Detective Strategies for Defective Code

No matter how skilled a programmer is, no one programs perfectly the first time. Finding and fixing errors is as much a part of the programming process as typing in the program to start with. If you believe that you are the exception and that you rarely create bugs in your code, it may be that you are failing to adequately test your code!

So, what do you do when your program just doesn't work?

A) Randomly change some stuff and try again
B) Go ask for help
C) Declare the program "finished" and hope no one notices
D) Employ methodical debugging strategies until the software passes all tests

While choice D might seem the "obvious" one, it may feel out of reach if you have never been taught how to debug. If you feel like debugging is harder than programming, you may be right. The programming process and the debugging process require two separate (but overlapping) skill sets, and some will be naturally better at one part than the other. However, just like programming skills, debugging skills can be honed with some training and practice. The aim of this article is to explicitly lay out some good strategies and tried-and-true principles for debugging so that when faced with buggy code we need not feel helpless. For this article, I will give "student" debugging examples, since that is my experience as a professor. I will also talk about "testing" code. This may refer to an automatic test suite or just simply running a program. Finally, "tracing" code is the process of stopping the running of a program and examining the state of the variables. This can be done with print statements, but is usually done with the help of a debugger.

We will approach the problem of finding code bugs by donning a deerstalker hat and chewing on a pipe as we attempt to play the role of detective. In order to solve the mystery, we will need to define the crime, victim, and suspects. Then we will follow the clues and collect evidence until we have located the problem and applied a fix.

Define the Crime

The first task is to define the crime. What makes you think the code does not work? Is there a test case that fails? Is an error message given? Is the wrong answer computed? Make note of any error messages you see. They are important clues.

Categorize the bug as a syntax error, runtime error, or logic error:

- A syntax (or compiler) error is when code violates the rules of the programming language. This could be misspellings, missing punctuation, or misusing a key word in the language. Usually these are the easiest to find and fix. Most IDEs will mark exactly where they occur. The important thing to remember is to fix syntax errors from the top of the program down. Early errors can cause other lines of code to be highlighted as incorrect that are not actually wrong. Only the first error is trustworthy.
- A runtime (or fatal) error is when code is running and crashes. Once again, the IDE will help by telling you which error occurred and where. The message that you see is called a "stack trace". You should focus on the line that comes closest to the top but also refers to a line of code you wrote (not anything imported from a library.)
- A logic error is the most difficult to find and fix. This type of error occurs when the code runs but simply doesn't do the right thing. Comprehensive testing is the only way to know this "silent" bug even exists.

After categorizing the error, you should compare the *expected* behavior of your program with its *observed* behavior. Articulate each, perhaps to a friend you can recruit to be your Dr. Watson. Often in explaining the desired and observed behavior of code we gain insight into the nature of the bug (and sometimes immediately see what the problem is.)

Next, challenge your assumptions. Make sure the answer or behavior you think is in error actually is. For example, one student came saying that his program was in an infinite loop ("frozen") and that he didn't actually have any loops in the code! I began to challenge those assumptions - a quick search told me there were no loops. Next, I looked for recursion - a loop in a clever disguise! - but that didn't exist either. Finally, I challenged the notion that the program was frozen. It was not. As it turns out, the program was waiting on input from the console (and the student had failed to write a prompt.) The path to the bug was in reviewing the expected behavior of the program (it was supposed to wait for input) and challenging our assumptions about the observed behavior.

Identify the Victim

Generally speaking, when code doesn't work, there is a (more or less) clear top-level entity in the code that we can identify as wrong (which I am calling the victim). For example,

- There's a compiler error on line X.
  Line X is the victim.
- The code crashed with a null pointer exception. The line that it crashed on is:
  ```
  alistair.getIdentity();
  ```
  The "alistair" variable is the victim.
- The program was supposed to calculate an answer of "100" but instead computed the answer "101".
  The answer quantity is the victim.
- The program is stuck in an infinite loop. Now the victim may not be so clear. With the debugger we should be able to find the loop that was entered, but it may take some tracing to determine why the stop condition is not met.

The challenge of debugging is finding how the victim came to be attacked, but skipping the step of identifying the victim means that we may go on a wild goose chase, investigating entities that are unrelated. For example, one student asked for help in debugging a null pointer exception occurring on this line of code:

```
account.runBalance(checker.getTest());
```

The student had spent a lot of time ensuring that the `account` variable couldn't possibly ever be set to null. A quick look at the values of the variables at the time of the crash would have helped the student identify the `checker` variable as the actual victim.

Re-create the Crime Scene

In order to systematically debug the program we need to be able to reproduce it at will. Note the inputs and sequence of actions that led to the bug. Try to define a sequence of actions that is as *short* as possible and uses as *small inputs* as possible and that faithfully recreates the bug each time. Often I have seen students give up on debugging because they only did one massive run of a big program, which failed. The task of tracing or hand-simulating the code for a massive test isn't practical, so they were stuck. To debug, it is vital to have as small a test as possible so that at each step of the code we are sure what the expected behavior should be, and so that our list of code entities potentially causing the problem (suspects!) can start out short.

Identify the Suspects

Now that you have the crime and victim identified, you can identify the suspects: any code entity that had means and opportunity to commit the crime. In other words, identify the classes and methods that have access to the victim. If the victim is a variable, is it local? public? final? If the victim is a line of code, look at the modifiers of the method - which entities are allowed to call the method? This step underscores the importance of information hiding - the more private we keep our data and methods the smaller our list of suspects.

Use the Process of Elimination

At this point, we generally have in mind the nature of the bug, the entity that is corrupt, and the places in the code that have access to that entity. Before we begin to trace code line by line, we want to narrow down the suspect list by getting the "alibis" of the suspects. That is, use either breakpoints or print statements for each large code entity that comes into contact with the victim to find where in the overall algorithm things went wrong. If, for example, there is a method that extensively uses a "victim" variable, that method will obviously be a strong suspect. However, if at the *end* of the method a print statement shows that the variable is correct, the method can be eliminated as a suspect without doing a line-by-line trace. Personally, when I do this process I tend to pay attention to the beginnings and ends of entities (methods, loops) and I generally work backwards from the line where the error occurred.

Follow the Trail of Evidence

The process described above can localize the area of code that you should examine further. At this point it is again important to articulate the expected and observed behavior of the code at hand (which may be a different focus than the overall bug). Use the debugger or print statements to trace the code line by line, and at each step make sure the observed and expected behavior match.

Don't ignore evidence - if there is a code entity other than the victim that gets an incorrect value, pay attention. If there is an error message given, read it and make sure you know what it means. Don't skip past errors because it isn't the exact error you are looking for. Remember that computers work linearly, and an error early on can cause many problems later. That also means that fixing a bug earlier in the code can cascade into other errors going away.

Formulate a Hypothesis and Test it

Once you have a guess at what might be occurring, test it out by making one change and re-running the test. Making more than one change at a time or making unfounded changes just to see what works (known as "shotgun debugging") is a *really bad idea* because you won't know which change made the effect. Many times I have pointed out an error to a student and heard them say, "but I tried changing that an hour ago." Most likely, the student changed more than one thing at a time and the other changes re-introduced the bug.

When you make a single change, note the effect. Make sure you understand *why* the effect happened so that you can tell if the change is a step in the right direction or not. If you do not understand why the code behaves as it does, you should stop and look up implementation details about the code so that you understand it.

Rehabilitate the Offender?

While the detective work may stop once you have found the bug, it is important to be thoughtful in fixing the bug. Resist the urge to make a quick "if statement" patch. Sometimes a special case if statement is exactly what is needed, but sometimes it only fixes the problem for the exact test you ran. Running the next test will likely fail, and your process will start all over again. The key to making a good bug fix is understanding why the bug occurred in the first place and fixing the source of the error.

Take Your Success - Don't Believe the Negative Press

When you are sure that you have successfully fixed the bug in your code, it is time to celebrate. This is true regardless of the status of the rest of your code. I have often seen a student delete an error fix because the code failed more test cases with the fix than without! The number of *visible* errors in terms of failed test cases or compiler messages is not always directly related to the number of *actual* bugs in the code.

The Usual Suspects (For this section, I will focus on the Java programming language.)

You know - every time I play the game Clue, the murderer is always one of the same six people! Weird, right? When we program, we often find the same bugs crop up over and over as well. Being on the lookout for these usual suspects will make them easier to find and fix.

- *Ms. Nell X. Ception (Null Pointer Exception)*

  A null pointer exception is one of the most common types of runtime errors. In order to understand what is happening, it is important to know what "null" means. When objects are declared but not created (that is, use the word "new" for example) the variable is given a space in memory which is filled with zero bits. Suppose for example, I have these instance variables:

```
int num;
Car vroom;
```

  If I proceed to use these variables:

```
int half = num/2;
vroom.drive();
```

  The first line would use the value zero for "num" and proceed to make a "half" variable also with value zero. However, the next line trying to use the "vroom" object would give a null pointer exception because a "zero" value for an object doesn't contain information on how to perform any functions.

  Here's what a Null Pointer Exception looks like in Eclipse:

```
java.lang.NullPointerException
        at application.DFS.noNext(DFS.java:35)
        at application.SearchAlgorithm.step(SearchAlgorithm.java:113)
        at application.MazeDisplay.oneStep(MazeDisplay.java:230)
        at application.MazeDisplay.step(MazeDisplay.java:222)
        at application.MazeDisplay.lambda$0(MazeDisplay.java:67)
        at application.MazeDisplay$$Lambda$108/973756543.handle(Unknown Source)
        at com.sun.scenario.animation.shared.TimelineClipCore.visitKeyFrame(TimelineClipCore.java:226)
        at com.sun.scenario.animation.shared.TimelineClipCore.playTo(TimelineClipCore.java:167)
        at javafx.animation.Timeline.impl_playTo(Timeline.java:176)
        at javafx.animation.AnimationAccessorImpl.playTo(AnimationAccessorImpl.java:39)
        at com.sun.scenario.animation.shared.InfiniteClipEnvelope.timePulse(InfiniteClipEnvelope.java:110)
        at javafx.animation.Animation.impl_timePulse(Animation.java:1102)
        at javafx.animation.Animation$1.lambda$timePulse$25(Animation.java:186)
        at javafx.animation.Animation$1$$Lambda$118/606599863.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at javafx.animation.Animation$1.timePulse(Animation.java:185)
        at com.sun.scenario.animation.AbstractMasterTimer.timePulseImpl(AbstractMasterTimer.java:344)
        at com.sun.scenario.animation.AbstractMasterTimer$MainLoop.run(AbstractMasterTimer.java:267)
        at com.sun.javafx.tk.quantum.QuantumToolkit.pulse(QuantumToolkit.java:447)
        at com.sun.javafx.tk.quantum.QuantumToolkit.pulse(QuantumToolkit.java:431)
        at com.sun.javafx.tk.quantum.QuantumToolkit.lambda$runToolkit$363(QuantumToolkit.java:298)
        at com.sun.javafx.tk.quantum.QuantumToolkit$$Lambda$42/618865975.run(Unknown Source)
        at com.sun.glass.ui.InvokeLaterDispatcher$Future.run(InvokeLaterDispatcher.java:95)
```

Notice that the first line of the stack trace is DFS.java - that's a file I wrote. Clicking on the blue part that tells me the exception occurred on line 35 takes me to the line:

```
maze.markVisited(current);
```

The problem being flagged is that the "maze" variable is null - it was never created. To fix the bug, you should navigate to the code where "maze = new …" was supposed to happen. Either that line got left out or the method that has the line was never called.

- *Mr. E. Qualls (== vs .equals)*

If your "victim" is an incorrect variable, check its type. Remember that primitive types (int, char, boolean, etc) check for equality with ==. However object types (notably, String) use .equals to check for EQUIVALENCY (that is, two identical "twin" objects) and == to check for EQUALITY (two names that refer to one object only). Don't forget that each class is responsible for implementing the .equals method for that class. Check the implementation details if you are unsure how equivalency is defined.

- *Ms. Wonoff (one off error)*

One of the most common types of logic errors is a "one off" error. If the expected result and calculated result differ by 1, check your loops. It could be that the initialization of an accumulator variable is incorrect, or that the loop runs one time too few or one time too many.

- *Mr. Otto Bounds (out of bounds exception)*

An "out of bounds" exception occurs when trying to use an invalid index within a collection. This generally occurs with Strings and arrays. Remember that index numbering starts at 0. The last valid index is the size of the collection minus 1. To debug this type of error, trace the value of the collection and the indexing variable.

- *Ms. Ordera Operashun (order of operations)*

If an if statement is executing and you don't know why, check for an order of operations error.

Most people are familiar with the order of mathematical operations from grade school, which tells us that multiplication happens before addition, and so forth.

Therefore a line of code like this:

```
int answer = 3+2*4;
```

gives the variable answer the value 11 (not 20). While misusing the order of operations in math is a potential source of errors, a more likely source of error comes from the order of logical operators. This misunderstanding in particular can make for if statements that execute at the wrong time. For example,

```
if(3 > 4 && 1 == 0 || 1 != 0)
```

If we perform the OR first, the boolean condition will be false, but if the AND happens first, the boolean is true. In fact, the AND does happen first. The order of operations for logical operators is !, followed by &&, followed by ||.

The good news is that you don't need to memorize the order of operations. You need only to make sure there are parentheses making your meaning clear.

Detective Strategies for Defective Code - Worksheet/Recap

Define the Crime
What is the *expected* behavior and *observed* behavior of your code? Be sure to justify it - don't assume. Be specific - categorize the kind of error messages you see.

Identify the Victim
Which coding entity do you believe to be corrupt? Justify your belief.

Re-create the Crime Scene
What is the shortest sequence of steps with the smallest inputs that consistently produces the bug?

Identify the Suspects
Using the stack trace and the modifiers of the "victim", which pieces of code could possibly contain the bug?

Use Process of Elimination
Narrow down the suspect code by using breakpoints or print statements at the beginning and end of entities to follow the victim.

Follow the Trail of Evidence
Re-articulate the observed and expected behavior of the code, this time line-by-line.

Formulate a Hypothesis and Test It
Make a single change in the code, note the effect it had, and seek to understand the effect.

Rehabilitate the Offender
Make a thoughtful fix to your bug, and thoroughly test the code again.