you build it. Even if you have already built the code, often it is easier to flush out bugs by using a testing driver like those suggested in the project document. For example, a good way to trap bugs in functions that read and print data is to write a testing driver that simply calls the data-reading function to read in some data, and then outputs the result with the print function. You can rapidly try out many combinations of inputs.

Take advantage of well-organized code to trap bugs in individual components. There is a major reason for the rules like avoiding global variables, and the approaches of structured programming and object-oriented programming. It is to help trap bugs in individual components, making them easier to find, and discouraging them from infesting other components.

In well-organized code, you can beat the bushes in an individual program component and flush out all of the bugs hiding there. You can then be pretty confident it is clean when used elsewhere in the program.

For example, if a data-reading function has been thoroughly gone over, say with a testing driver or some command in the program, then you don't have to clean it up again in another command and then again for some other function or command. If a bug is hanging out in these other functions, chances are it is not in the data-reading function but somewhere else. You can find it much quicker!

But if you have the functionality of reading the data is duplicated in every function that needs to read data, you now have multiple places to check — you have to repeat the same testing for each place. What a waste of time!

Check every hole where a bug might hide. You can't find the bug in a certain part of the code unless the program executes that part of the code. Examine your code to find such parts — these will be associated with conditional statements like if or while. Each one is a potential bug hide-out. Can you concoct a test case that will make your program go through that part of the code? If so, you can make the bug hiding there reveal itself. You can use the debugger to verify that the code is indeed actually executed. Try each place, and check each one off a marked-up hard copy of the code if you couldn't find a bug hiding in it. The concept here is *test coverage* — do your tests cover all of the possible pathways through the code?

Bugs often hide in error detection and handling code. The project specs tell you what errors a program is supposed to detect and what the program is supposed to do about handling the error. This code is often subtle, and so is a perfect breeding ground for bugs. Try to flush them out by trying to make your program get confused, break, or crash. For example, look at what the specifications say about

each type of error in the input. Try to make that type of error without the program catching it — can you confuse your program? Can you hide the error in a bunch of distractions? If you simply can't slip something past your program, then sadly, you have to check that feature off the list of bug hiding places.

Check boundary cases. A common place for bugs to lurk is at the extremes, or boundaries, of the program's behavior. Too often, programmers look for bugs in common, or typical, conditions. But often they hang out where the inputs or conditions are at their largest or smallest values. For example, will the data-save and -restore functions work correctly if there is no data? For each function or command, identify the extreme cases that your program might screw up, and try to catch it doing so.

Check the whole program at the end. If you have followed the advice about testing individual program components, good for you! But sometimes bugs hide in how the components are connected to each other. Good program organization (decoupling) minimizes these connections, but a bug might have slipped in anyway. So when you are satisfied with your components, use some of your individual component bug traps with the program as a whole, and try to stress the component interactions. For example, if one component produces a boundary case, does the other component react properly to it?

What to do When You Find a Bug

Find the smallest or simplest test case that reveals the bug. Try to play with the inputs until you have found the simplest situation in which the bug appears. Often you will learn some important facts about the bug as you do this, but it will be simpler to isolate and fix the bug, and verify the fix, if you have a simple test case.

Squash each bug right away. If you find a bug, it is usually best to fix it right away, then go look for another bug. Of course, you might mess around some more to make sure you understand the bug, but don't waste time trying to find more bugs. Bugs have a nasty habit of interacting with each other in crazy ways. So trying to understand and deal with more than one at a time can be very confusing.

Keep track of and re-use bug traps. Keep a list of the test inputs that flush out a bug. Then when you have installed a fix, you can quickly see if you actually squashed it.

Check for bugs caused by squashing a bug. After fixing a bug, naturally you check the fix with your bug trap. But be sure to back up and also try the test inputs that originally worked correctly. This is called *regression testing* — after a fix, regress and test things that worked correctly before. Fixing code sometimes makes new bugs appear in other places; they can multiply when you kill them — like a horror movie creature. You can trap the new ones with a retry of previous checks. Before blowing this off as too much work, remember that if you have trapped a bug in an individual component, you should be able to do your regression test on just that component.