UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

COP 4600

Ex4: Unit Testing

Overview

When writing code, particularly for complex systems, it is imperative to continuously check that your work is correct. Unit tests allow developers to write tests that demonstrate that output matches specifications, and mitigate the risk that changes in one area of the codebase cause unintended consequences.

Structure

This exercise is broken into 3 parts:

1. **Introduction to GoogleTest**. You will build and install the unit testing framework.

2. **White box testing**. In white box testing, tests are written with full knowledge of the implementation of the system; including knowledge of weak points and access to internal data structures and functions.

3. **Black box testing**. In black box testing, tests are written with no knowledge of the implementation of the system. Rather than testing internals, tests are written to ensure correct outputs based on a specification.

Part 1: Introduction to GoogleTest

Installing GoogleTest

Before beginning the exercise, you will first install **GoogleTest** – a unit testing framework for C++. GoogleTest is necessary to complete the remainder of the assignment.

You will begin by running the following sequence of commands:

```
sudo apt-get install cmake
git clone https://github.com/google/googletest.git -b v1.13.0
cd googletest                    # Main directory of the cloned repository.
mkdir build                      # Create a directory to hold the build output.
cd build
cmake .. -DBUILD_GMOCK=OFF       # Generate makefile for GoogleTest

make
sudo make install                # Install GoogleTest in /usr/local/
```

Exercise Files

The source files for this exercise can be obtained by running in the home directory

```
git clone https://github.com/ernestoamujica/OS-Ex4.git
```

Anatomy of a Unit Test

A unit test is a function which verifies the output or value of other code (known as the subject under test). This is achieved through an assertion. An assertion is a statement, which, if false, indicates that the test has failed. Unit tests are compiled along with the subject under test into a testing binary. The testing binary is then executed.

As an example, suppose the subject under test is the following function in `Factorial.h`:

```
int Factorial(int n);
```

Let us consider some simple unit tests for this integer function, written in `FactorialTests.cpp`. Unit tests in GoogleTest begin with the `TEST` preprocessor macro.

```
#include "gtest/gtest.h"
#include "Factorial.h"

TEST(FactorialTest, HandlesZeroInput) {
  int result = Factorial(0);
  ASSERT_EQ(result, 1);  //Assertion
}
```

Tests are organized into a "testing suite" (which simply denote groups of related tests). Here the testing suite name is `FactorialTest`. When the testing binary is run, the output will be organized by each testing suite.

Each test also contains a second field denoting the name of the test. Here the testing suite name is `HandlesZeroInput`. Fields must be valid C++ function name variables (no underscores).

The body of the test contains some code followed with an assertion. The test will pass if all assertions in the body pass.

You will not write a `main` entry point for your testing binary. Instead, you will link with the GoogleTest library which includes one for you.

We've already built a `Makefile` for this test program. To run this sample test, simply navigate to the "Part 1" folder and run the commands

```
make
./test.out
```

```
reptilian@localhost:~/OS-Ex4/Part1$ ./test.out
Running main() from /home/reptilian/googletest/googletest/src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from FactorialTest
[ RUN      ] FactorialTest.HandlesZeroInput
[       OK ] FactorialTest.HandlesZeroInput (0 ms)
[----------] 1 test from FactorialTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (2 ms total)
[  PASSED  ] 1 test.
reptilian@localhost:~/OS-Ex4/Part1$
```

Results of running the testing binary

Part 2: White box Testing

In this part of the exercise, you will be writing tests for the following function, whose implementation you are given:

```
//Returns true if a divides b
//Otherwise, returns false

bool isDivisible(int a, int b) {
    return (b % a == 0);
}
```

A significant challenge when writing tests is deciding how many tests are needed. Since we can't test every integer, we will have to test a few values that represent the functionality of the whole system.

For this exercise you will write the following tests:

- A is positive, B is positive, and A does divide B. The function should return true.
- A is positive, B is positive, and A does not divide B. The function should return false.
- A is positive, B is zero. The function should return true.
- A is negative, B is positive, and A does divide B. The function should return true.

Overall, there are 18 total tests that are needed to completely test this simple function! For this assignment, simply write the 4 tests requested. All of your tests should pass.

We've already built a `Makefile` for this program. To run your tests, simply navigate to the "Part 2" folder and run the commands

```
make
./test.out
```

Submit to Canvas:

(1) a screenshot showing the execution of your tests. The screenshot should be named `Part2_Lastname_Firstname.png`

(2) the source file containing your tests named `DivisibilityTests.cpp`

Part 3: Black box Testing

Often before development of a system begins, tests are first written which demonstrate the specified functionality of the system. In this exercise, you will be writing black box tests for the backend of a simple airplane trip booking system managing flights for a given date. You will not be implementing this system.

The architecture of the system is as follows:

1. An `Airport` is an object consisting of a 3 letter airport code representing a unique airport.

2. A `Flight` is an object that represents a scheduled flight from one `Airport` to another. It describes a source `Airport`, a destination `Airport`, a departure time (represented as an integer denoting the hour of takeoff in UTC), and a capacity (represented as an integer denoting the remaining seats available for purchase). It is assumed that all flights depart on the same date.

3. A `Booking` is an object that contains a list of `Flight` objects. It represents a single flight option to get from a source `Airport` to a destination `Airport`. It contains a single `Flight` object in the case of a direct flight; and two `Flight` objects in the case of a connection.

You will write unit tests for each function provided in `BackEnd.h`. You will use the provided interfaces to test the following specifications:

1. A query is a request made by an end user to view all available flight bookings from a source airport to a destination airport.

2. A flight booking cannot include any flights that are full.

3. If a flight booking is purchased, the capacity of all flights in the booking should be decremented.

4. For a given booking, a maximum of one connection is possible. A connection occurs only when there are no direct flights between the destination and source airports, and two flights; one from the source airport to the connection airport, and a second flight from the connection airport to the destination airport are available to travel from the source to the destination. The second flight's departure must occur at least 2 hours after the first flight's departure.

5. If multiple flight bookings are available for the query, bookings should be ordered by departure time, with the earliest flight first. In the case that a booking is a connection, then the departure time of the earlier flight shall be used for ordering purposes.

Note: The "subject under test" (SUT) are the functions in `BackEnd.h`. Assume that the all the other classes are correct. You may freely use any function you would like in any of the classes provided (including any constructors).

Your unit tests will be written in a file called `UnitTests.cpp` (located in "Part 3" in the exercise source repository cloned in Part 1) which will contain your test definitions.

Your unit tests should fully test the specifications of the system. In particular, your unit tests will need to

(1) Create `Airport` instances

(2) Create `Flight` instances

(3) Using the objects created in (1) and (2), demonstrate that the booking system produces all correct `Booking` objects sorted in order when using `flightBookingQuery`

(4) Demonstrate that the booking system correctly processes `Booking` objects when using `purchaseBooking`, and that subsequent `flightBookingQuery` calls adhere to the specification.

We have provided a dummy implementation of `BackEnd.cpp` so you can verify that your tests properly compile, link, and execute. Of course, many of your tests should fail since the dummy version of `BackEnd.cpp` does not adhere to the specification.

We will grade the quality of your tests by recompiling your tests on different implementations of `BackEnd.cpp`! Some of these implementations are correct; some are incorrect. Your unit tests should <u>all</u> pass on the proper implementation; and <u>at least one</u> test should fail on the incorrect implementations. You must write a sufficient number of unit tests to completely cover all items of the specification.

You may only modify `UnitTests.cpp`. Assume that all inputs to the functions in `BackEnd.cpp` are valid (i.e. do not test passing invalid strings or `nullptr`). Your tests must be fully automated (must run with no command line args/user input).

We've already built a `Makefile` for this program. To run your tests, simply navigate to the "Part 3" folder and run the commands

```
make
./test.out
```

Submit to Canvas:

(1) your source file `UnitTests.cpp`