Introduction to Testing

CIS*2750

Advanced Programming Concepts

Based on "The Practice of Programming" by

Brian W. Kernighan and Rob Pike, 1999.

Series on Testing

1. Defensive programming
    - *Detect problems as early as possible*
2. Intro. to testing
    - *Classic testing methods for “finished” code*
3. Debugging
    - *Fixing defects uncovered by testing*

Introduction to Testing

..testing can demonstrate the presence of bugs, but not their absence....Edsger Dijkstra

- Debugging and testing are not the same thing!
- Testing is a systematic attempt to *break* a program.
  - Correct, bug-free programs by construction are the goal but until that is possible (if ever!) we have testing.

Introduction to Testing

- It is not unusual for developers to spend 40% of the total project time on testing.
  - For life-critical software (e.g. flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined.
- Since testing is basically **destructive** in nature, it requires that the tester discard *preconceived* notions of the *correctness* of the software to be tested.

Software Testing Fundamentals

- Testing objectives include
  - Testing is a process of executing a program with the intent of *finding an error*.
  - A **good** test case is one that has a high probability of finding an as yet undiscovered error.
  - A **successful** test is one that uncovers an as yet undiscovered error.

Software Testing Fundamentals

- Testing should systematically uncover *different classes* of errors in a minimum amount of time and with a minimum amount of effort.
  - A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specification.
  - The data collected through testing can also provide an indication of the software's reliability and quality.
- But, testing cannot show the absence of defects -- it can only show that software defects are present.

The Nature of Software Defects

- Typographical errors are *random*.
- Logic errors and incorrect assumptions are *inversely proportional* to the probability that a program path will be executed.
  - General processing tends to be well understood while special case processing tends to be prone to errors.
- But, we often *believe* that a logical path is not likely to be executed when in fact, it may be executed on a regular basis.
  - Our unconscious assumptions about control flow and data lead to design errors that can only be detected by testing.

Classic Forms of Testing

- “Black box” testing
  - Consider only public interface of module or entire program
  - Apply inputs → examine outputs → compare to spec
  - Ideally done by independent testing group (*not* original programmer)

- “White box” testing
  - Involves looking at module’s source code, then targeting tests to exercise particular statements

White Box Testing

- White box testing is a test case design method that uses the **control structure** of the procedural design to derive test cases that:
  - guarantee that all *independent paths* within a module have been exercised at least once,
  - exercise all *logical decisions* on their true and false sides,
  - execute all *loops* at their boundaries and within their operational bounds, and
  - exercise internal *data structures* to ensure their validity.

3 Steps to Testing Nirvana

1) Think about potential problems as you design and implement. Make a note of them and develop tests that will exercise these areas.
   - Document all **loops** and their **boundary conditions**, all **arrays** and their **boundary conditions**, all **variables** and their **range** of permissible values.
   - Pay special attention to **parameters** from the command line and into functions and what are their valid and invalid **values**.
   - Enumerate the possible combinations and situations for a piece of code and design tests for all of them.
   - GIGO - what happens when garbage goes in?

3 Steps to Testing Nirvana

2) Test systematically, starting with easy tests and working up to more elaborate ones.
   - Often leads to “bottom up” testing, starting with simplest modules at the lowest level of calling
   - When those are working, test their callers
   - Document (and/or automate) this testing so that it can be repeated (**regression testing**) constantly as the code grows and changes.

3 Steps to Testing Nirvana

3) Within a module, test *incrementally* as you code
   - Write, test, add more code, test again, repeat
   - The earlier that errors are detected, the easier they are to locate and fix.

Tricks of the Trade

- Testing boundary conditions:
    - starting/ending values of loop counters, end condition of for/while/do…until
  - Check **loops** and **conditional statements** to ensure loops are executed the correct number of times and that branching is correct
  - If code is going to fail, it usually fails at a boundary!
  - Check for off-by-one errors, empty input, empty output (extreme cases)

When will this code fail?

```
// read stdin into s buffer till newline or full
int i;
char s[MAX];

for (i=0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i);
    //loop has no body; work done in condition

s[--i] = '\0';   // terminate the string
```

Strategies

- Know what output to expect.
  - It is obvious that you cannot know if your program is correct unless you know what output it *should* produce in all situations.
  - As programs become bigger and more complicated, this becomes harder and harder to do.
  - Use programs like cmp (compare files for identity) and diff (report differences) to compare against known results (aka “gold output”)

More Strategies

- Measure test **coverage**.
  - All lines in the code should be executed by one of your tests.
  - This is a very difficult thing to achieve or, indeed, to even know if you have achieved it!
    - Some profilers have a feature which provides a statement frequency count for all lines in the code.
    - Some IDEs visually indicate test coverage of statements
  - The bottom line is, “if you haven't tested it, you don't know if it has a bug in it!”

What is
Test-driven Development (TDD)?

- A rigourous SW development process that ensures the SW always "meets the spec" at any point in time → "simpler designs" "confidence"

  loop{ Write *automated* test cases to cover the next bits of functionality from spec;
  Write "just enough" code to pass the tests; **}**

- Drawbacks: How do you know the tests are *correct* (blind spots) and you have *enough*?
  - Frontloads your development with test-writing

Scaffolding

- Software scaffolding is built for the purpose of making it easy to **exercise** code.
- It consists of **temporary** programs and data that give programmers **access** to system **components**.
  - It is indispensable during testing and debugging but is usually never delivered to the customer.
  - If an error is detected in the code, then its scaffolding can be **reused** (so save it).

An Isolation Technique

- Scaffolding also allows some module(s)-under-test to be tested *without the risk of being affected by interactions with other modules*.
  - Really useful when test case setup would take much time and multiple manual steps because the code being tested is embedded in other code.
  - Scaffolding allows you to exercise code **directly**.
- Scaffolding is often **big**—estimates range from *half* as much code in scaffolding as there is in the product to *as much* scaffolding as delivered code!

3 Types of Scaffolding: The Stub

- A low-level module of “dummy” code that can be called by a higher-level module that is being tested. A stub can:
  - take no action and return control immediately,
  - test the data fed to it,
  - print a diagnostic message, perhaps just echoing the input parameters
  - return a standard answer regardless of the input,
  - provide timing—burn up the number of clock cycles allocated to the real module, or
  - function as a quick and dirty version of the real module.

Types of Scaffolding: The Driver

- A fake module that calls the real module being tested, also called a **test harness**. It can:
  - call the module to be tested with a fixed set of inputs,
  - prompt for input interactively, or take arguments from the command line, or read arguments from a file and call the module, or
  - run through predefined sets of input data in multiple calls to the module.

Types of Scaffolding: the Dummy File

- A small version of a real data file that has the same types of components as that file. Advantages include:
  - its small size allows you to know its exact contents, and
  - more chance of it being error-free.
  - Since it is designed for testing, the contents can be designed to make any error conspicuous.

Scaffolding Construction

- Write one or more modules (to be tested) and separately compile them.
  - Even if the modules are not meant to stand by themselves.
- Write a main() scaffolding module (=driver) and include calls to the other modules.
  - The main() module reads arguments from the command line (interactive program, file) and passes them to the modules being tested.
  - This exercises the modules on their own before integrating them with the rest of the system.

Scaffolding Construction

- After the tested module is integrated, *save* the scaffolding code
  - This code can even be left in the actual source file and preprocessor #ifdefs or comments can be used to deactivate it during the actual operation of the system.
  - If you need to use it again, time is saved in recovering the test framework since you know where it is.

On to Debugging!

- Testing **exposes defects** in the software

- Still have to fix them = debugging
  - Doesn't have to be hit-or-miss, can have strategies
  - *Good news:* Your debugging speed goes up as your experience grows → builds *intuition* that homes in on bugs more quickly