TSUnit - Unittests easy for C/C++ Code

Author: Hans-Peter Bestaendig, Kuehbachstr. 8, 81543 Munich, Germany
eMail: hdusel@tangerine-soft.de
WWW: http://www.tangerine-soft.de

Document Version 1.0 - 2022–05–06

Table Of Contents

- What is TSUnit?
- Author
    1. Prerequisites:
- What makes TSUnit so great?
    1. It enables you to work according the “FIRST Principles”
- How do I use TSUnit?
    1. The Anatomy of a Unittest
        - What is a Simple test ?
        - What is a Test fixture?
    2. The run and final Reporting
- What Assertion does TSUnit support?
- Hmm, this looks pretty good! May you show me an example?
    1. Further Examples
- The TSUnit license
- Common Questions & Answers
    1. How do I control the order of my tests.
    2. My test Fixture does not execute the StartUp() method!
    3. My test Fixture does not execute the SetUp() method!

What is TSUnit?

TSUnit is a Unittest framework written in C++ with the aim of checking C and/or C++ code in the context of unit tests.

Author

The prefix *TS* stands for “Tangerine-Soft” and this my Software Label “tangerine-soft”.

My name is Peter Beständig, I’m living in Munich “Untergiasing”. You’ll reach me via hdusel@tangerine-soft.-de. So drop me a mail! I’m looking forward to hear from you!

Prerequisites:

TSUnit requires a C++ compiler capable of generating C++11 code.

However this does not mean that jour *Code Under Test* (*CUT*) needs to be written in C++! So if your code written in Plain-C then TSUnit is even able to handle it.

What makes TSUnit so great?

- TSUnit is very *lightweight*: Its “runtime” just consists of a `.cpp` and a `.hpp` file.
- TSUnit is *flexible*: Due to its small size, you can easily adapt it to your own environment.
- It allows the execution of unit tests in “stand alone mode” or in “embedded mode” (see below)
- Thanks to the LGPL V3 license, TSUnit may also be used freely in commercial products.
- And last not least using TSUnit is fun! In accordance with the FIRST rules of test-driven development, its ease of use does not stand in the way of you as a developer doing your actual work.

It enables you to work according the “FIRST Principles”

I developed TSUnit to allow a developer to work with the focus on the “FIRST Principles”.
So I think it’s a good time to recap the “FIRST Principles”
(copied from https://www.appsdeveloperblog.com/the-first-principle-in-unit-testing/):

- **F**ast
  The first letter in the F.I.R.S.T principle stands for – Fast. Unit tests are small pieces of code that perform one, specific task. Because unit tests are small, and unlike integration tests, unit tests do not communicate over the network and do not perform database operations. Because unit tests do not communicate with remote servers or databases, they run very fast. Developers who practice the test-driven development, run unit tests very often as they implement app features.
- **I**ndependent
  The next letter in the F.I.R.S.T principle stands for – Independent. Unit tests must be independent of each other. One unit test should not depend on the result produced by another unit test. In fact, most of the time, unit tests are run in a randomized order. The code you are testing or the system under test should also be isolated from its dependencies. To make sure that the bug in a dependency does not influence a unit test, the dependencies are usually mocked or stubbed with predefined data. This way, a unit test, can test a system under test in isolation from its dependencies and produce an accurate result.

- **R**epeatable
  The next letter in the F.I.R.S.T principle stands for – Repeatable. A unit test should be repeatable and if run multiple times should produce the same result. If a unit test is run on a different computer it should also produce the same result. This is why unit tests are made independent of the environment and from other unit tests.
  The input parameters that the function under test requires are usually predefined and hardcoded. And if function under the test needs to be tested with valid and invalid input parameters, then two or three different unit tests are created. Each unit test tests the function under the tests with its own set of predefined parameters. This way, the unit test becomes repeatable and can run multiple times in different environments and will still produce the same result each time it runs.
- **S**elf-validating
  The next letter stands for Self-validating. This means that to learn if a unit test has passed or not, the developer should not do any additional manual checks after the test completes.
  The unit test will validate the result that the function under the test has produced and itself will make a decision whether it passes or fails. After the test completes, the result will be clear.
- **T**horough
  The next letter stands for thorough and developers who practice test-driven development also spell it as timely. Talking about thoroughness. This means that when testing a function, we should consider a happy path as well as a negative scenario. And thus most of the time, we create multiple unit tests to test a function that accepts input parameters. One unit test will test a function under test with a valid input parameter. And another unit test will test the function under the test with invalid input parameters.
  If there is a range like MIN and MAX value, then we should create additional unit tests to test for minimum and maximum values as well.
  Unit tests should also be timely. It is better to create a unit test at the time you are working on the app feature. This way you will have more confidence that the feature does work as expected and fewer chances, that you will introduce a bug that will be then released to production. So before you promote the code to production it should be covered with unit tests.

How do I use TSUnit?

As stated above using TSUnit is quite simple. You actually just need these three things:

- A "*Code under Test*" (called *CUT* up from now). This is the actual part of the *production code* you want to *verify with a Unittest*
- A "*Test Harness*" that is represented by this Package- TSUnit
- A C++ Compiler that is capable to compile *C++11 compliant* code. Don't be confused! Albeit TSUnit itself needs a C++ Compiler your *CUT* may be written in C because TSUnit itself does not demand that your *CUT* is written in C++(11)!

So in fact TSUnit is a *Test Execution environment* that has the responsibility to:

- Collect all of your Testcases (and it does this *magically* easy ;-)). The really **cute thing** is that if you write your Testcases you **don't need** to explicitly *register* them on the Test Executor! But we'll see this in a minute… promised!
- Execute these Testcases and report the results.
- Perform a *consolidated Report* after all test has been run.

The Anatomy of a Unittest

1. A Unittest that has to be a C++ File with an arbitrary name. Since it uses TSUnit which is written in C++ it has to be a C++ File either meaning its file Suffix is `.cpp`
2. This Unittests file has to include `TSUnit.hpp`
3. This Unittests may now execute either **"Simple tests"** or a **"Test Fixture"**:
4. An include of "TSUnit.hpp" - hence the Test file itself needs to be a C++ File

What is a Simple test ?

A *simple Test case* is supposed to group the tests that belong to a certain function of the “Code under test”. A simple Test case consist of a *group name* and a *distinct_name_of_the_test*. This tuple has to be unique all over the complete suite of tests.

In TSUnit a *“Simple Test”* has to be introduced by the (TSUnit) macro

TSUNIT_TEST(*<GROUPNAME>, <NAME_OF_TEST_WITHIN_THIS_GROUP>*)

Note that TSUnit will form a symbolic name from these two parameters!

This means that you are restricted to the *common symbolic chars* that are allowed for C-function names, meaning that you **cannot** use german umlauts or special character such as “@#+-/.,<>*%&()§"!” except the underscore (_) character!

Example

The Example below shows 4 *Simple Tests* that checks if a (fictive) Arithmetic Logic Unit (the Code Under Test (*CUT*)) works correctly for its add and sub function.

Every Test is introduced by the TSUnit Token ‘TSUNIT_TEST’ which characterizes a “*simple testcase*”:

```
#include "TSUnit.hpp"
...
TSUNIT_TEST(AluAdderTest, checkIfPositveNumbersAreAddedCorrectly)
{
    UT_EXPECT_EQ(alu_add(+3, +4), +3 + +4);
}

TSUNIT_TEST(AluAdderTest, checkIfNegativeNumbersAreAddedCorrectly)
{
    UT_EXPECT_EQ(alu_add(+3, -4), +3 + -4);
    UT_EXPECT_EQ(alu_add(-3, +4), -3 + +4);
    UT_EXPECT_EQ(alu_add(-3, -4), -3 + -4);
}

TSUNIT_TEST(AluSubtractTest, checkIfPositveNumbersAreSubtractedCorrectly)
{
    UT_EXPECT_EQ(alu_sub(+3, +4), 3 - 4);
}

TSUNIT_TEST(AluSubtractTest, checkIfNegativeNumbersAreSubtractedCorrectly)
{
    UT_EXPECT_EQ(alu_sub(+3, -4), +3 - -4);
    UT_EXPECT_EQ(alu_sub(-3, +4), -3 - +4);
    UT_EXPECT_EQ(alu_sub(-3, -4), -3 - -4);
}
```

What is a Test fixture?

A Test fixture is a more elaborate testcase which consists of a Custom defined Helper class. To be more specific - the test fixture runs within an instance of this Test class. In addition the helper class of such a test fixture may contain two distinct methods.
One is usually named `SetUp()` and this called before your actual Test starts and another one named `TearDown()` that will be called after the testcase has been *finished*.

In TSUnit a “Test Fixture” has to be introduced by the (TSUnit) Macro

TSUNIT_TESTF(*<NAME_OF_THE_FIXTURE_CLASS>*, *<NAME_OF_TEST_WITHIN_THIS_GROUP>*)

Example

```
#include "TSUnit.hpp"
...
/*
 * Define a test Fixture class that will be used by all Test Fixtures that refer to it.
 * Note that a Test FixtureClass has to be derived from tsunit::Test!
 *
 * If a test Fixture referring this class is executed then the implicitly call order is as fol
 *
 * - First the test executor temporarily creates an instance of this class
 * - Due to the construction the constructor of this class will be called
 * - Then the method `SetUp()` (if implemented) will be called by the test executor.
 * - Then all the code mentioned within the actual TSUNIT_TESTF() call will be executed.
 * - Then the method `TearDown()` (if implemented) will be called by the test executor.
 * - Finally the class destructor will be called hence the FixtureTest object is destroyed
 *   at the end of the test case by the test executor.
 */
class MyFixtureTest : public tsunit::Test
{
public:
    MyFixtureTest() = default;
    ~MyFixtureTest() = default;

    void SetUp() {
        CodeUnderTest_init(); // Assume this call is one of your 'CUT' part
    }

    void TearDown() {
        CodeUnderTest_deinit(); // Assume this call is one of your 'CUT' part
    }

    /*
     * For example add a custom method supposed to "ease" the repetitive usage
     * of the CUT. Note that this is just to show that the Fixture runs within
     * the scope of an object of this class!
     */
    void configMode(int inMode)
    {
        CodeUnderTest_configureMode(inMode); // Assume this call is one of your 'CUT' part
    }
};

TSUNIT_TESTF(MyFixtureTest, checkIfTheModuleIsRunningAfterInitialzation)
{
    /* Note that this code now runs within the context of an temporary object
     * of the class MyFixtureTest!
     *
     * Furthermore note that already the ctor of MyFixtureTest and
     * MyFixtureTest::SetUp() (in this order) has already executed
     * when we reach here!
     */

    configMode(CodeUnderTest_MODE_START); //

    // Now assume that the Module is running
    UT_EXPECT_EQ(CodeUnderTest_isRunning(), true);
```

```
    /*
     * Now the test ends.
     * The Test Executor (implicitely) runs the methos MyFixtureTest::TearDown()
     * and the dtor of MyFixtureTest in this order.
     */
}
```

This means the test fixture MyFixtureTest, checkIfAllMemebersAreZeroAfterInitialization effectively runs the command as follows (in this order):

1. `MyFixtureTest::MyFixtureTest();`
2. `MyFixtureTest::SetUp();`
    - `CodeUnderTest_init();`
3. All the code of `TSUNIT_TESTF(MyFixtureTest, checkIfTheModuleIsRunningAfterInitialzation)`
4. `configMode(CodeUnderTest_MODE_START);`
5. `MyFixtureTest::TearDown()` with
    - `CodeUnderTest_deinit();`
6. `MyFixtureTest::~MyFixtureTest();`

The run and final Reporting

At the end you will see a report that may look as follows:

```
================================================================================
Report of TSUnit V2.0
================================================================================
Running Basictests::checkIfInitiallyEmpty ............................ [PASSED]
Running Basictests::allocateOneAndCheckIfOccupied ..................... [PASSED]
Running Basictests::allocateAllAfterAnotherAndCheckIfOccupied ........ [FAILED]
\*\*\* Assertion failed in Basictests::allocateAllAfterAnotherAndCheckIfOccupied @line 78

Running Basictests::checkIfClearFreesAllMemory ....................... [PASSED]
Running Basictests::checkIfAllocationBeyondBoundariesWillreturnANullptr ... [PASSED]
Running Basictests::checkIfWeGetEightDifferentPointers ............... [PASSED]
Running Basictests::checkIfAllFAllocatedPointerWillBeRegainedUponFree ... [PASSED]
Running Basictests::checkIfAttemptingToFreeANullptrWillSucceeed ...... [PASSED]
Running Basictests::checkIfAttemptingToFreeMorePointersThanAllocatedWillFail ... [PASSED]
Running TestFixture1::checkIfAllocatingANewPointerWillNotAffectAlreadyAllocatedStores ...
[PASSED]
Running TestFixture1::checkIfFreeingAPointerWillNotAffectAlreadyAllocatedStores ... [PASSED]
Running Basictests::checkIfFreeingAnEntryMoreThanOnceBailsAnFailure ... [PASSED]
================================================================================
= Finished all Tests: Run 12 Tests, 11 passed, 1 failed.
= 105 assertions in total, 1 failed of these.
================================================================================
```

(Here CSI Sequences has been enabled (you may disable this either) which increase the readability of your output. So passed tests are marked in **green** color, failed are in **red**.)

What Assertion does TSUnit support?

Well TSUnit currently supports only 4 Kind of assertions:

- `UT_EXPECT_TRUE(arg)`:
  Checks if `arg` evaluates as **true**. If not then this check fails and will reported accordingly.
  For example `UT_EXPECT_TRUE(3==4)` will be reported as a **failed** test.
- `UT_EXPECT_FALSE(arg)`:
  Checks if `arg` evaluates as **false**. If not then this check fails and will reported accordingly.
  For example `UT_EXPECT_TRUE(3==3)` will be reported as a **failed** test.
- `UT_EXPECT_EQ(arg, expect)`:
  Means “Expect Equality of two arguments”. This Checks is `arg` *match exactly* with `expect`. If not then this check fails and will reported accordingly.
  For example `UT_EXPECT_EQ(3, 4)` will be reported as a **failed** test.
- `UT_EXPECT_NE(arg, expect)`:
  Means “Expect **NOT** Equality of two arguments”. This Checks is `arg` **does not** *match exactly* with `expect`. If not then this check fails and will reported accordingly.
  For example `UT_EXPECT_NE(3, 3)` will be reported as a **failed** test.

On the first glance these seems very low compare with other Test Frameworks out there: However from my personal perspective up to now these were pretty
sufficient in my daily work. Besides of this I think you may be able to extend them if you have special demands. You have the sources of TSUnit - so go for it! ;-)

Hmm, this looks pretty good! May you show me an example?

Sure! Honestly I was a bit worried that you don’t ask! ;-)
So let’s assume that we have some “Code under Test” (*CUT*). This is the actual software unit you want to test.

For this session let’s develop a small “Arithmetic Logic Unit” (*ALU*) that is capable to do some arithmetic operations.
So our specification says, that our *ALU* should support a function that is capable to add two signed integer numbers and return ist result.

So lets assume that our Code under Test consists of 2 Source files namely `alu.c` with an header file `alu.h` accordingly:

```
/* alu.h */
#ifndef __ALU_H__
#define __ALU_H__

signed int alu_add(signed int a, signed int b);
signed int alu_sub(signed int a, signed int b);

#endif /* __ALU_H__ */
```

…with the implementation (`alu.c`):

```
/* alu.c */
#include "alu.h"

signed int alu_add(signed int a, signed int b)
{
    return a + b;
}
```

```
signed int alu_sub(signed int a, signed int b)
{
    return a - b;
}
```

According this *CUT* lets assume that our test code is named “UT_Alu.cpp”

```
// UT_Alu.cpp
#include "TSUnit.hpp" // this is the header of our TSUnit Test
#include "alu.h" // This is the Header for the Code under Test
TSUNIT_TEST(ALU_BasicTests, checkIfAddWorks)
{
    UT_EXPECT_EQ(alu_add(3, 4), 3 + 4);
    UT_EXPECT_EQ(alu_sub(3, 4), 3 - 4);
}
```

So let’s compile and run the code in a terminal…

```
asra:coding_session hdusel$ g++ -std=c++11 TSUnit.cpp UT_Alu.cpp alu.c -o MyUnitTest && ./MyU
nitTest
================================================================================
Report of TSUnit V2.0
================================================================================
Running ALU_BasicTests::checkIfAddWorks ............................. [PASSED]
================================================================================
= Finished all Tests: Run 1 Tests, 1 passed, 0 failed.
= 2 assertions in total, 0 failed of these.
================================================================================
```

Further Examples

You’ll find further (more elaborate) examples within the subdirectory `examples/`.

One which illustrates the usage of TSUnit together with CMake
…and another one which illustrates the usage of TSUnit together with a simple make file.

The TSUnit license

TS Unit is subject to the “GNU Lesser General Public License LGPL V3”

Common Questions & Answers

How do I control the order of my tests.

Well according the “FIRST Principles” any tests has to be independent from another.
So the simple answer ist that you **cannot** determine the order of your tests. If a particular tests needs a specific order or has dependencies from other Modules that you may think to employ *Test Fixtures*.

My test Fixture does not execute the StartUp() method!

Yep, because you spelled it wrong. The expected method name has to be `SetUp()` and not `StartUp()`. ;-)

My test Fixture does not execute the SetUp() method!

I bet you wrote `TSUNIT_TEST` instead of `TSUNIT_TESTF` (with the leading **F** for **F**ixture)?