Chapter 6: Testing and Debugging

Testing – the process of running a program to try and ascertain whether or not it works as intended.

Debugging – the process of trying to fix a program that you already know does not work as intended.

pg. 70 “Good programmers design their programs in ways that make them easier to test and debug. The key to doing this is breaking the program up into components that can be implemented, tested, and debugged independently of each other.”

pg. 71 “The most important thing to say about **testing** is that its purpose is to show that bugs exist, not to show that a program is bug-free.”

pg. 71 “The key to **testing** is finding a collection of inputs, called a **test suite**, that has a high likelihood of revealing bugs, yet does not take too long to run. The key to doing this is partitioning the space of all possible inputs into subsets that provide equivalent information about the correctness of the program, and then constructing a test suite that contains one input from each partition.”

A **glass-box test suite** is **path-complete** if it exercises every potential path through the program. This is typically impossible to achieve, because it depends upon the number of times each loop is executed and the depth of each recursion.

A **path-complete test suite** does not guarantee that all bugs wil be exposed.

6.1.3 Conducting Tests

Testing is often thought of as occurring in two phases. One should always start with **unit testing.** During this phase testers construct and run tests designed to ascertain whether individual modules (e.g. functions) work properly. This is followed by **integration testing**, which is designed to ascertain whether the program as a whole behaves as intended. In practice, testers cycle through these two phases, since failures during integration testing lead to making changes to individual modules.

Whenever any chance is made, no matter how small, you should check that the program still passes all of the tests that it used to pass.

6.2 Debugging

If a program has a bug, it is because you put it there.

Overt-bug- has an obvious manifestation, e.g. the program crashes or takes far longer (maybe forever) to run than it should.

Covert-bug has no obvious manifestation. The program may run to conclusion with no problem—other than providing an incorrect answer.

pg. 77 “Many bugs fall between the two extremes, and whether or not the bug is overt can depend upon how carefully one examines the behavior of the program.”

persistent bug- occurs every time the program is run

intermittent bug occurs only some of the time, even when the program is run on the same inputs and seemingly under the same conditions.

“the best kinds of bugs to have are overt and persistent.” - pg. 78

Good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent. This is often called **defensive programming.**

pg. 78 “Bugs that are both **covert** and **intermittent** are almost always the hardest to find and fix.”

6.2.1 Learning to Debug

Debugging starts when testing has demonstrated that the program behaves in undesirable ways. **Debugging** is the process of searching for an explanation of that behavior. **The key to being consistently good at debugging is being systematic in conducting that search**

Start by studying the available data. This includes the test results and the program text. Remember to study all of the test results. Examine not only the tests that revealed the presence of a problem but also those tests that seemed to work perfectly.

Next, form a hypothesis that you believe to be consistent with all the data.

Next, design and run a repeatable experiment with the potential to refute the hypothesis... decide before running the experiment how you would interpret various possible results.

Finally, keep a record of what experiments you have run.

6.2.2 Designing the Experiment

”Think of debugging as a search process, and each experiment as an attempt to reduce the size of the search space. It's often useful to think of the size of the search space as being the product of the amount of the code to be examined and amount of testing needed to provoke the bug.” - pg. 79

pg. 80

Binary Search- Find some point about halfway through the code, and devise an experiment that will allow you to decide if there is a problem before that point that might be related to the symptom. (of course, there may be problems after that point as well, but it is usually best to hunt own one problem down at a time).

Page 80-81 provides an example of binary searching

6.2.3 When the Going Gets Tough

Stop asking yourself why the program isn't doing what you want it to. Instead, ask yourself why it is doing what it is.

Keep in mind that the bug is probably not where you think it is. If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking here the bug

cannot be.

Try to explain the problem to somebody else... a good thing to try to explain is why the bug cannot be in certain places.

Don't believe everything you read. In particular, don't believe the documentation.

Stop debugging and start writing documentation. This will help you approach the problem from a different perspective. \

6.2.4 And When You Have Found “The” Bug

Remember that the goal is not to fix one bug, but to move rapidly and efficiently towards a bug-free program.

Always make sure that you can get back to where you are. Use [your disk space] to store old versions of your program.

pg. 83 “Finally, if there are many unexplained errors, you might consider whether finding and fixing bugs one at a time is even the right approach. Maybe you would be better off thinking about whether there is some better way to organize your program or some simpler algorithm that will be easier to implement correctly.”