Accept or Reject Your Hypothesis?

/run/media/lijunjie/资料/Learning Video/writing-running-fixing-code/03_testing-and-debugging/02_debugging/03_accept-or-reject-your-hypothesis_instructions.html

Once we have formed our hypothesis, we want to test it. In the simplest case, this may be trivial, as the evidence in front of us immediately convinces us beyond any doubt. If our hypothesis is "When x is 0, line 57: y / x crashes the program." and we just formed this hypothesis by printing x in our debugger when the program crashed on line 57, then we are already convinced the hypothesis is true. In such a case, we accept the hypothesis immediately and move on with no further testing.

However, do not be lulled into a false sense that you should often or eagerly accept your hypothesis without any further testing. Ten or twenty minutes spent testing a hypothesis you are "pretty sure of" is a much better investment of your time than wasting 5–10 hours making a larger mess of your code because you are acting on incorrect information.

To illustrate the benefits of such a tradeoff, suppose that your notion of "pretty sure" corresponds to your hypothesis being correct 95% of the time—meaning your hypothesis is wrong one time in 20. On the one hand, if you spend (on average) 10 minutes testing each of 20 hypotheses, that amounts to 200 minutes (3 hours, 20 minutes) of testing. On the other hand, if you blindly proceed, assuming your hypotheses are correct with no further testing—in 19 out of 20 cases, you will save yourself 10 minutes. However, in the case where you are incorrect, you might waste 5–10 hours "fixing" your code based on an incorrect hypothesis about its broken behavior! This tradeoff represents a false economy—you think you are saving time by skipping the testing of the hypothesis, but you are actually wasting more time than you have saved when you are incorrect.

You may think those time ratios sound a bit exaggerated, but, once you convince yourself of incorrect information, it is quite easy to "go down the rabbit hole." You make one seemingly logical conclusion from your false information after another. Once you change your code based on incorrect information, you are breaking your code even more—introducing new errors rather than fixing the existing one(s). You now have to debug all of these problems, which is even harder because you are convinced of a false premise. Debugging when you are convinced of a false premise is especially frustrating because you start to reach a point where nothing makes sense. This frustration makes such situations especially important to avoid.

Testing our hypothesis may proceed in a variety of ways—sometimes by a combination of them—such as:

Constructing test cases

Sometimes our hypothesis will suggest a new test case (in the sense of the testing we discussed in the Testing lesson), that we can use to produce the error under a variety of circumstances. Generally, these follow from the specific cases that your hypothesis makes

predictions about: “My program will (something) with inputs that (something).” When your hypothesis suggests this sort of result, construct a test case with the values you were thinking of, and see if your program exhibits that behavior. Also, construct other test cases that do not meet the conditions to see if you were too specific.

Inspecting the internal state of the program

We can use the same tools that we used for gathering more information (print statements, or the debugger) to inspect the internals of the program. This sort of testing is particularly useful when our hypothesis suggests something about the behavior or our program in its past—before it reaches the point where we observe the symptom. This past may be recent (“…but that means in the previous iteration of the loop…” or “then x had to be odd on the last line….”) or the distant past (“…but for that to be the case, I had to have deleted (something) from (some data structure)…” or “…then y’s value has to have changed in a way I did not expect…”). Here we want to use the debugger to inspect the state we are interested in and see if it agrees with our hypothesis. Frequently, when we take this approach, discovering the surprising change in state will give us a clue to the problem.

Adding asserts

We earlier discussed *asserts* as a testing tool, however, they are also useful for debugging. If our hypothesis suggests we are violating an invariant of our algorithm, we can often write an assert statement to check this invariant. If the *assert* does not fail, we refute the hypothesis. If the *assert* fails, it not only gives us confidence in our hypothesis, but also makes our program fail faster—the closer it fails to the actual source of the problem, the easier it is to fix.

Code inspection

Another method to test your hypothesis is to inspect and analyze your code. Sometimes a simple hypothesis can be tested with a quick inspection. For example, if you hypothesize that you forgot to initialize a particular variable, you can frequently just look at the relevant portion of the code and see whether or not you have an initialization statement or not. While this technique is generally the best for “easy” hypotheses (such as the one just described), it typically becomes much harder as you deal with more complex hypotheses.

Of course, here “difficult” is relative to the programmer’s skill level. Novice programmers who seek the help of a much more experienced programmer may get the wrong impression about the debugging process by seeing their problems debugged by inspection. While the problems that the novice programmer faces seem complex and daunting, the experienced programmer—for whom the problems are easy and familiar—may debug it by inspection in a matter of seconds. Unfortunately, many novice programmers faced with such an experience take away the *wrong* lesson: debugging is magic or lucky guesswork.

As we test, we will either convince ourselves that our hypothesis is correct, and accept it, or we will find that it is not true and reject it. In the former case, we now know what is wrong with our program and are ready to correct it. Typically, identifying the precise problem and cause are 90% of the battle—thus if our hypothesis was a good one, we are most of the way there. Of course, sometimes our problem is severe and requires significant modifications to our program. In the worst cases, a significant redesign and implementation of large portions of the code from scratch.

The decision to throw away large portions of code and redo them from scratch is not one to be taken lightly, nor an easy one to make. In making such a decision, the programmer should be wary of *The Poker Player's Fallacy* — the temptation to make a decision based on prior investments rather than future outcomes. This term comes from a fallacy that many novice poker players succumb to: betting based on how much they have already put into the pot, rather than how likely they are to win the hand ("I'm already in for 200,soImayaswellbetanother200,soImayaswellbetanother10 on the off chance I win."). If you are not likely to win the pot, betting another $10 is just throwing that money away. The smart poker player will only bet on her current hand if she thinks she can win (whether by a better hand, or a bluff).

Similarly, when evaluating whether to modify the current code or throw it away and start fresh, you should not consider how much time you have already put into it, but rather how much time it will take to fix the current code versus redesigning it from scratch. Note that this is not intended to suggest you should throw away your code and redesign it from scratch every time there is an error. Instead, you should be contemplating the time required for both options, and making a rational tradeoff.

If instead of accepting your hypothesis, you find that you must reject it, do not despair. In investigating this hypothesis, you have gained valuable information which will inform your next hypothesis. You may find a new hypothesis immediately after you reject your current one ("Aha! the problem is not if *z* is even, but rather if it is a multiple of 8!"). If not, return to the information gathering stage and repeat the hypothesis formation process.

One thing to be wary of when rejecting a hypothesis: there may be multiple errors in your code. Do not be mislead by symptoms of other errors masking your current problem. For example, suppose that the program you are testing and debugging has two errors in it. One of these errors causes the program to crash on line 99 if x is a multiple of 17. Another error causes the program to crash before it reaches lines 99 if x is greater than 10,000. You have formed a hypothesis that accurately describes the first error, and are testing it with x=17,000 (which is a multiple of 17, and greater than 10,000).

The fact that the program crashes *sooner* than we expect should not cause us to reject the hypothesis immediately. We must instead consider the possibility that there is another error, which is triggered by overlapping conditions. When faced with such a possibility, we have two options.

One option we might take is to defer our investigation of the first error, while we try to debug the second. If we can fix this second error, we can retest the case, and find that it does not contradict our original hypothesis on the first error.

The other option we have is to confirm our suspicion that the difference in behavior (between what we observed and what we hypothesized) is in fact a symptom of a different problem, then proceed with testing the current hypothesis. Here, we must proceed with caution—we do not want to reject a valid hypothesis, but at the same time we must be careful not to accept a false one. We should confirm that the test case in question is actually not triggering the situation we intended to test—perhaps it is not reaching that point in the code, or not exhibiting the intended circumstances when it does reach that point. Once we have confirmed that the test case is not actually testing the hypothesis, we can continue with other cases. Of course, after we fix the current error, we should return to this case and find out what the other error is.