CS2303 In Class Exercises

Jan 23, 2020

January 10, 2020

Abstract

Today's goal is that you can build a test suite for the functions in your production code.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 Functions in Production Code

The messages in the sequence diagram are realized in C code as function invocations.
The invocation implies the existence of a prototype and a function definition, in the production code.
For functions we code, we are responsible for all three.

1.1 Test functions

In test-driven development, (which is preferred in industry, and gets the development job done faster) the existence of a production function implies the existence of a test function for that production function. We use a compartment for test functions, because compartments help us focus our efforts.

There is a process by which code is developed in the test-driven development context. Suppose the prototype of a production function (call it functionA) is known. This is not an unreasonable assumption, because the sequence diagram, which is started (if not finished) before code development, contains the information for the function prototype, namely the parameters, including datatype, and the returned information's datatype. The prototype tells us how the function is to be invoked: we must supply arguments to match the parameters.

1.2 Stub Implementation of Production Function

We will exercise this invocation of the production function, functionA, prior to completing the implementation of functionA. We will create (in the .c file of the

compartment in which functionA will reside when complete) a stub function. A stub function has the first line, the curly braces, and, if it returns anything, that return is present. See the example.

Example:

```
bool functionA(int a, double b)
{
    bool areEqual = true;
    //TODO: this stub needs to be filled out
    return areEqual;
}
```

Note in the above example that the first line of the function, and the curly braces are present as they will be in the finished function definition. Note also that, as this function returns a value, a variable of the appropriate datatype is declared and initialized, and returned. The placeholder for later development is marked with a comment. This stub always returns true.

An example invocation of this function is:
bool answer = functionA(7, 24.3);

The values 7 and 24.3 constitute a test case, because they are a set of argument values.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Write a stub implementation for a function that returns an integer, is called theRightInteger, and takes three parameters, an integer, a double and a pointer to integer (int*).

```
int theRightInteger(int x, double d, int* intP){
int answer = -1;
//do some stuff
return answer;
}
```

2. Write a stub implementation for a function that returns a pointer to a user-defined datatype called Room, is called getNextRoom, and takes a single parameter, of user-defined type LLNode*.

```
room* getNextRoom(LLNode*){
room newRoom = (...)
return newRoom*;
}
```

```
LLNode* checkList(LLNode* llnode, room* room){
LLNode something = (...);
//do some stuff
return something*;}
```

3. Write a stub implementation for a function that returns a pointer to LLNode, is called checkList, and takes two parameters, an LLNode pointer and a Room pointer.

1.3 Test cases

For most test functions it makes sense to have multiple test cases. Suppose the right answer for "functionA(7, 24.3);" is true. Then our test function would not be able to distinguish a working implementation from the stub implementation. We also need a test case, the correct answer of which is false. Then, invoking with that set of arguments would reveal that the stub implementation was inadequate to the task of the full implementation.

Test cases are a valuable component of software development that the software industry maintains, often over a much longer interval than that of any particular product. Nevertheless, test cases are expensive to maintain, and should be meaningful. Often test cases demonstrate the extent of the set of values that a parameter is expected to support. If we think of a "space" being defined by using each parameter as a dimension, then sets of parameters could be used to define the boundaries of that space. A simple example is that a function might not expect a parameter to be negative. If defensive programming is used, the program will reject an inappropriately negative input and return a rejection response. The test function invoking with a negative argument can check for that rejection.

It is worth emphasizing that for a test case, we must have an independent means of knowing the right answer. Then the test function compares the answer it receives from the invocation with the correct answer, to determine whether the implementation has passed the test case.

There is more guidance on test cases besides that more than one answer should be elicited from the implementation, and that the parameter space should be explored. There is also the idea of satisfying the problem statement. Though it is often infeasible to test every case, it is desirable to consider how the overall problem statement influences the needs for this function under test. The goal is, a deserved sense of confidence that the function under test meets its requirements.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. There is a function that calculates the day of the week from the date. (See `https://en.wikipedia.org/wiki/Determination_of_the_day_of_the_week`.) Gauss's algorithm works for a certain date range (years greater than 1582). Suppose you had to check an implementation of this algorithm. Describe some test cases for this algorithm. For example, would you test a leap year? A standard year? Both? Are there particular days of the year you might check?

```
Feb 29th (not a leap year)
Jan 1st of this year
January 40th
```

1 cell alone
3 cells next to each other
8 cells in donut shape
full board

2. Suppose you are working on a two dimensional grid, carrying out Conroy's "Game of Life", which you can read about at `https://en.wikipedia.org/wiki/Conway\%27s_Game_of_Life`. Given that the rules for a cell being alive or dead are:
   Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:
   Any live cell with two or three neighbors survives.
   Any dead cell with three live neighbors becomes a live cell.
   All other live cells die in the next generation. Similarly, all other dead cells stay dead.

   Which situations would you pose as test cases?

1.4 Test-driven Development Process

Our implementation of the test-driven development code has a list of tests, as well as individual test functions.

Example:

```
bool tests()
{
    bool answer = false;
    bool ok1 = testReadFile();
    bool ok2 = testGotAdjacencyMatrix();
    bool ok3 = testSomethingElse();
    bool ok4 = testRemoveFromList();
    answer = ok1 && ok2 && ok3 && ok4;
    return answer;
}
```

The example above shows a list of tests. Each test is invoked. The result is only true if every test function returned true. Test functions should only return true if every test case within the test function got the right answer.

The test-driven development process has steps:

1. Start with the function prototype.
2. Write a stub implementation for the function.
3. Prepare to write a test function, and put its invocation in the test suite list (as in the example above).
4. Create values for the parameters, for which the correct answer to function invocation is known.
5. Within the test function, invoke the function with each such set of parameters.

6. Each time the function returns, compare the returned answer with the known correct answer.

7. If the test function shows all test cases passing, while the implementation is only a stub, the test function, i.e., the set of test cases, is inadequate.

8. The test function should fail, correctly detecting that the stub implementation is inadequate. (At this point in development, during homework, we take a screenshot of the stub code showing the failing test results.)

9. Next, complete the implementation, updating the stub to a full implementation.

10. Run the test suite again.

11. If all test cases pass, the test function returns a true, meaning the implementation has satisfied all of the test cases.

Practice creating test cases: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. List some test cases for a function that carries out division of two parameters. Would you use values near zero for the denominator?

   division by the same number
   division by a big number
   division by a small number
   15/5
   15/0
   0/15
   0/0

2. List some test cases for a price comparison program. This program reads a bar code, accepts a price as input, and consults GPS. It locates stores within a parameter radius that are selling the specified product for less than the product is being sold in the current location.

   lowest price available
   all prices same
   r = 100
   r = 0

3. Recall that a function has three manifestations: invocation, prototype and implementation. This is also true for functions in test code, i.e., functions

whose job is to test other functions. Where do the three manifestations of test functions go, in the project's collection of files?

prototypes go in test.h
innvocation goes in main.c
implementation goes in test.c