7COM1025
Programming for Software Engineers
Lecture 25

Dr. Renato C. de Amorim

REQUIRED READING

This class relates to chapter 10: versioning
Book:

Reddy, M. (2011). API Design for C++. Elsevier.

Pages: 291 - 328

TESTING

Every good software developer understand the need to write test cases.

If your code is buggy, unpredictable or just slow, clients will look for alternative solutions.

Why test:

- Increase confidence. You will be confident that your changes won't break client's code. It also reduces fear of implementing change.
- Ensuring backward compatibility.
- Saving costs. Fixing defects early will save you money. Some estimates state that fixing a bug after release can be 10-25 times more costly than fixing during development (read your book).
- Codify use cases. Developing tests for use cases before their implementation will help you to know when the required functionality has been achieved.

-Compliance assurance. Certain safety or security-critical applications may have to pass regulatory tests.

Note: you will need to maintain the test cases.

API TESTING

API testing has some similarities with testing end-user software.

White box testing. Tests are developed with knowledge of the soruce code and are normally written using a programming language.

Black box testing. Tests are based on product specifications and without any knowledge of the underlying implementation.

Gray box testing. A combination of the above.

Unit testing. Used to verify a single minimal unit of source code, such as an individual method or class. This is a white box technique used to verify the behaviour of functions and classes in isolation.

Integration testing. It's concerned with the interaction of several components cooperating together. This is a black box technique.

QUALITIES OF A GOOD TEST

You should always bear in mind the qualities below.

Fast. Your suite of tests should run very quickly so that you get rapid feedback on test failures.

Stable. Tests should be repeatable, independent and consistent: every time you run a specific test you should get the same result.

Portable. If your API is multi platform, so should be your tests.

High coding standards. Test code should follow the same coding standard as the rest of your API

- You should not slip standards just because the code will not be run directly by your users.

Reproducible failure. If a test fails, it should be easy to reproduce this failure. This means logging as much information as possible.

WHAT TO TEST

There are standard QA techniques that you can employ to test your API, such as:

- Condition testing. Exercise all combinations of if/else, for, while and switch expressions.

- Equivalence classes. An equivalence class is a set of test inputs that all have the same expected behaviour. We need different such sets that will lead to different classes of behaviour. Eg. if you have a function that works for an input from 0 to 65535, there are three equivalence classes: negative numbers, numbers in the range and numbers above the range.

- Boundary conditions. Most errors occur around the boundary of expected values. Eg. if inserting a element in a linked list of size n, test inserting at position 0, 1, n-1 and n (at least).

- Parameter testing. Test all possible parameter combinations.
- Return value assertion. As above, but focusing on the correctness of the returned values.
- Getter/Setter pairs. Test all pairs, including getters that have not been previously set (default value).
- Negative testing. Force error conditions and see how the code reacts to unexpected situations.
- Buffer overruns. Check all buffers and make sure your API does not write to memory beyond the size of a buffer.

Etc. (check chapter 8 in the book!)

WRITING TESTABLE CODE

Test-driven development
Involves writing automated tests to verify desired functionality before the code that implements this functionality is written.

This helps you to stay focused on the key use cases for your API. It forces you to think about your API before you start writing its code.

This approach does not need to be used solely in the initial development of your API. It can also be helpful in the maintenance stage.

Stub and Mock objects
Create test objects that can stand for real objects in the system.

This lets you substitute an unpredictable resource with a lightweight controllable replacement for the purpose of testing.

You can also use these test objects to simulate error conditions that are difficult to simulate in the real system.

WRITING TESTABLE CODE

Assertions
You can use them to verify assumptions your code makes,
You can use them to document and verify programming errors that should never occur.

```
#include<cassert>
#include<iostream>
class MyClass{
private:
    std::string* mStrPtr=NULL;
public:
    MyClass(): mStrPtr(new std::string("Hello")){}

    void PrintString(){
    assert(mStrPtr!=NULL);
    std::cout<<*mStrPtr<<std::endl;}
};
int main(){
    MyClass test;
    test.PrintString();
    return 0;
}
```

What happens if the constructor couldn't initialise mSTRPtr?

When you compile you will get an error:

```
 void MyClass::PrintString(): Assertion `mStrPtr!=NULL' failed.
```