Remark: TODO: Change \iftrue in commands.tex \def\rem to \iffalse in final copy

COMP2123 self-learning report
Unit testing

December 17, 2018

Abstract

This report discusses the motivation, available techniques and difficulties of unit testing.

Contents

1 What is unit testing?

Instead of just running the program and seeing if the output is correct, unit testing splits down a software into small components and checks if the desired behaviour of each component is correct.

2 Motivation

2.1 Discover bugs early

As the scale of a software project grows, debugging becomes more complicated. It may take a long time to discover edge case bugs in an old component, which is very difficult to debug after a long time. Unit testing allows identification of bugs as soon as possible with little impact.

2.2 As a method of specification

Unit testing can also be used as a means of project requirement specification.

3 Unit testing methods

3.1 Testing for expected result

The intuitive way is to write a test that tests the output of each function.

```
class SimpleSpec {
public:
    void testFooBar() {
        ASSERT_EQUAL(fooBar(), "qux")
    }
}
```

The `ASSERT_EQUAL` macro function would compare the result of `fooBar()` with `"qux"` and trigger an error if they are not equal. This macro function can be implemented very easily:

```
#define ASSERT_EQUAL(actual, expect) if(actual != expect) \
    throw sprintf("Test for %s failed: expected, %s, got %s", \
    #actual, expect, actual);
```

Some other common assertions include:

- Null checks
- Arithmetic comparisons $>$ $>=$ $<$ $<=$
- That an exception must be gracefully thrown

By running a series of similar tests every time before moving to another project subcomponent, bugs can be identified before it spreads to other components. This is particularly helpful when certain bugfixes might result in prototype changes, resulting in incompatibility with other components during bugfixes.

3.2 Generating test parameters

If a function accepts a parameter, it is not possible to execute a test on every possible parameter value. Instead, the parameters can be generated randomly in every test. By supplying a test sample large enough, most bugs can be discovered quickly.

Remark: Generate parameters randomly, possibly by reverse calculation

3.2.1 Testing for edge cases

Remark: E.g. test for empty strings, Float.INFINITY, 0, etc.

4 Unit testing techniques

4.1 Test coverage

Test coverage is a criterion to assess the representativeness of the unit tests of a project by counting the number of lines executed in the test.

Remark: Talk about the integration of debuggers and how to interpret coverage (codecov.io)

4.2 Test-driven development (TDD)

As a consequence of unit testing, development flow becomes more fluent if each development subtask is based on certain test cases.

4.3 Behaviour-driven development (BDD)

Remark: Refer to cucumber's framework

Remark: TDD and BDD are a bit different, but if we can't explain the difference, we can merge them together

4.4 Dependency mocking

Remark: https://enterprisecraftsmanship.com/2016/06/09/styles-of-unit-testing/ provides some insight on why mocking is bad

5 Difficulties

5.1 Coupling

Coupling is the cyclic dependency between units to be tested.

Remark: Explain what is coupling, why it is bad, how hard it is to prevent, and common practices e.g. abstraction, interfaces, mocking to prevent coupling

5.2 Test case selection

As the number of variables to a feature increases, the number of test cases might increase exponentially.