Chapter 6 Notes

Tests

Testing is the process of running a program to try and ascertain whether it works as intended. Debugging is the process of trying to fix a program that you already know does not work as intended. The key to doing this is breaking the program up into separate components that can be implemented, tested, and debugged independently of other components.

The first step in getting a program to work is getting the language system to agree to run it – that is, eliminating syntax errors and static semantic errors that can be detected without running the program.

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. This can be achieved using partition. **A partition** of a set divides that set into a collection of subsets such that each element of the original set belongs to exactly one of the subsets. We can use one example from each subset as a test.

Heuristics based on exploring paths through the code fall into a class called **glass-box testing.** Heuristics based on exploring paths through the specification fall into a class called **black-box testing.**

Black Box Testing

In principle, black-box tests are constructed without looking at the code to be tested. A good way to generate black-box test data is to explore paths through a specification.

We should always check boundary conditions for each type of data. When perform black-box testing, always think about aliasing.

Glass Box Testing

A glass-box test is a test based on the code. A glass-box test suite is **path-complete** if it exercises every potential path through the program. However, even a path-complete test suite does not guarantee that all bugs will be exposed. There are a few rules of thumb that are worth following:

* Exercise both branches of all “if” statements.
* Make sure that each except clause is executed.
* For each for loop, have test cases in which:
  + The loop is not entered
  + The loop is executed exactly once
  + The loop is executed more than once
* For each while loop:
  + Look at the same kinds of cases as when dealing with for loops.
  + Include test cases corresponding to all possible ways of exiting the loop
* For recursive functions, include test cases that cause the function to return with no recursive call, exactly one recursive call, and more than one recursive call.

Tests

Testing is often through 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 units of code work properly. This is followed by **integration testing**, which is designed to make sure whether the program behaves as intended.

During unit testing, we often need to build **stubs** as well as drivers. Drivers simulate parts of the program that use the unit being tested, whereas stubs simulate parts of the program used by the unit being tested. Ideally, a stub should:

* Check the reasonableness of the environment and arguments supplied by the caller.
* Modify arguments and global variables in a manner consistent with the specification, and
* Return values consistent with the specification.

Debugging

Runtime bugs can be categorized along two dimensions:

* Overt/Covert: An **overt bug** has an obvious manifestation, e.g., the program crashes or takes far longer (maybe forever) to run than it should. A **covert bug** has no obvious manifestation. The program may run to conclusion with no problem – other than providing an incorrect answer
* Persistent/Intermittent (not continuous): A **persistent bug** occurs every time the program is run with the same inputs. An **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. 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 called **defensive programming.**

Debugging is the process of searching for an explanation of a behavior. Here are the steps in the debugging process:

* Start by studying the available data. This include the review of test results and program text. Try to understand why one test work and another did not is the way.
* Next, form a hypothesis that you believe to be consistent with all the data.
* Design and run a repeatable experiment with the potential to refute the hypothesis. You can achieve this by putting print() statements before unassertive statements.
* Keep a record of what experiments you have tried.

Designing Debugging Experiment

Think of debugging as a search process, and each experiment as an attempt to reduce the size of the search space. One way to reduce the size of the search space is to design an experiment that can be used to decide whether a specific region of code is responsible for a problem uncovered during integration testing.

Look for the usual suspects:

* Passed arguments to a function in the wrong order
* Misspelled a name
* Failed to reinitialize a variable
* Tested that two floating point values are equal (==) instead of nearly equal
* Tested for value equality (forgot to use id())
* Forgotten that some built-in function has a side effect
* Forgotten the () that turns a reference to an object of type function into a function invocation
* Created unintentional alias

When you have found the bug, try to ask yourself if this bug explains all the observed symptoms, or whether it is just the tip of the iceberg. Before making any change, try and understand the ramification of the proposed “fix”. Will it break something else? Does it introduce excessive complexity? Does it offer the opportunity to tidy up other parts of the code? In conclusion, always make sure that you can get back to where you are.