Using the componentTests

July 5, 2017

# Purpose

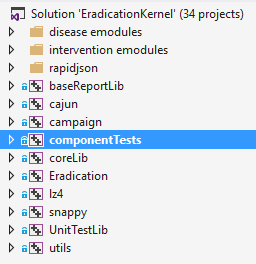
The purpose of the componentTests is to allow testing of specific classes or methods without running the entire DTK. Some people would call these unit tests but we have chosen broader name to encompass a set of tests that are a little larger than a “unit”.

As of DTK V2.13, there are tests for a variety of DTK features. There are tests for reading demographics files, testing the pair forming agent (PFA) logic to see that it generates the distribution indicated by the input parameters, and numerous tests checking different error conditions. Combined with the regression tests and science feature tests, we can achieve excellent test coverage.

# Building

## Visual Studio

There is a project within the EradicationKernel.sln for building the componentTests.



If you build the solution, it should build this project by default.

## SCONS

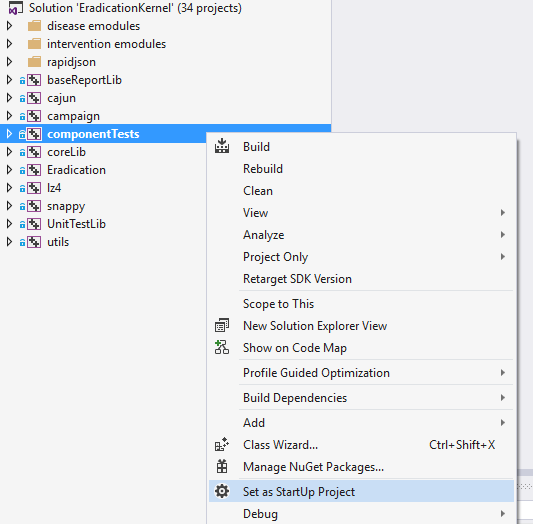
When building the DTK via SCONS, the componentTests executable will be built automatically. It is built on both Windows and Linux.

One should note that as of v2.13 not all of the tests are built for Linux. The SerializationTest and SchemaTest are omitted when building Linux. Serialization is not supported for Linux and more work is required to get the SchemaTest passing for Linux

# Running

## Visual Studio

To run the tests within Visual Studio, you must first make the componentTests project the Start Up project. Right-click on the project and select “Set As StartUp Project”. The project name should turn bold.

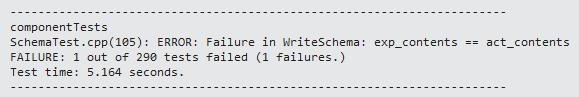


Once it is the startup project, you can just press the “Local Windows Debugger” button or press the F5 key.

When the tests are done running, the results will be in Visual Studio’s Output Window. If the tests pass, you should see output like:



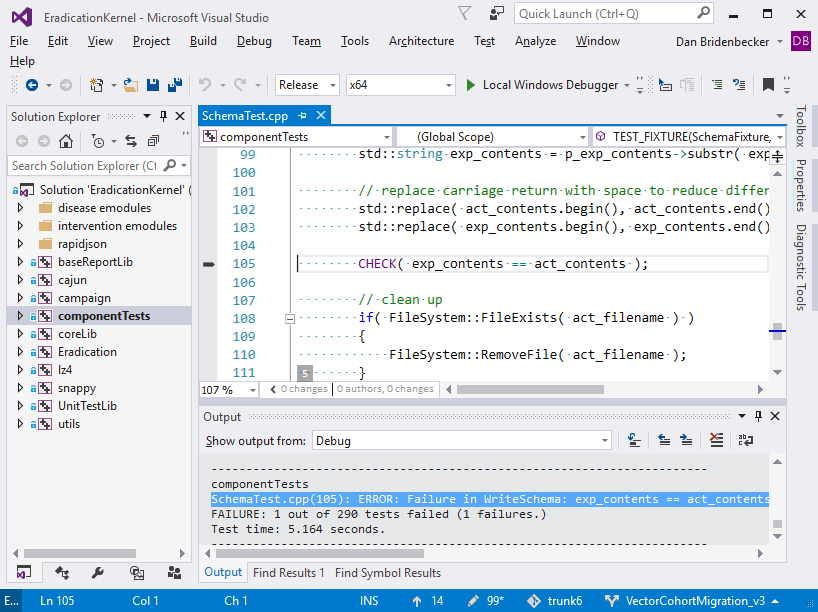
If a test fails, you should see output like:



From this output, we see that one out of 290 tests failed and that the failing test was in SchemaTest.cpp. If we double-click on the failing test line:



Visual Studio will open this file and jump to the line indicated



From here, you should be able to start figuring out why the test failed.

## Command Line

The componentTests assume that they are run from within the componentTests directory. That is, they assume that the testdata directory is in the executables current working directory. For example, if you checked out the source code into a directory called C:\DtkTrunk, then you should run the tests in the C:\DtkTrunk\componentTests directory like the following:

C:\DtkTrunk\componentTests> x64\Release\componentTests.exe

## regression\_test.py

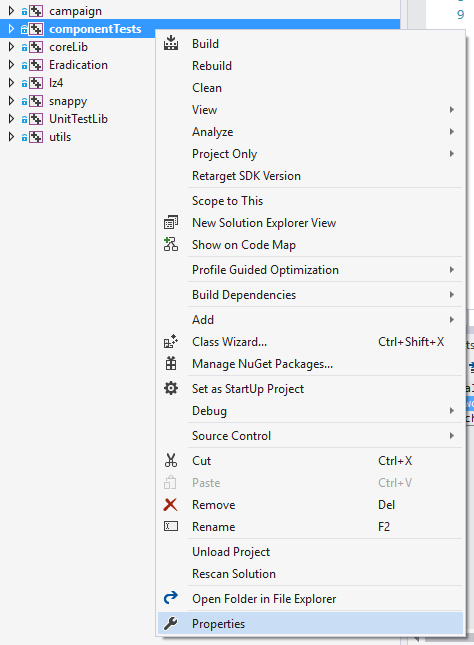
The tests can also be run using regression\_test.py. To do this, you simply add the --component-tests flag to the command line. This will run the component tests after running the regression tests. If you want to see more than the summary output of the tests, you must also include the flag --component-tests-show-output.

When the --component-tests flag is present, regression\_test.py will look for the componentTests executable depending on --scons and --debug flags. If neither of these flags are present, then it will look in the <DtkTrunk>\componentTests\x64\Release directory. If it finds the executable there, it will run the tests. If the executable is not found, it will not run the tests.

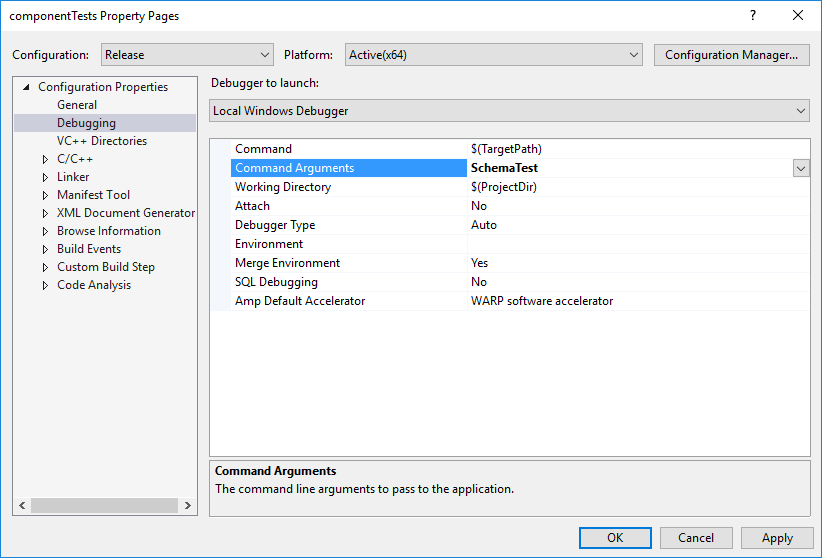
## Running a Specific Test Suite

Sometimes you don’t want to run all of the tests. You just want to run a specific suite that you are working on or where tests are failing. To do this, you specify the name of the suite on the command line.

In Visual Studio, right-click on the componentTests project and select Properties.



In the project Property Pages dialog, select the Debugging option and then add the name of the test suite to the Command Arguments edit box.



On the command line, you just need to add the name of the test suite. For example,

X64\Release\componentTests SchemaTest

In either case, if you do not include the name of a specific test suite, it will run all of the tests.

### One Test of a Specific Suite

Currently, there is not a good way of running just one test. One way to achieve this is using the compiler macro #if. You can use this to exclude all of the other tests in the suite so only the one you are working on is being compiled. Between this and including the test suite name on the command line, you can run just one test.

# Writing

The componentTests make use of the open source test framework called UnitTest++. We have made some changes to this source to improve its use in the DTK development environment. To find out more on UnitTest++, see: <https://github.com/unittest-cpp/unittest-cpp/wiki>

## Adding Tests

When adding a new suite of tests, you can copy one of the existing test files and modify it for your needs. Typically, the name of the file is the name of class plus the suffix “Test”. For example, NodeDemographicsTest is a test suite that tests the NodeDemographics class.

## Simple Tests

There are two sorts of test suites to consider when trying to figure out which one to copy. If you have a fairly simple class to test and it doesn’t require much setup to run the code, then you might want to copy FileSystemTest.cpp. It uses the UnitTest++ macro “TEST()” for each test where each test is run as is.

## Complex Tests

If you have a more complex class to test or it requires some setup of other DTK objects, then you should consider copying LoadBalanceSchemeTest.cpp. This test uses the UnitTest++ macro “TEST\_FIXTURE()” for each test. This macro adds the unit test concepts of “setup” and “tear down” by constructing a “Fixture” at the beginning of each test and destructing “Fixture” at the end of each test. This is useful so that you can have the environment and other parameters configured the same for each test. It is also handy, because the destructor of “Fixture” is called regardless of whether the tests pass or not. It gives the person writing the tests a chance to make sure everything is cleaned up so other tests are not affected.

## Data for Your Tests - testdata

There is a “testdata” directory where the test developer can have data files associated with their test. If you are going to have just one file, it is best to name the file something like <TestSuite>.<TestName>.<extension>. This makes it easy to know what files go with what tests. If you need multiple files, then it is best to create a subdirectory in “testdata” for your files. Again, it is best if the directory has the same name as the test suite.

## Mock/Fake Objects

One should note that as of DTK V2.13, there are several mock or fake classes. For example, there is a RandomFake.h that the test writer can use to ensure that the code under test gets a specific random number. As another example, there is INodeContextFake.h. If you have a class/method that is modifying some aspect of a INodeContext, you can let your class operate on this fake object. This is handy because you can do things in the fake object to support testing that you can’t easily do in the normal code.

# Information About Specific Tests

## SchemaTest – WriteSchema

The SchemaTest-WriteSchema test is a simple test that generates the schema and compare is it to a pre-approved version of the schema, much like the regression tests.

One of the benefits of this test is that it forces the developer to look at the schema when things change. It doesn’t take a lot of time to do this, but it gives the developer the opportunity to consider:

* Did the schema change as expected?
* Is the parameter defined appropriately?
* Does the parameter have the right “depends-on” option?

When this test fails, the following file is generated:

<DtkTrunk>\componentTests\testdata\SchemaTest.WriteSchema-actual.json

This is the schema generated from your latest code. If you compare this file with:

<DtkTrunk>\componentTests\testdata\SchemaTest.WriteSchema.json

You can quickly see what has changed. If all of the changes look correct, then you can replace the existing SchemaTest.WriteSchema.json with the one you just generated.

## NodeDemographicsTest – TestReadLargeFile

The passing of this test is somewhat dependent on the computer on which the tests are being run. It is a timing test to see how long it takes to read a large demographics input file. If this test fails, you should first consider the computer conditions. For example, if the computer’s processor if completely maxed out, then this test could fail because the operating system is doing a lot of task switching. If you are working on the demographics code and this test fails, then you might want to look more closely at this test.