Testing Tutorial

CS 246

May 25, 2018

1 Teaser Problem

Draw the following on the board with rounded boxes around the symbols[1]:

> 3 6 B C
>
> "If a card has an even number on one side, then its opposite side has a B."

"Which cards do you need to flip to test this statement?"

The correct answer is 6 and C, 6 to check for a B, C to check there isn't an even number. In most groups, only 1 in 10 people get both parts of the correct answer.

2 Introduction

As programmers, how do we know that our programs are correct?

- When we ship code to our clients, how can we be confident it does what they asked us for?
- When you submit an assignment to the TAs, how can you know that you'll get a good mark?

The answer to all these questions is **testing**.

The goal of this tutorial is to give you some basic testing terminology and techniques you can apply yourself.

3 Terminology

Fault static defect in the software (e.g. root illness)

[1] An instance of the *Wason selection task* or *four-card problem*

Error incorrect internal state that is the manifestation of the fault (e.g. high blood pressure)

Failure external incorrect behaviour with respect to the requirements or other description of the expected behaviour (e.g. symptoms)

We'll often use **bug** in a loose sense to talk about any of these, but with more precision, existence of an observable *failure* implies an internal *error* due to some root *fault*.

Testing and **debugging** are related, but often confused. *Testing* is about detecting failures in your program, while *debugging* is about locating and fixing the root faults. You have a second tutorial on debugging, so this tutorial will focus on testing.

4 Payroll example

```
// Calculates the pay for a worker.
// First 40 hours paid at ‘hourly_rate’
// Next 5 hours paid at 1.5 * ‘hourly_rate’
// Any more hours paid at 2 * ‘hourly_rate’
int payroll(int hourly_rate, int hours_worked);
```

"Which values should we test this function with?"

One approach is called **equivalence partitioning** — we pick one representative from each equivalence class to test.

- take one example from each of the three classes.
- note that you only need a few representative examples, not the entire set of values.

Another approach is **boundary value** testing — it has been said that the two hardest problems in programming are naming things, memory management, and off-by-one errors. As the joke suggests, a good place to look for bugs is at the boundary values of classes.

- Test below, on, and above boundaries
- In this case, for hours worked: 39, 40, 41, 44, 45, 46
- It might not be feasible to test all possible combinations of boundary cases.

A third approach is **error guessing** — as you get more experience with a particular problem domain, you'll develop a sense for what sort of inputs are likely to cause failures, and learn to test them too.

Edge cases the programmer is likely to forget:

- What if `hours_worked` is greater than 168 (the number of hours in a week)?
- What if `hours_worked` is zero?
- What about `hourly_rate`?
- What if either is negative?
- How does the floating-point multiplication `1.5 * hourly_rate` work if `hourly_rate` is an odd integer?

Some of these questions are really bugs in the specification – maybe the `int` parameters should be `unsigned int`, or the documentation should describe the behaviour or state it to be undefined.

- If you see issues like this in an assignment specification, testing what the reference solution does is a good idea if you have one.
- If you don't have a reference solution (the case in a lot, but not all of the professional programming you'll do), ask whoever wrote the spec (for this course, the course staff on Piazza).

It takes a bit more effort, but for some problems it is worthwhile to randomly generate a large set of plausible inputs to a program to try and break it. This is called **fuzz testing**.

- Requires that you have a second way to test if the program behaved correctly
- Often used to find failures where a program crashes; in this case "behaved correctly" means "didn't crash" rather than "correct answer"

In practice, you can use any or all of these approaches together in your own testing.

5 White-box testing

The tests we discussed on the previous example were all **black-box** tests, so called because we can't see the code implementing it, and are simply trying to guess what its behaviour should be from the specification.

An alternative when you do have the code is **white-box** testing

- Attempts to test every branch of every single function in the code
- May even try to test every single path through the code
- It may not be possible to actually test all possible execution paths, though tools exist to measure **test coverage** (how much of your code is being run by the tests).

6 Assertions

It's not quite white-box testing, but as you've seen in CS 136, if you can modify source code, you can insert **assertions** to make the code verify assumptions you're making.

Pre-condition Test properties of parameters at top of function

Invariant Test some property in an intermediate stage of the computation

Post-condition Test some property before returning from a function

Non-null Checking that pointers are not null is a common use of assertions.

Unreachable Use `assert(false);` to state that a particular code path will never be executed.

```
int payroll(int hourly_rate, int hours_worked) {
assert( hourly_rate > 0 && "hourly rate should be positive");
assert( hours_worked >= 0 && "cannot work negative hours");
// ...
}
```

Because of some quirks of the C++ type-system, a string literal is treated as a (non-null) pointer, which is a *true* value. We can rely on this "one weird trick" to include a message in our assertions, which will be printed (along with the source file and line number) if the assertion triggers. For "unreachable" assertions, `assert(!"unreachable");` works similarly.

To use assertions in C++, you need to `#include<cassert>`. Don't put any computations that other code depends on in assertions, because they can be turned off for more performance by passing the `-DNDEBUG` flag to the compiler.

7 Unit Testing & Integration Testing

Unit Testing Test individual components of a program (like the interface of a single function).

Integration Testing Test the interactions between larger parts of the program (up to and including the whole thing).

You can write your tests bottom-up by starting with unit tests first, then writing integration tests, or top-down by writing integration tests and then unit tests for particularly tricky components, but it's a good idea to have both sorts of tests.

8 Automated testing

Testing can be tedious — it is a mechanical process you often need to run many times. Luckily we have computers to do tedious mechanical repetition for us.

A program we write to do our testing for us is called a **test harness**.

- We asked you to write a simple test harness in Assignment 1.
- We use a similar but more sophisticated test harness to test your assignments.

(Walk through pre-written test harness for payroll example here, just a simple line-based I/O program to call the function.)

9 Conclusion

To review:

- *Testing* is a way to find *failures* in a program or piece of code, places where it doesn't behave as it should.
- Once you've found a failure, you *debug* to find and fix the underlying *fault*.
- Some good *black-box* tests to use are representatives of equivalent classes of input, boundary values (especially zero), and weird cases that might have been missed by the spec or original programmer. You might even randomly generate test cases for *fuzz testing*.
- If you have access to the code, you can do *white-box* testing on different code paths, or insert assertions to do testing at specific points in execution.
- *Automation* is a good way to make testing easier.

As a final tip, research on how people do testing[2] shows that people tend to test cases that "should work" much more than tests which "should fail". However, the "should fail" cases can often be more useful at detecting faults – the usual case people miss in the example from the start of class is the one where it "should fail", so remember to test those cases too.

[2] Leventhal et al. "Positive test bias in software testing among professionals: A review", `http://www.springerlink.com/content/y6p0062610u13027/`