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

Chan Kwan Yin, Lee Chun Yin

December 22, 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 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:

```
#include <iostream>

#define ASSERT_EQUAL(actual, expect) {\
    auto actualValue = actual; \
    auto expectValue = expect; \
    bool eq = actualValue == expectValue; \
    if (eq) { \
        std::cout << std::string("Successful: ") + #actual + " = " + actualValue << std::endl; \
    } else { \
        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:

```
#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};
}
```

Testing the simple case of rounding 1235 to 124×10^1 is successful:

```
#include "sciNot1.cpp"
#include "assert.h"

void testSciNot() {
    ASSERT_EQUAL(sciNot(1235), SciNotation(124, 1))
}
```

```
Executing test testSciNot
Successful: sciNot(1235) = 124e1
```

However, adding more test parameters, such as negative inputs, would turn out that the function does not always function as expected:

```
#include "sciNot1.cpp"
#include "assert.h"

void testSciNot() {
    ASSERT_EQUAL(sciNot(1235), SciNotation(124, 1))
    ASSERT_EQUAL(sciNot(-1234), SciNotation(-123, 1))
}
```

```
Executing test testSciNot
Successful: sciNot(1235) = 124e1
Failed test for testSciNot: Expected sciNot(-1234) to return -123e1, got 0e2147483646
```

From this, we can identify that a negative input results in a negative value, apparently because of the `log10` function call. So a simple workaround of fixing the input to a positive number can be made:

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

SciNotation sciNot(double number) {
    int sign = number >= 0 ? 1 : -1;
    number *= sign;
```

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

```
Executing test testSciNot
Successful: sciNot(1235) = 124e1
Successful: sciNot(-1234) = -123e1
```

3.3 Generating test cases

Since a larger test size is more likely to catch bugs, it is a good idea to generate a large sample of test cases. This can be achieved through the `rand()` function:

```
#include <cstdlib>

#define SAMPLE_INTS(iterations, min, max, as, i) \
    auto i = iterations; \
    for(int as = rand() % (max - min) + min; i > 0; i--, as = rand() % (max - min) + min)
```

Then tests can be generated using these functions.

3.3.1 Generating expected results: $f(x)$ vs $f^{-1}(y)$

Remark: Contradiction: write good tests vs write good code: is the function itself or the test more likely to have bugs?

3.3.2 Filtering test cases

```
#include <cmath>
#include "sciNot4.cpp"
#include "assert.h"
#include "sample.h"

void testSciNot() {
    SAMPLE_INTS(100, -999, 999, digits, i) {
        SAMPLE_INTS(2, -14, 14, exp, j) { // limit to +-14 because of float precision
            bool roundUp = rand() & 1;
            int sign = digits > 0 ? 1 : -1;
            double input = digits + 0.1 * (rand() % 5 + (roundUp ? 5 : 0)) * sign;
            input *= pow(10, exp);
            std::cout << "Testing input = " << input << std::endl;
            ASSERT_EQUAL(sciNot(input), SciNotation(digits + roundUp * sign, exp))
        }
    }
}
```

```
Executing test testSciNot
Testing input = 4.785e+06
Successful: sciNot(input) = 479e4
Testing input = 4.786e-11
Successful: sciNot(input) = 479e-13
Testing input = 762200
Successful: sciNot(input) = 762e3
Testing input = 7.628e-06
Successful: sciNot(input) = 763e-8
Testing input = 8.71e+09
Failed test for testSciNot: Expected sciNot(input) to return 87e8, got 871e7
```

Remark: Need to filter test parameters because $087 = 87 < 100$

3.3.3 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.

6 Continuous integration

Remark: Only add this part if we don't have enough materials to add