COMP2123 self-learning report
Unit testing

Chan Kwan Yin (3035466978), Lee Chun Yin (3035469140)

December 23, 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:

```
#ifndef ASSERT_H
#define ASSERT_H

#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; \
```

```
    } \
}

#endif
```

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:

```cpp
#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:

```cpp
#include <cmath>
#include <vector>
#include <rapidcheck.h>
#include "sciNot5.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=9745278035080689903

- requirement 2: 3 significant figures
Falsifiable after 4 tests and 1 shrink

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=BQiclFXdpJXZtVmb0BiM6AyMgMXan5Waml2Yh5GdgYWanVnclN374amf96iPH+Oum5Xvu4zhvjrZ
    +1rL+c474amf96iPHiAAEMAAAAQABA"
```

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 (The emoji issue actually caused major bugs in iOS for several times)
- 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`)

Exercise: Implement a generator that yields both random and edge-case values for all primitive data types

Exercise: Using the edge cases method, find out the bug in `sciNot5`

4 Unit testing techniques

4.1 Code 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.

In GNU C++, code coverage is supported by the `gcov` utility. First, the `-fprofile-arcs -ftest-coverage` flag should be provided both when compiling source files of interest into intermediate object files and when compiling the object files into final executable files.

```
g++ -c -o fooBar.a -fprofile-arcs -ftest-coverage fooBar.cpp
g++ -o fooBar fooBar.a qux.a -fprofile-arcs -ftest-coverage
```

When the executable `fooBar` is executed, the file `fooBar.gcda` is also generated, containing the statistics during the execution. Coverage reports can hence be generated:

```
gcov fooBar
```

Applying code coverage into `sciNot5` from the previous section, 70 gcov reports are generated:

```
Any.hpp.gcov
ApplyTuple.h.gcov
Arbitrary.hpp.gcov
Assertions.hpp.gcov
Capture.h.gcov
Check.hpp.gcov
Common.hpp.gcov
Container.hpp.gcov
Create.hpp.gcov
ExecRaw.hpp.gcov
```

And way more. This is because every header or source file included in the compilation of `sciNot5.a` is flagged for coverage reporting. The file `sciNot5.cpp.gcov` might be of more interest:

```
        -:    0:Source:sciNot5.cpp
        -:    0:Programs:6
        -:    1:#include <cmath>
        -:    2:#include "SciNotation.h"
        -:    3:
```

```
     2644:    4:SciNotation sciNot(double number) {
     2644:    5:    int sign = number >= 0 ? 1 : -1;
     2644:    6:    number *= sign;
     2644:    7:    int exp = (int) floor(log10(number)) - 2;
     2644:    8:    int digits = (int) round(number * pow(10.0, -exp));
     2644:    9:    return SciNotation{digits * sign, exp};
        -:   10:}
        -:   11:
```

All lines in `sciNot(double)` got called 2629 times, and the file has been tested with full coverage. While this is obvious, coverage may be more useful when the file has multiple conditions, loops or even error conditions:

```
#include <string>
#include "fibo1.h"

int fibo(int times) {
    int a = 1, b = 1;
    if(times <= 0) throw std::string("input out of bounds"); // error value
    if(times <= 2) return 1; // init condition
    int i = 3;
    while(true) {
        int tmp = a + b;
        a = b;
        b = tmp;
        if(i == times) break;
        i++;
    }
    return b;
}
```

```
#include "assert.h"
#include "assert_int.h"
#include "fibo1.h"

void testFibo() {
    ASSERT_EQUAL(fibo(2), 1)
    ASSERT_EQUAL(fibo(3), 2)
}
```

```
        -:    0:Source:fibo1.cpp
        -:    0:Programs:6
        -:    1:#include <string>
        -:    2:#include "fibo1.h"
        -:    3:
        2:    4:int fibo(int times) {
        2:    5:    int a = 1, b = 1;
       2*:    6:    if(times <= 0) throw std::string("input out of bounds"); // error value
        2:    7:    if(times <= 2) return 1; // init condition
        1:    8:    int i = 3;
        -:    9:    while(true) {
        1:   10:        int tmp = a + b;
        1:   11:        a = b;
        1:   12:        b = tmp;
        1:   13:        if(i == times) break;
    #####:   14:        i++;
    #####:   15:    }
        1:   16:    return b;
        -:   17:}
```

Apparently, the error condition is never run. The init condition is run once, and the loop is only looped once.

This shows that some lines are not run yet. Let's add more test cases to cover everything!

```
#include <string>
#include "assert.h"
#include "assert_int.h"
#include "fibo2.h"
```

```

void testFibo() {
    bool ok = false;
    try {
        fibo(-1);
    } catch(std::string error) {
        ok = true;
    }
    if(!ok) {
        std::cerr << "fibo(-1) does not throw an exception!" << std::endl;
    }
    ASSERT_EQUAL(fibo(2), 1)
    ASSERT_EQUAL(fibo(3), 2)
    ASSERT_EQUAL(fibo(5), 5)
}
```

```
        -:    0:Source:fibo2.cpp
        -:    0:Programs:6
        -:    1:#include <string>
        -:    2:#include "fibo2.h"
        -:    3:
        4:    4:int fibo(int times) {
        4:    5:    int a = 1, b = 1;
        4:    6:    if(times <= 0) throw std::string("input out of bounds"); // error value
        3:    7:    if(times <= 2) return 1; // init condition
        2:    8:    int i = 3;
        -:    9:    while(true) {
        4:   10:        int tmp = a + b;
        4:   11:        a = b;
        4:   12:        b = tmp;
        4:   13:        if(i == times) break;
        2:   14:        i++;
        2:   15:    }
        2:   16:    return b;
        -:   17:}
```

Now all lines are covered.

While it is cool to have full code coverage, it is not always practical, especially when a test has to be written for a specific error condition that is very obvious. Always aiming for minimal code and full code coverage is called "extreme programming", which is usually not a feasible practice.

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:

```
#include "Calculator.h"
#include "assert.h"
#include <string>

using namespace std;

void testSum() {
    Calculator c
    ASSERT_EQUAL(c.add("1 2 3"), 6);
}
```

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"
using namespace std;

int Calculator::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:

```
Executing test testSum
Successful: c.add("1 2 3") = 6
```

And we can continue with implementing the next requirement.
Exercise: Write the test and implementation 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:

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario: Sunday isn't Friday
    Given today is Sunday
    When I ask whether it's Friday yet
    Then I should be told "Nope"
```

Feature Describes a feature that the software should provide on the high-level, with some description

Scenario Describes a situation (similar to example) where that the software should be able to handle properly, defines the behavior of the program for a certain (more specific) 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.

Why is such grammar important? As a software developer, we may often find it difficult to translate requirements given by product designers/users into code. By having the product designers write their requirements in such a fashion which is both "human readable" and "machine prasable", it is possible to translate those requirements in to unit tests **automatically**. For example, in Cucumber, the following can be generated automatically from the above BDD, which can be further manipulated to contain actual machine-runnable tests (in fact a BDD is just a combination of multiple unit tests, but written in a high-level manner):

```
@Given("^today is Sunday$")
public void today_is_Sunday() {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^I ask whether it's Friday yet$")
public void i_ask_whether_it_s_Friday_yet() {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^I should be told \"([^\"]*)\"$")
public void i_should_be_told(String arg1) {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```

If you are interested to learn more, head over to http://docs.cucumber.io/guides/10-minute-tutorial/ to get started with the Cucumber framework!

4.4 Summary: TDD vs BDD

As you can see, the description of the program behavior in BDD is more "high level" compared to TDD. The person writing BDDs ususally writes them from a user's perspective, where they might not have full knowledge on the internals of the software. On the other hand, TDDs focus more on the internal working on the software, and whether each part functions correctly.

Often in BDD, when a test fails, it is difficult to locate the root cause of the error, as the BDD usually involves the whole piece of software instead of a single modular component and it is difficult to locate a bug in the whole program as opposed to a single function. (in Cucumber, the framework subclassifies a task into multiple components to help to identify the exact source of error, but the subclassification is not as small as those in TDD). Furthermore, as BDD involves running the whole software to conduct behavior tests, it takes much longer to conduct a single test (a BDD test can take up to a minute, while a TDD test may only take a few seconds).

The existence of BDDs is to keep developers on track on building products that directly address the user's requirements, instead of building functions that satisfy a low-level requirement. This means, that in BDD the software development is driven by how the user will use the software, and the program is built accordingly, while for TDD, it involves the concept where the correctness of a software implies that all sub-components

are correctly writen, meaning that code is built according to the feature of a certain component, but not the whole picture.

This means if a BDD test fails, it implies that there is a use case that the software will have a bug in regular use, and should not be released. On the other hand, if BDD passes and a TDD fails, this implies that although some components may contain bugs, in regular usage, they will not affect the running of the whole software, and the software can continue with its release, leaving the bug fix for later.

4.5 Dependency mocking

Often (if not always), sub-components in a software will have other components as dependencies. Take our calculator class as an example. Suppose we want to write a function `int eval(int a, int b, char operation)`

```
#include "Calculator.h"

int Calculator::eval(int a, int b, char operation){
    if (operation == '+') return a + b;
    if (operation == '-') return a - b;
    if (operation == '*') return a * b;
    if (operation == '/') return a / b;
}
```

The implementation is done, and we have unit tests for this code. Now lets say that there is a new requirement, that if the result of the calculation exceeds the value of `int` ($2^{31} - 1$), it should throw an OverflowException. A trivial idea for implementation would be to implement the internal calcualtion of the operation via another data type that supports larger range (e.g. `long long`, `BigInteger`), and compare this value with $2^{31} - 1$. In other words, we can write the following instead:

```
#include "Calculator.h"
#include "BigInteger.h"

int Calculator::eval(int a, int b, char operation){
    BigInteger bigA(a), BigInteger bigB(b);
    BigInteger result(0);

    if (operation == '+') result = bigA + bigB;
    if (operation == '-') result = bigA + bigB;
    if (operation == '*') result = bigA + bigB;
    if (operation == '/') result = bigA + bigB;

    if (result > BigInteger((1<<31)-1))
        throw std::overflow_error("too big");
    else
        return result.value();
}
```

The `Calculator` now depends on `BigInteger`. Assume that we have not implemented or tested our implementation of `BigInteger` yet, how are we going to test the component of `Calculator`? One of the ways is to substitute `BigInteger` with another "fake" BigInteger, which for the purpose of explanation, we assume the implementation is correct.

```
#ifndef BIGINTEGER_H
#define BIGINTEGER_H
#include <string>
using namespace std;

class BigInteger {
    long long val;
    public:
        BigInteger(int a){
            val = a;
        }
        BigInteger operator+(BigInteger rhs){
            return val + rhs.val;
        }
```

```
        BigInteger operator-(BigInteger rhs){
            return val - rhs.val;
        }
        BigInteger operator*(BigInteger rhs){
            return val * rhs.val;
        }
        BigInteger operator/(BigInteger rhs){
            return val / rhs.val;
        }
        bool operator>(BigInteger rhs){
            // implementation omitted
        }
        int value(){
            return (int) val;
        }
};

#endif
```

Now our `Calculator` class can compile successfully, and we can test the implementation of `Calculator` without caring about the correctness of `BigInteger`.

This is exactly what "Dependency mocking" is. If inside the target we want to perform unit tests on, has a function call which either is untested, or we do not want to call (due to lack of resources, take long time to run), we can essentially substitute that function with another one which produces the "expected result" of the original function, assuming that the implementation of the original function was correct. Mocking is useful when it is impractical to call a function many time during the testing stage of development, such as those which consume a lot of time like database calls.

However, although mocking effectively isolates the target component from other components, making unit testing relevant to the implementation of only the target, mocking has several disadvantages for software development. By essentially "negelecting" the implementation of other components, it makes the target component prone to unexpected behavior of the target component. For instance, if there were any updates to the dependencies, such as a new exception being thrown, the current unit tests would not be aware of such new behavior, and the tests may not be accurate. For example, the unit test may run successfully with mocks, but with the actual "unmocked" implementation, it has a runtime error, and the developer may be unaware to this. This is particularly important in large projects with many dependencies, perhaps with dependencies not being written by the in-house developers, and it is difficult to update mocks to fully reflect the current behavior of the dependencies.

If you want to know more about mocking and its advantages/disadvantages, check out https://enterprisecraftsmanship.com/2016/06/09/styles-of-unit-testing/.

5 Difficulties

5.1 Coupling

Coupling is the cyclic dependency between units to be tested.

In large software projects, components are usually dependent on one another. For example, the dependency graph of a web server might look like figure 1.

A relation worth noting is between logging and i18n. I18n requires a logger to log its loading process, and logger requires i18n to display proper messages. While this is complicated to write from the beginning, it is even harder to unit test, because it is not possible to only unit test logger without having a functional i18n component.

In other words, it is not possible to decide which component to unit test first, because both modules are mutually dependent.

This condition is called modular coupling. Common practices to prevent it include:

Abstraction : Create an interface for logger and simulate a simple dummy logger to unit test i18n. Create an interface for i18n and simulate a constant dummy i18n system to unit test logger.

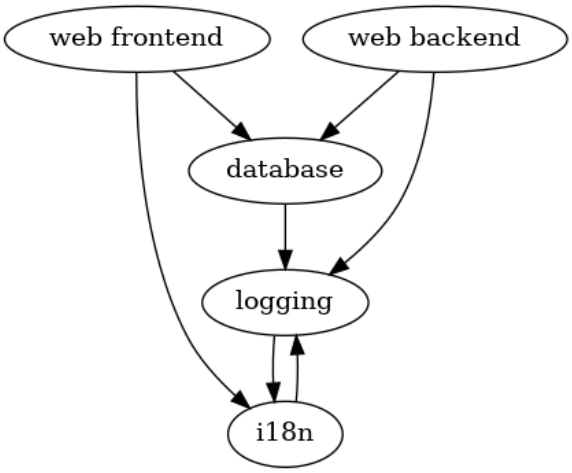

Figure 1: Dependency graph of a web server

Dependency mocking As explained in the previous section, dependency mocking can be used to prevent dependencies while testing, but there are many problems about dependency mocking too.

Therefore, the best solution to coupling is to prevent cyclic dependencies from the beginning.

6 Sources

The source for building this report and all test cases can be found at https://github.com/chankyin/comp2123-project