Unit Testing for the Application Control Language (ACL) Software

Christina Heinich
Kennedy Space Center
Major: Computer Science
KSC-FO
31-03-2014

Unit Testing for the Application Control Language (ACL) Software

Christina M. Heinich[1]
Texas State University, San Marcos, Texas, 78666

In the software development process, code needs to be tested before it can be packaged for release in order to make sure the program actually does what it says is supposed to happen as well as to check how the program deals with errors and edge cases (such as negative or very large numbers). One of the major parts of the testing process is unit testing, where you test specific units of the code to make sure each individual part of the code works. This project is about unit testing many different components of the ACL software and fixing any errors encountered. To do this, mocks of other objects need to be created and every line of code needs to be exercised to make sure every case is accounted for. Mocks are important to make because it gives direct control of the environment the unit lives in instead of attempting to work with the entire program. This makes it easier to achieve the second goal of exercising every line of code.

I. Introduction

Designing software is complicated. You have to understand what your users expect from you, and you have to know the best way to give the user what they expect. In addition, you need to understand how to make the process as safe as possible for the user and eliminate all possible errors.

The Application Control Language (ACL) project is no different. The goal of the project is to provide engineers with a simplified programming language so they can create scripts for their real world components without having to learn the intricacies of real programming languages and computer resource management. The Application Services and Framework (ASF) team also creates an environment for engineers to create and test their ACL scripts without bringing in the real world components.

There are three parts to this project: the ACL language itself, a development environment (IDE) that coders can use to run and test their code, and the framework defining the processes that turns the ACL code into C++ code that a computer can work with. This is no small feat, and before users can even touch the environment it needs to be thoroughly tested for bugs and inefficiencies. This is where I come in.

My task for this internship was to test various components of the ASF framework in order to meet the March 21 deadline and make sure all portions of the code were covered and working as expected. Unexpected input should be handled, and there should be no errors appearing when they should not. There were many challenges in doing this. First, I had to learn how to use the testing environment and tools used by ASF. Second, I had to understand what the code itself was doing, and know how to deal with the bugs I found.

[1] Software Engineering Intern, NE-C3, Kennedy Space Center, Texas State University-San Marcos.

II. Understanding the Tools

There are many different ways to test software, but the method used by ASF is primarily unit testing. The goal of unit testing is to individually test every unit to make sure that it is safe and can handle any input. The definition of a unit is very loose and can mean anything from a few lines to an entire file; it all depends on the project. In order to test the code, I had to use three different tools: the Linux terminal, CxxTest, and gcov. By using these tools, as well as the processes laid down by the ASF team, I was not only able to create and run tests for the ACL, but see my progress and figure out what cases I still needed to cover.

A. Linux Terminal

All of our software development is done using the Linux operating system. Linux places a large emphasis on using the terminal. The terminal is a quick and powerful tool to navigate around the directories in a computer and execute a program. For example, if I wanted to run a program called hello, I would first write the program in a programming language (let's say C++ since that's the language used for ACL) and save in a .cpp file. Let's call it hello.cpp. To run hello.cpp, I first compile it[2] using this command:

```
g++ -o hello hello.cpp
```

This command uses the g++ compiler to convert hello.cpp first into assembly language (which is much closer to the kind of instructions a computer can understand), and finally into machine language (which is binary ones and zeros). We save this into the executable hello.o, which we can now run with the command:

```
./hello
```

This system works fine for simple projects, but most projects (like this one) have many files, many of which depend on other files. In order to handle this, we put all these commands in to a makefile, which is a handy tool that not only allows us to build multiple files at once, but also keeps us from wasting time recompiling files that haven't been changed. In addition, you can use extra features to have more power over your compile. For example, gmake is a basic compile that creates all the required executables, but gmake clean will remove any previously created executables, just in case you suspect one is corrupted for whatever reason. In regards to unit testing, we have two commands we use:

```
gmake unittest CLASS=<CLASSNAME>
```

and

```
gmake unittestcoverage CLASS=<CLASSNAME>
```

The command gmake unittest will compile and run the unit test we wrote for a particular class[3] (specified by CLASSNAME) using the Cxx tool (which will be explained next), which will tell us how many tests we failed. The command gmake unittestcoverage allows us to actually see how many lines of code within a class we are covering in our tests. The command also executes the gmake unittest command.

2 Compiling is process to turn a computer language like C++ or Java into the binary ones and zeros that a computer can actually understand.

3 The word "class" is going to be used a lot in this paper. Most modern day programming uses the principles of object oriented programming. The goal of this style of programming is to create objects that have both values (such as color or density) and ways with which to interact with the object (such as pick up the object, or change its color). A class is basically the blueprint for an object. You can create an instance of a class by creating an object, initialize its values, and define the ways to interact with the object by creating methods (functions associated created in a class).

Before this internship I had used the terminal before and knew, if not anything advanced, at least the bare minimum I needed to get by, but during this project my terminal skills were definitely stretched to the limit.

B. CxxTest

The tool we used to create the tests was a program called CxxTest. CxxTest works off of test suites. A test suite skeleton is generated for a given class. In the test suite you can declare new classes, global variables, functions, as well as tests. Tests are implemented as functions. If the test suite was automatically generated, a test function will be created for each public method of a class with a name like test_methodName. The test suite also has two functions called setup and teardown. The setup function is always called at the beginning of a test function, while teardown is called at the end. This functionality is useful for when you have initialization and deinitialization procedures (such as setting up an object or destroying it) that you need to do every time.

When writing a unit test using CxxTest, I first create an object of that class. Then I decide what process I wish to test. Say the process in question is opening a file, and there is a method called openFile(FILENAME). First I might see what happens when I give openFile all good data. I would expect openFile to successfully open a file without any errors, but if that is not the case, I know that there is a bug in the code.[4] Next, I might test what happens when I give the method bad information, such as a nonexistent file name or a filepath without the proper permissions. The expected outcome varies. Depending on what the documentation of the code says, I might expect openFile to create a file if it doesn't exist. If openFile is unable to create/open the specified file, I might expect it to throw an error, or I might expect it to print a warning and do nothing else.

CxxTest comes with a set of TS_ASSERT functions I can use to test methods. The TS_ASSERT functions tests the truth value of the parameters passed to it. A basic TS_ASSERT function takes in one parameter, and if the parameter is true, CxxTest marks a success. If the parameter is false, TS_ASSERT will mark the test as a failure, display a message, and tell you the input it expected as opposed to what it got. In addition to TS_ASSERT, there are other functions like TS_ASSERT_EQUALS or TS_ASSERT_GREATER_THAN (which take two parameters and compare them for a truth value) or TS_FAIL, which always fails the test.

Using the TS_ASSERT functions, I was able to find bugs in the system. Whatever bugs I do find, I need to report to my mentors. Usually they were the ones who went to fix the bugs, but occasionally I would fix the bugs myself, which I will get into later.

C. Gcov

Often the goal of unit testing is to cover every single line of code for 100% coverage. The tool we used to figure out if we were getting that coverage is Gcov. Gcov generates a PDF that shows you how many times you have hit a certain line of code, as well as clearly mark the lines of code that have not been touched. It's a great way to make sure you are covering every case.

Now very often it is impossible to get 100% coverage for many reasons. For example, sometimes the code is using commercial software. While it is perfectly reasonable to make sure in code that the software is working properly, it is sometimes difficult to impossible to simulate a commercial software failure. In addition, sometimes certain sections of code are simply unreachable, such as when there are redundant error checks or bad defunct code whose use case does not exist anymore. Most of my classes I was able to cover completely, but a few of them I had to leave them after a certain point because I could not squeeze out any more coverage.

III. Testing and Error Fixing

Time was of the essence for the unit testing project, so I ended up testing multiple classes of code as opposed to just one. I worked on six classes: AsfTime, CommandVerifier, RuleThread, AsfClientApi, AsfCmdListener, and

4 Or, at minimum, there was a poor design choice. Perhaps the function openFile doesn't open a file at all; instead it is designed to print out a message or some other process that is not opening a file. While this is not exactly a bug, it is still an issue that needs to be corrected.

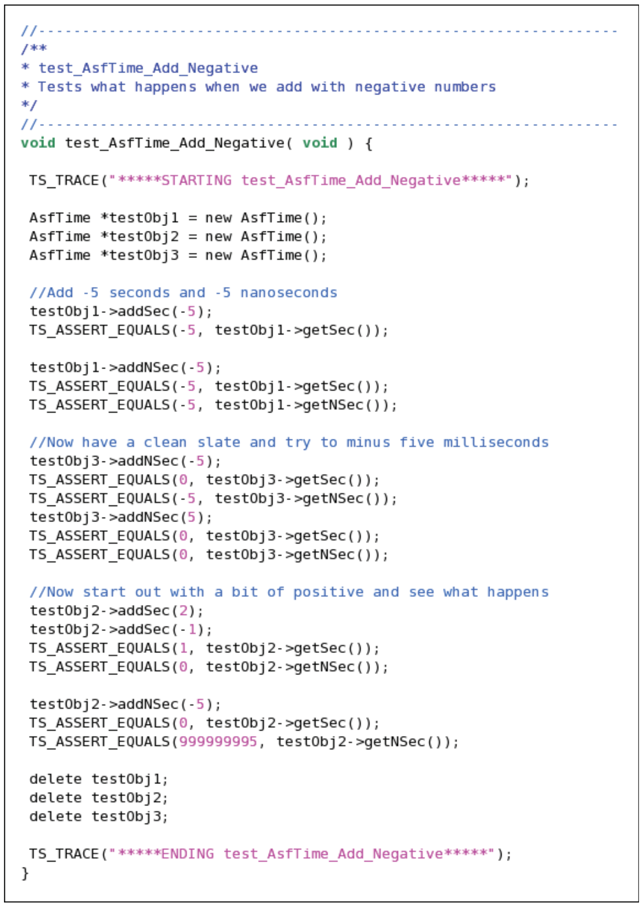
finally AsfCapeRouter. There were challenges and interesting factors in all of the tests, but for want of space I shall go into only three: AsfTime, AsfClientApi, and AsfCapeRouter.

A. AsfTime

AsfTime was the first class I tackled. AsfTime stores nanoseconds and seconds. The class defines basic operations such as addition and subtraction of seconds and nanoseconds. For example, if you add enough nanoseconds, the excess nanoseconds will be converted into seconds. It seemed like fairly straightforward code to test, but I ran into a major problem: negative time.

The problem comes from crossing the boundary between positive and negative. Coordinating the nanoseconds with seconds when going into negative was difficult as originally, nanoseconds could never be negative, but seconds could.

Fixing the issue first required a discussion about whether or not we even needed negative time. After coordinating with a user, we came to the conclusion that AsfTime needed to account for the case of negative time, and thus the issue needed to be fixed. In order to fix this issue, I had to write out tables on paper trying to account for every case that could possibly happen in the addition and subtraction of the seconds/nanoseconds. (What if seconds is zero, nanoseconds is positive, and we subtact a full second? What if seconds is positive, nanoseconds is positive, and we subtract a number greater than seconds + nanoseconds? All were cases that needed to be accounted for.) Once I did that, I was able to create a sequence of use cases that would account for all situations, and once the aforementioned user helped with streamlining the code to make more readable, the fixes were ready to join the rest of the project.



```
//---------------------------------------------------------------
/**
* test_AsfTime_Add_Negative
* Tests what happens when we add with negative numbers
*/
//---------------------------------------------------------------
void test_AsfTime_Add_Negative( void ) {

 TS_TRACE("*****STARTING test_AsfTime_Add_Negative*****");

 AsfTime *testObj1 = new AsfTime();
 AsfTime *testObj2 = new AsfTime();
 AsfTime *testObj3 = new AsfTime();

 //Add -5 seconds and -5 nanoseconds
 testObj1->addSec(-5);
 TS_ASSERT_EQUALS(-5, testObj1->getSec());

 testObj1->addNSec(-5);
 TS_ASSERT_EQUALS(-5, testObj1->getSec());
 TS_ASSERT_EQUALS(-5, testObj1->getNSec());

 //Now have a clean slate and try to minus five milliseconds
 testObj3->addNSec(-5);
 TS_ASSERT_EQUALS(0, testObj3->getSec());
 TS_ASSERT_EQUALS(-5, testObj3->getNSec());
 testObj3->addNSec(5);
 TS_ASSERT_EQUALS(0, testObj3->getSec());
 TS_ASSERT_EQUALS(0, testObj3->getNSec());

 //Now start out with a bit of positive and see what happens
 testObj2->addSec(2);
 testObj2->addSec(-1);
 TS_ASSERT_EQUALS(1, testObj2->getSec());
 TS_ASSERT_EQUALS(0, testObj2->getNSec());

 testObj2->addNSec(-5);
 TS_ASSERT_EQUALS(0, testObj2->getSec());
 TS_ASSERT_EQUALS(999999995, testObj2->getNSec());

 delete testObj1;
 delete testObj2;
 delete testObj3;

 TS_TRACE("*****ENDING test_AsfTime_Add_Negative*****");
}
```

Figure 1. Small test for AsfTime. This is a small test for the AsfTime class. Here I am testing the addSec and addNSec methods when they are given negative time

B. AsfClientApi

AsfClientApi was often called the behemoth in meetings, and rightly so. AsfClientApi was approximately 17,000 lines total, with around 6,000 lines of code to actually cover.[5] Because of this, multiple people were always working on ClientApi. When I joined the testing, there were nine other people working on different sections of the code. My job was to cover the File I/O section, or the section responsible for opening a file, writing to it, and closing it. No errors were found, although I did have difficulty with something called system calls. A system call is a tool in C/C++ that allows you to send a command directly to the terminal. This gives your program enormous amounts of power, which is why a lot of other computer languages do not include this tool. Using system calls, you can create and delete files, or directories. With the right command, you

5 Many times comments and brackets fill up lines so that there are many lines in a file, but actual number of executable lines of code is generally a lot less.

could even delete your entire home directory[6], hence the dangerous part. I had never done system calls before, so getting the hang of them was a challenge. I did succeed in not doing any severe damage to my computer, so I would call that a success.

The most difficult part of ClientApi was coordination. Because it was so large, it had many people working on it. It is impossible to have every person working on the same test suite; every time somebody saved they would override the work of somebody else. Therefore, we had one person who had the master file, and whenever one of us finished a test we sent it to him. He would then add it to the master file. This way we avoided any massive merge issues.

C. AsfCapeRouter

Testing AsfCapeRouter was the most difficult task I had. The CapeRouter's job is to receive a command to start a script or a rule and fire off said script or rule. Figuring out how to create those commands was hard enough, but the most difficult part had to do with threads, which requires a little explanation.

In all computers, we have something called a processor. The processor is what executes the commands we write in our programs. Inside there is a clock, which “ticks” at a certain rate (when you are buying a computer and you see it had a 1.10 GHz processor, that means the clock ticks 1,100,000,000 times a second). Every time the clock ticks, an instruction is executed. With a single core processor, we are limited to one instruction per clock cycle, but most modern day processors are multicore, meaning they can execute more than one instruction per clock cycle. The problem is in order to utilize those extra cores we need to write our code using parallel programming standards. One popular method is to use threads, which can be created from a main program to run next to the parent thread. The problem with threads is synchronization. When there are shared resources between threads, there is a very real possibility that two threads could access the data at the same time, causing a data race.

CapeRouter in particular had issues with segmentation faults. Due to the way that CapeRouter was set up, it was completely possible for you to stop CapeRouter while the scripts (which were implemented using threads) where still running. A thread will keep running until its job is finished, and then it returns back to the main program. So a thread could still be executing, finish up, and try to return to the main program, only to find that nobody is there. This causes a segmentation fault.

Testing CapeRouter soon became about finding out just how much time I needed to wait to allow for those errors to not happen. There was simply no other way to test the code if I didn't put waits after and inside each test. What was even more challenging was that because it was a timing issue, and the amount of time a program takes to execute is not constant, sometimes errors would appear without warning and disappear just as quickly. In the end, the test took about five minutes to run every time, and it would pass all the tests about 90% of the time.

These faults would only appear in the case of rapid creation and destruction of thread a common occurrence in a testing situation, but very rare in a real world situation; it would be extremely difficult for a human to manually add scripts at the same speed of a processor. Nonetheless, these errors were decided to be a known defect in the CapeRouter that, due to time constraints, would have to be fixed in the next iteration.

IV. Conclusion

Working with the ASF team, while not in the original plan, was definitely rewarding. I learned some new tools, and was able to improve my threading skills as well as my programming skills in general. I look forward to returning in the summer to continue working with the team.

Acknowledgments

The author would like to thank her new mentors, Linda Crawford, Randy Lane, and Caylyne Shelton, for always being helpful, patient, and great to work with, as well as her old mentor, David Slaiman, who had to deal with the loss of two of his interns in the name of unit testing.

6 In case you're interested, I haven't tested this, but the command is something like rmdir --ignore-fail-on-non-empty -p /directory/directory/etc. Code safe.

References

[1]Siever, E., Friggins, S., and Weber, A., *Linux in a Nutshell: A Desktop Quick Reference*, 4th ed., O'Reilly, Sebastopol, CA, 2003.
[2]Gcov, GNU CC Suite, Ver. 4.1.2, Free Software Foundation, Inc, Boston, MA, 2006.
[3]CxxTest, Ver. 4.0.3, CxxTest Devopment Team, 2012.
[4] Redhat Enterprise Linux, Linux Operating System, Ver. 5.10, Red Hat, Inc, Raleigh, NC, 2006.