Unit Testing, Contracts, Debugger

John Businge

January 17, 2023

Brief info about Unit Testing and Contracts can be found at the end of this document.

Google C++ Testing Framework is a framework that allows you to run your code to test efficiently.

We will also use simple macros to teach working with contracts.

Exercise 1: Install The Testing Framework

Follow the steps described in Loading TicTacToe in CLion, to start your testing framework and create a new project.

Follow the following tutorial: on a quick introduction to the Google C++ Testing Framework.

Exercise 2: What would you test?

First read the explanation in Appendix A about Unit Testing.

a) Consider the following function:

```
int largest(vector<int> list){
/* code */
}
```

Write at least 5 tests using the Google Test framework for this function. Think of General Correctness, Edge/Corner Cases and Errors (lookup what they mean in software testing terms).

Consider the following (incorrect) implementation of the above function. How much of the tests you wrote before fail on this? In Appendix C you can create a find implementation that works better.

```
int largest(vector<int> list){
    int max = numeric_limits<int>::max();
    for(int i = 0; i < list.size() - 1; i++){
        if(list[i] > max){
            max = list[i];
        }
    }
    return max;
}
```

b) Consider the `Fighterplane` class:

(in the `Fighterplane.cpp` file)

Write tests in the Google Test framework for each method of this class.

Exercise 3: Pre- and Post Conditions

Read the explanation in Appendix B about Contracts.

a) Consider the function from Exercise 2a:

Think of suitable pre- and post-conditions for this function.

b) Consider the class in Exercise 2b:

Think of appropriate pre- and postconditions for each method in that class.

Use the header file: DesignByContract.h to implement pre-conditions (REQUIRE) and post-conditions (ENSURE).

Appendix A: Unit Testing

Software Testing aims to evaluate a particular feature or functionality of a program or system and verify that it matches the expected result. There are different types of Software Testing:

- **Unit Testing:** Verifies the functionality of a specific snippet of code, usually on a single class or a single method.
- **Integration Testing:** Tests the cooperation of the modules.
- **System Testing**: Tests the fully integrated system.
- **System Integration Testing**: Tests the cooperation between different systems.

A unit test is code that tests a very small specific portion of the functionality of testing a system. In most cases, a test is going to perform 'one' method in a certain context. Of course, this does not mean that every method is tested by one test. You are going to write different tests for the same method in different contexts.

A good Unit Test is:

- **Repeatable:** Every time the test is run, it should obtain the same result. (avoid randomness, use of "current time", etc...)
- **Independent:** Test one method at a time. Different tests are not allowed to depend on each other.
- Valuable: Testing simple getter/setter methods is not real valuable.
- **Thorough:** Test ALL pre/post conditions, edge cases, . . .
- **Reliable:** Unit tests should fail only if there's a bug in the system under test. That seems pretty obvious, but programmers often run into an issue when their tests fail even when no bugs were introduced.
- **Readable:** The intent of a unit test should be clear. A good unit test tells a story about some behavioral aspect of our application, so it should be easy to understand which scenario is being tested and — if the test fails — easy to detect how to address the problem.
- **Reliable:** Unit tests should fail only if there's a bug in the system under test. That seems pretty obvious, but programmers often run into an issue when their tests fail even when no bugs were introduced. For example, tests may pass when running one-by-one, but fail when running the whole test suite, or pass on our development machine and fail on the continuous integration server. These situations are indicative of a design flaw.

To test thoroughly, you need to check a few things:

- **General correctness:** These are usually the simplest tests to write. These test the general "good" behavior of the use cases.
- **Edge cases**:
  - *Order*: Does a different order have an effect on the result?
  - *Range*: How does it respond to zero, the minimum, the maximum, positive values, negative values, . . .
  - *Existence*: What if you pass a nullptr as a parameter? What about empty collections (empty array, empty list, empty vector, . . .), what about empty Strings.
  - *Cardinality*: What is the expected number of items?
- **Errors**: Are the correct error messages being given? What about I/O issues, such as missing files, unreadable or empty files?

Appendix b: Contracts

Contracts are used to ensure that your software is correctly built. It is a way of formally documenting the functionality of a particular function or method. You can define this programmatically using asserts (or, in our case, with the macros ENSURE and REQUIRE, defined in DesignByContract.h). Look at how contracts are implemented, for example, in the `class--TicTacToe.cpp` of the project `TicTacToe12`.

We use `REQUIRE` for pre-conditions and `ENSURE` for post-conditions. `REQUIRE` determines what state the system must be in before calling the function or method. `ENSURE` records how the state of the system will be changed after calling the function or method.

It is the job of the code calling the function or method to ensure that the system is in the correct condition (as described in the precondition). So you no longer have to implement extra checks in the function or method itself and generate error messages when it comes from the wrong state called. So these checks and error messages must be implemented on the calling side.

Each side of a method/function call benefits from a contract, but must fulfill its obligations:

Contract	**Advantages**	**Obligations**
Method or function pre- and post- conditions (the PROVIDER)	- No need to check the input values. - The input is guaranteed to meet the precondition	Must ensure that the post condition is met.
Code that calls the provider (the CLIENT)	- No need to check the output values. - The result is guaranteed to meet the post condition.	Must ensure that the Provider's preconditions are met.

Appendix C: A better implementation for the function largest

```
#include <exception>
#include <vector>

using namespace std;
class IllegalArgException: public exception{
    const char* message;
    public:
        IllegalArgException(const char * message){
        this->message = message;
    }
    virtual const char* what() const throw(){
        return message;
    }
};

int largest2(vector<int>* list) {
    if (list == nullptr) {
        throw new IllegalArgException("list cannot be nullptr");
    } else if (list->empty()) {
        throw new IllegalArgException("list cannot be empty");
    }
    int max = numeric_limits<int>::min();
    for (int i = 0; i < list->size(); i++) {
        if (list->at(i) > max) {
            max = list->at(i);
        }
    }
    return max;
}
```