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

Remark: In case we don't have enough content, just add more code examples, and maybe compare specific unit testing frameworks in some frameworks.

Remark: Talking about CI (continuous integration) platforms like Travis-CI is also possible

COMP2123 self-learning report
Unit testing

December 21, 2018

Chan Kwan Yin, Lee Chun Yin

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 a software grows in scale, there would be some "stale" components of the software unmodified for a long time. New components are built upon these older components, but if the older components are not stable enough, their bugs would propagate to the new components, creating confusion for new developers.

In addition, if a bug is only discovered a long time after writing the corresponding code, the developer is very likely to forget about what the code was intended to do.

Unit testing allows identification of bugs as soon as possible with little impact, overall reducing the development cost.

2.2 As a method of specification

Unit tests can also be used as a means of project requirement specification. It is common for programmers and users to misunderstand each other's requirements and waste a lot of time. Unit tests can provide a technical and strict definition of behaviour.

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.

If we have a `fooBar.cpp` with the following definition:

```
#include <string>

std::string fooBar() {
    return "qux";
}
```

To ensure `fooBar()` always return `"qux"`, we can write a test to test this behaviour:

```
#include "fooBar.cpp"
#include "assert.h"

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:

```cpp
#include <iostream>

#define ASSERT_EQUAL(actual, expect) \
    auto actualValue = actual; \
    auto expectValue = expect; \
    bool eq = actualValue == expectValue; \
    if (!eq) { \
        throw std::string("Expected ") + #actual + " to return " +
    expectValue + ", got " + actualValue; \
    }
```

In this implementation, if the values are not equal, an exception string is thrown.

Different unit testing frameworks may have different error behaviour, and some are able to integrate with IDEs for advanced analysis.

Some other common assertions include:

- Null checks
- Arithmetic comparisons $< \leq > \geq$
- That an expected exception must be 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 Increasing the test size

If a function accepts parameters, multiple calls with different values should be passed to the function.

Suppose we want to test a function that converts a number to scientific notation rounded to 3 significant figures:

```cpp
#include <cmath>
#include "SciNotation.h"

SciNotation sciNot(double number) {
    int exp = (int) floor(log10(number)) - 2;
    int digits = (int) round(number * pow(10.0, -exp));
    return SciNotation{digits, exp};
}
```

3.3 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 by the unit test.

Suppose we want to test a function that rounds a number to 3 significant figures:

> Remark: Generate parameters randomly, possibly by reverse calculation

3.3.1 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.

> Remark: TODO: Add flowchart

4.3 Behaviour-driven development (BDD)

While TDD provides a reliable method to test individual function executions, it cannot precisely and concisely define functions that depend on global/object states. BDD defines the behaviour of a function using some specific terms:

Scenario

Given

When

Then

Remark: TODO: fill in the blanks

This framework also subclassifies a task into multiple components to identify the exact source of error.

Remark: TODO: Insert example code here

Remark: Refer to cucumber's framework

Remark: To read: https://enterprisecraftsmanship.com/2016/06/09/styles-of-unit-testing/

4.4 Dependency mocking

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

Remark: Consider moving this to the part about coupling

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

Remark: Consider cyclic dependency checks in golang, which is an excellent example of why it is hard

5.2 Test case selection

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