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 23, 2018

Abstract

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

Contents

1 What is unit testing? 1

2 Motivation 1
2.1 Discover bugs early . . . 1
2.2 As a method of specification . . . 1

3 Unit testing methods 1
3.1 Testing for expected result . . . 1
3.2 Increasing the test size . . . 2
3.3 Generating test cases randomly . . . 3
3.3.1 Random output, inverse-evaluated input . . . 3
3.3.2 Property-based testing . . . 3
3.3.3 Reimplement algorithm by brute-force . . . 4
3.3.4 Testing for edge cases . . . 5

4 Unit testing techniques 5
4.1 Test coverage . . . 5
4.2 Test-driven development (TDD) . . . 5
4.3 Behaviour-driven development (BDD) . . . 6
4.4 Dependency mocking . . . 7

5 Difficulties 7
5.1 Coupling . . . 7
5.2 Test case selection . . . 7

6 Continuous integration 7

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 randomly

Since a larger test size is more likely to catch bugs, it is a good idea to generate a large sample of test cases.

Generating random test cases also avoids the case where a developer is writing specific test cases (e.g. only multiples of 7) to avoid unit tests from reflecting a known bug in the code.

3.3.1 Random output, inverse-evaluated input

But a contradictory condition arises: If the test cases are generated, how to test if the result is correct? It is not possible to calculate the value in the test generator, because that would involve reimplementing the tested function.

Instead, if an inverse of the tested function can be written (or to generate any of the possible inverse values if the function is not an injection), the random generator can be used to generate random results instead.

Nevertheless, the inverse function is often harder to write and likely to have bugs than the function to test for, and is usually unintuitive to be used as a way of software specification. A better approach is to use property-based testing.

3.3.2 Property-based testing

Instead of comparing if the output is equal to an expected result, property-based testing compares the characteristics of the output to determine if it is reasonable.

In C++, the rapidcheck library provides a property-based testing framework that generates a random sequence of data by type for assertion.

For example, in the 3-sigifnicant-figure example above, the requirements are:

1. Given an input n, the output $\hat{n} = d \times 10^x$ is returned.
2. The result significand d must have exactly 3 significant figures, i.e. $d \in [100, 999]$
3. The error must be correct to the 3^{rd} significant figure: $\hat{n} - n \in [-0.5 \times 10^x, 0.5 \times 10^x)$

The 1^{st} requirement is defined through the function prototype, and the 2^{nd} and 3^{rd} requirements can be specified by the following test:

```
#include <cmath>
#include <vector>
#include <rapidcheck.h>
#include "sciNot4.cpp"

using namespace std;
using namespace rc;

void testSciNot() {
    check("requirement 2: 3 significant figures", [](const vector<double> &vec) {
        for(double input : vec) {
            SciNotation sci = sciNot(input);
            int digits = sci.getDigits();
            RC_ASSERT(100 <= digits && digits < 999);
        }
    });
    check("requirement 3: error within bounds", [](const vector<double> &vec) {
```

```
        for(double input : vec) {
            SciNotation sci = sciNot(input);
            double error = sci.toDouble() - input;
            error /= pow(10, sci.getExp());
            RC_ASSERT(-0.5 <= error && error < 0.5);
        }
    });
}
```

```
Executing test testSciNot
Using configuration: seed=14108553352369218528

- requirement 2: 3 significant figures
Falsifiable after 7 tests and 2 shrinks

std::tuple<std::vector<double>>:
([0])

sciNot5.test.cpp:14:
RC_ASSERT(100 <= digits && digits < 999)

Expands to:
100 <= 0 && true

- requirement 3: error within bounds
OK, passed 100 tests
Some of your RapidCheck properties had failures. To reproduce these, run with:
RC_PARAMS="reproduce=BQiclFXdpJXZtVmb0BiM6AyMgMXan5Waml2Yh5GdgYWanVnclNH4j0pDGa6yDD+
    Id6ghmu8wgPSnOYopLPM4j0pDGa6yDDEAHYAAAAgACEA"
```

The data generated by rapidcheck (or any other property-based testing framework) shoild be further filtered in the test to eliminate incorrect values, or a custom input generator can be written to generate inputs appropriate for the tested unit. The rapidcheck framework provides a `Gen` API to customize input data generation.

3.3.3 Reimplement algorithm by brute-force

Even though the test only expresses the software specification, it may still be difficult to implement. Suppose the following function is specified:

- Parameter `int n`: the number of nodes
- Parameter `vector<pair<int, int> > edges`: each pair is two numbers indicating two node IDs of a directed edge.
- Node IDs are integers in the range $[0, n)$.
- If a Hamiltonian circuit is present, return any possible `vector<int> *` such that the node IDs in the vector represent a Hamiltonian circuit for the graph.
- If a Hamiltonian circuit is not present, return `NULL`.

Assume the function is implemented in a certain polynomial-time algorithm (which is possible given more specifications on the graph, but it is omitted as Hamiltonian circuit algorithm is not the focus of this project).

While it is simple to test for the correctness for returned vectors, it is very difficult to test if returning `NULL` is correct. The specification "Hamiltonian circuit is not present" is very well-defined, but it is still not possible to validate. In this case, a brute-force approach can be used to test for the correctness.

Although the function was implemented in an efficient polynomial-time algorithm, tests do not strictly require a high performance. It is therefore appropriate to test the new algorithm with a potentially less error-prone and more reliable algorithm using brute-force, i.e. NP time. This is because unit tests do not have to be capable of tackling with large data sets; light-weightedness is one of the core concepts of unit testing. (Stress testing is another topic not related to unit testing)

3.3.4 Testing for edge cases

Some common special cases should be included in the test data set explicitly, since they are common sources of error:

- Empty strings, multi-line strings, strings with Unicode, strings woth emoji
- Small numbers: $0, 1, 2, -1, -2, 0.5, -0.5$
- Large numbers: $2^{31} - 1$, $2^{32} - 1$, $2^{63} - 1$, $2^{64} - 1$
- Special floats: `1/0` ($+\infty$), `-1/0` ($-\infty$), `0/0` (`NaN`)

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. By using TDD, it ensures that every part of the program can be tested properly, as nearly all code is writen based on pre-written test cases. It also prevents writing duplicated code in large collaborative projects, and TDD can avoid waste of time for implmenting already done features/requirements.

The workflow of TDD is as follows:

1. Create a new unit test for the software for a requirement that has not yet been implemented.
2. Run the unit test. If the unit test passes, this means the new test overlaps with previously written tests, and that a new test should be written to cover the new requirement
3. Write code to pass (only) the newly added test, while ensuring previous tests also pass
4. Run the tests
5. Repeat the above process for every requirement

Suppose we want to write a calulator with a fucntion `int add(string numbers)` that has the following requirements:

1. The method can take in any amount of space-separated integers as an input, and return their sum.
2. If the string is empty, the method should return `0`.
3. If non-space-and-digit characters are passed within the string, the method throw an "Invalid character" exception.

We begin the development process by writing the test for the first requirement as follows:

Remark: C++

test

As the `sum` method is currently empty, the test fails as expected. So we can proceed to write the code which solves this test:

```
#include <string>
#include <sstream>
#include "calculator.h"

int add(string numbers){
    stringstream ss (numbers); int current, ans = 0;
    while (ss >> current)
        ans += current;
    return ans;
}
```

To verify our implementation, we run the test we have written in test 1:

Remark: Run
test, success

And we can continue with the next test.

Exercise 1: Write the test and the code for the second test.

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.

The main difference between TDD and BDD is that while TDD ensures that every sub-component of the software is correct, BDD provides a more "high-level" view of what the entire software should provide to the user. It is often regarded as a "bridge" between product designers and developers, as through BDD, product designers can effectively communicate their needs to devlopers who specialize in implementing those requirements.

BDD defines the behaviour of a function using some specific terms or keywords. Each of these is written in the format of a "user story", and follow special "syntax" via the usage of certain keywords. The usage of these keywords are important, as a user story can be automatically phrased into "testable" code with a framework (such as Cucumber, on next section), given that it is formatted correctly with those "grammar". Here we present some of the keywords, along with the syntax general to most BDD frameworks:

The following is an example BDD of Cucumber:

```
Scenario Outline: eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

  Examples:
    | start | eat | left |
    |    12 |   5 |    7 |
    |    20 |   5 |   15 |
```

Scenario Outline Describes a situation where that the software should be able to handle properly, defines the behavior of the program for a certain situation.

Given The initial state of the program when this scenario takes place

When Similar to natural English, "When" implies that the following statement should be done "when" this statement is completed

Then Describes the expected output of the program after the preceeding "when" statement is fulfilled.

The code above are extracted from the official Cucumber Documentation, which you can access at https://docs.cucumber.io/gherkin/reference/

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

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