Aeryn

A C++ Testing Framework

User Guide

Paul Grenyer
paul (at) paulgrenyer.co.uk

What is Aeryn?

Aeryn is a C++ testing framework. Although it is primarily intended for unit testing, Aeryn is adaptable enough to handle integration testing and can be adapted for most other forms of C++ testing.

Aeryn is intended to be light weight with the minimal of code needed to create a test fixture. Unlike other testing frameworks Aeryn does not require all test fixtures to be inherited from a particular class. Test fixtures can be standalone functions or standalone classes.

Aeryn is adaptable via context objects that can be passed to test fixtures prior to running and through its call back reporting interface.

How do I get Aeryn?

SubVersion

By far the best way to get Aeryn is direct from its SubVersion [SVN] repository. This way you can get all the latest fixes easily. If you do get Aeryn from the repository it is also a good idea to join the commits mailing list by sending an email to commits-subscribe@aeryn.tigris.org . That way you will receive notification every time Aeryn is updated.

Every SVN repository has a URL which identifies it. The Aeryn trunk URL is:

> http://aeryn.tigris.org/svn/aeryn/trunk

You can also checkout stable revisions from:

> http://aeryn.tigris.org/svn/aeryn/tags

A user name and password is required to checkout the repository. If you have a Tigris [Tigris] username and password you can use that. If not the following can be used:

> Username: `guest`
> Password:

The password is blank. "`guest`" has read-only access and therefore does not require a password.

SubVersion Command Line Client

Download and install the SVN command line client for your operating system from the download area of SVN website. Open a command prompt and type the following to make sure the SVN command line client is setup correctly:

```
svn --version
```

the output should be something like this:

```
Subversion is open source software, see
http://subversion.tigris.org/
This product includes software developed by CollabNet
(http://www.Collab.Net/).

The following repository access (RA) modules are available:

...
```

Assuming the SVN command line client is correctly installed, change to the directory where you want to check the Aeryn repository out and type:

```
svn co http://aeryn.tigris.org/svn/aeryn/trunk aeryn --username
guest
```

This will generate the following prompt:

```
Authentication realm: <httpaeryn.tigris.org:80> CollabNet Subversion
Repository
Password for 'guest':
```

As explained above, "`guest`" has a blank password, so just press return. This will check out the Aeryn repository into a directory called Aeryn which is a subdirectory of the current directory. The output looks like this:

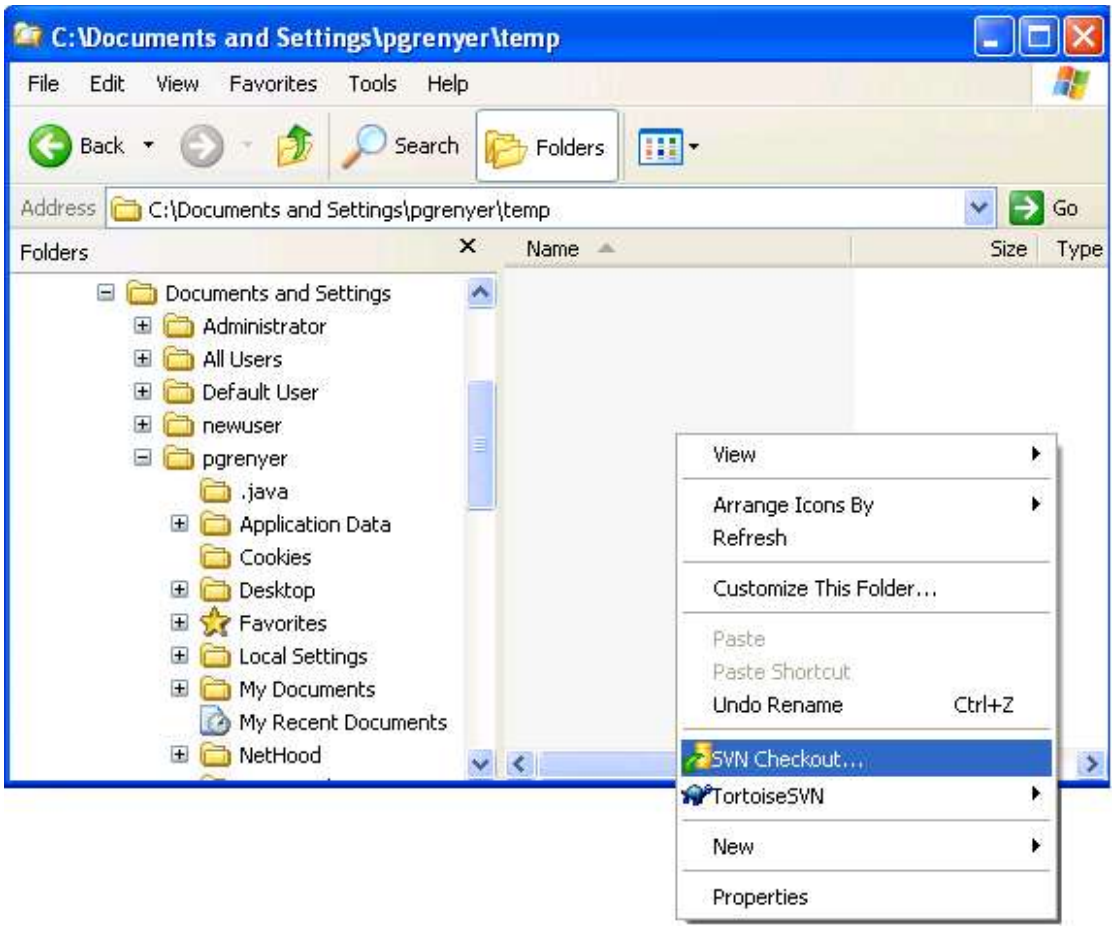
```
A    aeryn\trunk
A    aeryn\trunk\license.txt
A    aeryn\trunk\gpl_aeryn.txt
...
Checked out revision 71.
```

The revision number will be greater than 71.

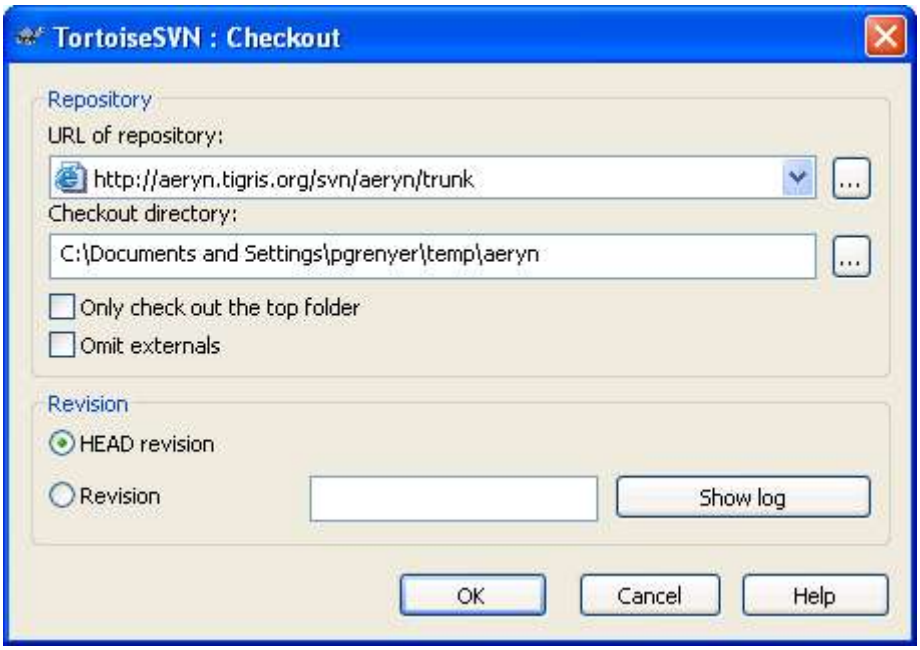
TortoiseSVN

Download and install TortoiseSVN [TortoiseSVN] from the download page of the TortoiseSVN website. TortoiseSVN is only available for Windows and requires a reboot after installation.

After the reboot, open Windows explore and go to the folder you'd like to check the Aeryn repository out to. Right click in inside the folder in the right-hand pain and select SVN Checkout as show below:



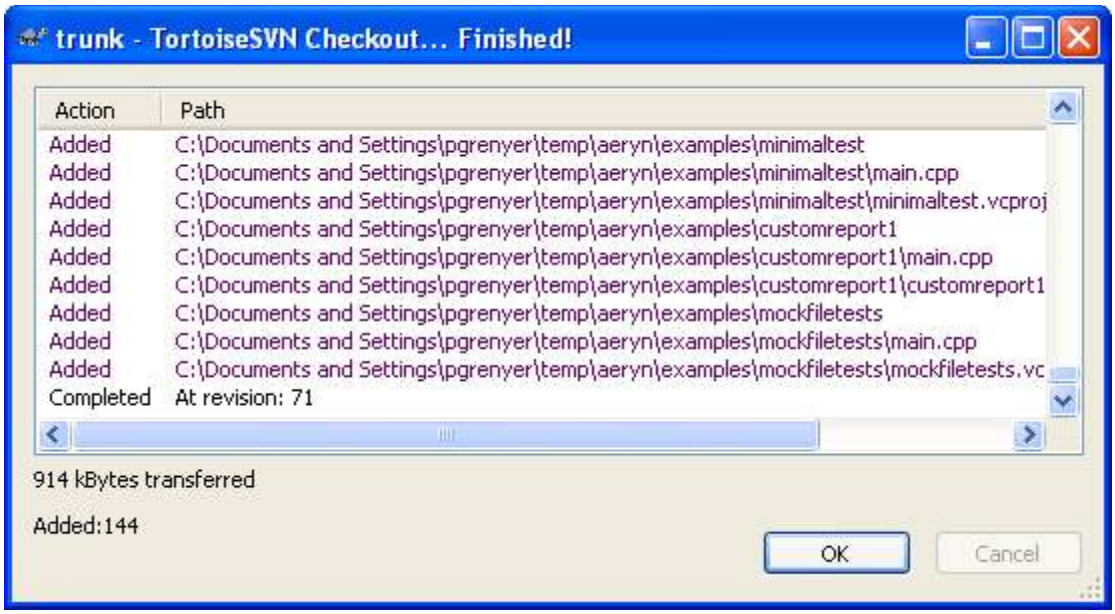
That will bring up the "TortoiseSVN: Checkout" dialog. Enter the Aeryn URL and add "aeryn" to the "Checkout Directory" as shown below:



Click “Ok”. A dialog will come up asking if you want to create the “Aeryn” folder:

Click “Yes”. Then another dialog box will come up asking for a username and password. Enter either “guest” and no password or your Tigris username and password and click “Ok”.

The Aeryn repository will then be checkout:



The revision number will be greater than 71. Dismiss the dialog by clicking “Ok”

Download

Aeryn can be downloaded from: http://www.aeryn.co.uk/

What do I need to build Aeryn?

Aeryn uses up-to-date C++ and therefore requires a modern compiler. It has been tested on, and provides make files or project files for the following compilers:

Microsoft Visual C++ 7.1
MinGW
GNU G++
Xcode

How do I build Aeryn?

Aeryn consists of a set of public header files and a static library (**corelib**), which must be built before Aeryn can be used. Once the library has been built all you need to do is give your programming environment access to the include files in **`aeryn2/include`** and link against the appropriate (depending on compiler) static library in **`aeryn2/build/lib`**.

Compiler	Library name
Microsoft Visual C++ 7.1	`aeryncorelib_debug.lib` (debug) `aeryncorelib.lib` (release)
MinGW	`libaeryn_core.a`
g++	`libaeryn_core.a`

Aeryn also comes with a number of other projects which are a collection of unit tests for Aeryn itself and some simple examples. The following describes how to build Aeryn's static library and associated unit tests.

There are two sets of unit tests. The first set, contained within the **tests** and **testrunner** projects are unit tests for Aeryn itself and the Simple Date class and example unit tests (kindly donated by Steve Love). The second set, contained in **testrunner2**, are unit tests for the new test registry feature (explained in detail later).

Microsoft Visual C++ 7.1

To build the Aeryn library and build and run the unit tests for Microsoft Visual C++ 7.1, simply open the Aeryn2 solution, located in the top level Aeryn directory, and select Build Solution from the Build menu.

MinGW

To build the Aeryn library and unit tests for MinGW, simply open a command prompt and change to the top level Aeryn directory and type:

```
mingw32-make
```

g++

To build the Aeryn library and unit tests for g++, simply open a command prompt and change to the top level Aeryn directory and type:

```
make
```

All compilers run both sets of tests as part of the build system. The output looks like this, but the copyright message and number of tests might be different:

```
Aeryn 2.1.2 (c) Paul Grenyer 2005-2006
http://www.aeryn.co.uk/
------------------------------------------------------------

Ran 48 tests, 48 Passed, 0 Failed.

…

Aeryn 2.1.2 (c) Paul Grenyer 2005-2006
http://www.aeryn.co.uk/
------------------------------------------------------------

Ran 4 tests, 4 Passed, 0 Failed.
```

All tests should pass. If they don't, please contact me via the Aeryn website: http://www.aeryn.co.uk.

Test Condition Macros

Aeryn uses a number of macros to actually test conditions in code. Generally speaking, in C++, macros are frowned upon. However, test code is not production code and many leading experts agree that testing is a legitimate use for macros. Without macros writing test conditions, yielding important information such as the file and line number would become very cumbersome.

An example of a test condition is as follows:

```
IS_TRUE( lifeTheUniverseAndEverything == 42 );
```

If `lifeTheUniverseAndEverything` holds the value 42 the expression `lifeTheUniverseAndEverything == 42`, evaluates to true and the code put in place by the `IS_TRUE` macro does nothing. If `lifeTheUniverseAndEverything` holds something different to 42 an exception of type `TestFailure` is thrown.

Tests are intended to pass. Test failure is an unexpected condition which causes an exception to be thrown. This of course prevents further tests from running. The fail case must be fixed first before further tests are run.

The `TestFailure` class has the following member functions:

Function	Return Type	Description
`Failure()`	`std::string`	A description of the failure. This varies for each test condition macro.
`Line()`	`Long`	The line number on which the failure occurred. This is determined by the `__LINE__` macro.
`File()`	`std::string`	The file in which the failure occurred. Whether this includes the full path to the file or just the file name is dependant on the compiler's implementation of the `__FILE__` macro.

The data member accessed by the corresponding member function is populated following a test condition failure and prior to the `TestFailure` instance being thrown.

The following test condition macros all have their own header file; however the `test_funcs.hpp` header file can be used to include *all* the test condition macros.

`IS_TRUE( code )`

Header file: `is_true.hpp`

Failure condition: The `IS_TRUE` test condition throws if the result of executing `code` evaluates to `false`. Eg:

```
IS_TRUE( false );
```

Failure message: A string containing the complete test condition. Eg:

```
IS_TRUE( false )
```

Example:

```
IS_TRUE( lifeTheUniverseAndEverything == 42 );
```

IS_FALSE(code)

Header file: `is_false.hpp`

Failure condition: The `IS_FALSE` test condition throws if the result of executing `code` evaluates to `true`. Eg:

```
IS_FALSE( true );
```

Failure message: A string containing the complete test condition. Eg:

```
IS_FALSE( true )
```

Example:

```
IS_ FALSE( lifeTheUniverseAndEverything != 42 );
```

FAILED(msg)

Header file: `failed.hpp`

Failure condition: The `FAILED` test condition *always* throws. It is intended to be used to indicate when code that should not be executed under normal conditions has been reached.

Failure message: The string literal contained within `msg`.

Example:

```
try
{
    …
}
catch( const std::exception& e )
{
    FAILED( e.what() );
}
```

IS_EQUAL(lhs, rhs)

Header file: `is_equal.hpp`

Failure condition: The `IS_EQUAL` test condition throws if the values held in `lhs` and `rhs` are not equal. Eg:

```
IS_EQUAL( 4, 2 );
```

Failure message: If the types of `lhs` and `rhs` are streamable a message detailing the failure is created. Eg:

```
'4' does not equal '2'.
```

If the types are not streamable the test condition is used. Eg:

```
IS_EQUAL( 4, 2 );
```

Example:

```
IS_EQUAL( lifeTheUniverseAndEverything, 42 );
```

There is an `IS_EQUAL` overload which ensures that the contents of string literals and `char` arrays pointed by `const char*` are compared instead of the address of the pointer.

IS_NOT_EQUAL(lhs, rhs)

Header file: `is_not_equal.hpp`

Failure condition: The `IS_NOT_EQUAL` test condition throws if the values held in `lhs` and `rhs` are equal. Eg:

```
IS_NOT_EQUAL( 42, 42  );
```

Failure message: If the types of `lhs` and `rhs` are streamable a message detailing the failure is created. Eg:

```
Expected not to get '42'.
```

If the types are not streamable the test condition is used. Eg:

```
IS_NOT_EQUAL( 42, 42 );
```

Example:

```
IS_NOT_EQUAL( lifeTheUniverseAndEverything, 43 );
```

There is an `IS_NOT_EQUAL` overload which ensures that the contents of string literals and `char` arrays pointed by `const char*` are compared instead of the address of the pointer.

`THROWS_EXCEPTION( statement, exception_type )`

Header file: `throws_exception.hpp`

Failure condition: The `THROWS_EXCEPTION` test condition throws if the statement contained in the `statement` parameter does not throw an exception of the type `exception_type`. Eg:

```
THROWS_EXCEPTION( NonThrower(), std::exception );
```

Where `NonThrower` is a class that does not throw from its constructor.

Failure message: A simple message saying that the code failed to throw an exception of the right type. E.g:

```
Failed to throw std::exception
```

Example:

```
IS_NOT_EQUAL( Thrower(), std::exception );
```

Where `Thrower` is a class that throws from its constructor.

`DOES_NOT_THROW_EXCEPTION( statement )`

Header file: `does_not_throw_exception.hpp`

Failure condition: The `DOES_NOT_THROW_EXCEPTION` test condition throws if the statement contained in the statement parameter throws an exception of any type. Eg:

```
DOES_NOT_THROW_EXCEPTION( Thrower() );
```

Where `Thrower` is a class that throws from its constructor.

Failure message:

```
Exception Thrown
```

Example:

```
IS_NOT_EQUAL( NonThrower() );
```

Where `NonThrower` is a class that does not throw from its constructor.

`MISSING_TEST( msg )`

Header file: `missing_test.hpp`

Failure condition: The `MISSING_TEST` test condition always throws. It is intended to be used to indicate that a test needs to be written.

Failure message: The string literal contained within msg.

Example:
```
MISSING_TEST( "Test needs to be written" )
```

`MISSING_TEST` works differently to all the other test condition macros. It throws `TestMissing` instead of `TestFailure` and is handled differently. It is intended to be used when a test is missing and needs to be written. It serves as a reminder to go back and write the test and is handled differently by the report interface.

Creating a Test Fixture

Creating a test fixture should be really easy and require the minimum of code to encourage people to create more tests. Some testing frameworks, such as CPPUnit [CPPUnit] and Test Crickett [TestCrickett] require all test fixtures to inherit from a particular class. This is overkill as, in most cases, a test fixture can require no more than a simple function and the power provided by a class is not needed. There are other times when a test fixture requires setup and tear down code and a class with a constructor and/or destructor provides a convenient mechanism.

Aeryn can run function based test fixtures, class based test fixtures and a combination of the two (e.g. a class with one or more static member functions).

Creating a Function based Test Fixture

To create a function based test fixture, simply write a function taking no parameters with a return type of `void` and place the test code and test conditions inside the function. For example:

```
void TestForTheMeaningOfLife()
{
    …
    IS_EQUAL( lifeTheUniverseAndEverything, 42 );
}
```

Sometimes it may be necessary to pass some form of context object [EncapsulateContextPattern] to the test fixture. This can be achieved by adding a *single* parameter of any type. For example:

```
void TestForTheMeaningOfLife(int lifeTheUniverseAndEverything)
{
    IS_EQUAL( lifeTheUniverseAndEverything, 42 );
}
```

Creating a Class based Test Fixture

To create a class based test fixture, simply write a class and put any setup code in the constructor and tear down code in the destructor (remembering that a destructor should never throw [Sutter]). For example:

```
class CalculatorTest
{
private:
    Calculator* calc_;

public:
    CalculatorTest()
    {
        calc_ = new Calculator;
    }

    ~CalculatorTest()
    {
        delete calc_;
    }

    void TestBasics()
    {
        double result = calc_->evaluate("1 + 1");
        IS_TRUE(2.0 == result);
    }
};
```

Sometimes it may be necessary to pass some form of context object [EncapsulateContextPattern] to the test fixture. This can be achieved by adding a *single* parameter of any type to the constructor. For example:

```
class TestForTheMeaningOfLife
{
private:
    const int lifeTheUniverseAndEverything_;

public:
    TestForTheMeaningOfLife( int lifeTheUniverseAndEverything)
      : lifeTheUniverseAndEverything_( lifeTheUniverseAndEverything )
    {
    }

    …
};
```

Class based test fixtures can be written so that the setup and teardown code (constructor and destructor) is called once regardless of how many separate test functions there are or once for each test function.

To call the setup and teardown code once for the whole test fixture make all the test functions `private` and write a single `public` function which calls the other test functions and create a test case for the `public` function (see later section). For example:

```
class CalculatorTest
{
    ...

public:
    void Run()
    {
        TestBasics();
        TestVariables();
        TestCompound();
    }

private:
    void TestBasics()
    {
        ...
    }

    void TestVariables()
    {
        ...
    }

    void TestCompound()
    {
        ...
    }

};

…

TestCase( "Calculator tests",
          Incarnate( &CalculatorTest::Run ) );
```

To call the setup and teardown code for each test function, make all the test functions public and create a test case for each one. For example:

```
class CalculatorTest
{
    ...

public:
    void TestBasics()
    {
        ...
    }

    void TestVariables()
    {
        ...
    }

    void TestCompound()
    {
        ...
    }

};

…

TestCase calculatorTests[] =
{
 TestCase( "Basics", Incarnate( &CalculatorTest::TestBasics ) ),
 TestCase( "Variables", Incarnate( &CalculatorTest::TestVariables) ),
 TestCase( "Compound", Incarnate( &CalculatorTest::TestCompound) ),
 TestCase()
};
```

Creating Test Cases

A test case is a wrapper for a test fixture function (which can be a standalone function or a class member function) which enables it to be given a name and added to a `TestRunner`. The `TestCase` class has two constructors. One takes a name used to identify the test fixture function and the test fixture function itself. The second constructor takes just the test fixture function and the name defaults to an empty string.

Creating a Test Case for a Function based Test Fixture

Creating a test case for a function based test fixture is straight forward. Simply pass the name of the test fixture function to the constructor of `TestCase` with or without a name. For example:

```
TestCase( "Life the universe and everything",
          TestForLifeTheUniverseAndEverything );
```

or

```
TestCase( TestForLifeTheUniverseAndEverything );
```

If a function based test fixture takes a context object it must be specified, along with the test fixture using the `FunctionPtr` class, when creating a test case. For example:

```
const int lifeTheUniverseAndEverything = 42;

…

TestCase( "Life the universe and everything",
          FunctionPtr( TestForLifeTheUniverseAndEverything,
                       lifeTheUniverseAndEverything ) );
```

or

```
const int lifeTheUniverseAndEverything = 42;

…

TestCase( FunctionPtr( TestForLifeTheUniverseAndEverything,
                       lifeTheUniverseAndEverything ) );
```

Creating a Test Case for a Class based Test Fixture

The `Incarnate` class is needed to create a test case for a class based test fixture, so that the test fixture is not instantiated until it is run. To create a test case pass the qualified test fixture function to the `Incarnate` class and then pass the `Incarnate` class, with or without a name to the `TestCase` constructor. For example:

```
TestCase( "Basics",
          Incarnate( &CalculatorTest::TestBasics ) );
```

or

```
TestCase( Incarnate( &CalculatorTest::Run ) );
```

If a class based test fixture takes a context object it must be specified, along with the test fixture function, and passed as a second parameter to `Incarnate`, when creating a test case. For example:

```
const int lifeTheUniverseAndEverything = 42;

…

TestCase( "Life the universe and everything",
          Incarnate( &TestForLifeTheUniverseAndEverything::Run,
                     lifeTheUniverseAndEverything ) );
```

or

```
const int lifeTheUniverseAndEverything = 42;

…

TestCase( Incarnate( &TestForLifeTheUniverseAndEverything::Run,
                     lifeTheUniverseAndEverything ) );
```

The `USE_NAME` Macro

The test fixture name passed to a test case is often the same as the test fixture function name itself. Therefore Aeryn provides a macro which extracts the test fixture function name and passes it to the test case. For example, the following function based test fixtures:

```
TestCase( USE_NAME( TestForLifeTheUniverseAndEverything ) );

TestCase( USE_NAME(
              FunctionPtr( TestForLifeTheUniverseAndEverything,
                           lifeTheUniverseAndEverything ) ) );
```

Both result in the following:

```
TestForLifeTheUniverseAndEverything
```

being used as the test fixture name. `USE_NAME` works in the same way for class based test fixtures removing `Incarnate&` from the test fixture function name.

Adding and Running Test Cases

Test cases must be added to a test runner in order to be run. `TestRunner` is declared in `test_runner.hpp` and is default constructible:

```
#include <aeryn/test_runner.hpp>

…

using namespace Aeryn;

TestRunner testRunner;
```

Once an instance of `TestRunner` has been created, test cases can be added to it. There are basically two ways of adding tests cases: one at a time or any number as an array. Along with the test case(s) a name for the test set (one or more test cases added together) can also be added. If the test set name is not specified it defaults to a blank string.

Adding a Single Test Case

To add a single test case to a test runner, simply pass it to `TestRunner`'s `add` member function with or without a test set name. For example:

```
testRunner.Add( "HHGTTG",
                TestCase(
                       USE_NAME( TestForTheMeaningOfLife ) ) );
```

Or

```
testRunner.Add( TestCase(
                     USE_NAME( TestForTheMeaningOfLife ) ) );
```

Adding an Array of Test Cases

To add an array of test cases to a test runner, simply pass it to `TestRunner`'s `Add` member function with or without a test set name. For example:

```
TestCase calculatorTests[] =
{
    TestCase( "Basics",
              Incarnate( &CalculatorTest::TestBasics ) ),
    TestCase( "Variables",
              Incarnate( &CalculatorTest::TestVariables ) ),
    TestCase( "Compound",
              Incarnate( &CalculatorTest::TestCompound ) ),
    TestCase()
};

...

testRunner.Add( "Calculator", calculatorTests );
```

or

```
TestCase calculatorTests[] =
{
    TestCase( "Basics",
              Incarnate( &CalculatorTest::TestBasics ) ),
    TestCase( "Variables",
              Incarnate( &CalculatorTest::TestVariables ) ),
    TestCase( "Compound",
              Incarnate( &CalculatorTest::TestCompound ) ),
    TestCase()
};

...

testRunner.Add( calculatorTests );
```

The test case array must be terminated by a default constructed `TestCase`. This is so that the test runner can detect when it has reached the end of the array.

`USE_NAME` Macro

The `USE_NAME` macro can also be used with `TestRunner`'s `Add` member function. For example:

```
testRunner.Add( USE_NAME( calculatorTests ) );
```

will give the test set the name:

```
CalculatorTests
```

Running Tests

To run the test cases that have been added to the test runner, simply call `TestRunner`'s `Run` member function. For example:

```
testRunner.Run();
```

There are four of overloads of the `TestRunner::Run()` function. The simplest takes no arguments and uses the minimal report (which is described later) and therefore writes the results of the tests to `cout`. All the versions of the `TestRunner::Run()` member function return `0` if all the tests pass and `1` if there are any failures. This makes it ideal for use as a `main` return value if, for example, running the tests is integrated into an Integrated Development Environment (IDE) or a build system.

The features described above are all that is needed to run tests using the Aeryn testing framework. The following sections contain optional extras.

The other `TestRunner::Run()` overloads are detailed below:

```
int Run( const IReport& report )
```

Runs all the reports using the report specified by `report`. Any one of the supplied reports (described later) or a custom report can be used. For example:

```
…
TestRunner testRunner;
…
VerboseReport verboseReport;
Return testRunner.Run( verboseReport );
…
```

```
int Run( const CommandLineParser& commandLine )
```

Selects which tests to run, which report to use and controls other behaviour according to a `CommandLineParser` object. For example:

```
...
TestRunner testRunner;
...
CommandLineParser commandLineParser( argv );
return testRunner.Run( commandLineParser );
...
```

The `CommandLineParser` class and its use are explained later in detail.

```
int Run(  const CommandLineParser& commandLine,
          IReport& report )
```

Selects which tests to run and controls other behaviour according to a `CommandLineParser` object. The report specified by the `CommandLineParser` object is overridden by the report passed via `report`. For example:

```
...
TestRunner testRunner;
...
CommandLineParser commandLineParser( argv );
VerboseReport verboseReport;
return testRunner.Run( commandLineParser, verboseReport );
...
```

The `CommandLineParser` class and its use are explained later in detail.

Reports

Aeryn uses a call back mechanism to report on tests. Custom reports must implement the `IReport` interface. There are a number of different types of report provided with Aeryn.

By default the test runner uses the minimal report. To use another report, either a provided one or a custom one, simply declare it and pass it to one of the `TestRunner::Run()` overloads. For example:

```
int main()
{
    TestRunner testRunner;
    …
    VerboseReport report;
    return testRunner.Run( report );
}
```

Each of the provided reports is compiler aware and will give an appropriate failure message for the corresponding IDE (Integrated Development Environment) so that it can be clicked on in the output window to take the user to the failed test. For example failure messages for Microsoft Visual C++ look like this:

```
Test : IsTrue( false )
c:\...\test_func_test.cpp(86): IS_TRUE( false ) failed.
```

And for GCC (KDevelop):

```
Test    : IsTrue( false)
test_func_test.cpp:86: IS_TRUE( false) failed.
```

Terse Report

Header file:	`terse_report.hpp`	
Constructor Parameters:	`out`	A reference to an output stream to write the main report to.
	`err`	A reference to an output stream to write the report progress (".", "F", etc) to.
Author:	Pete Goodliffe	

The terse report (contributed by Pete Goodliffe) is based on the CPPUnit [CPPUnit] reporting system and displays the Aeryn copyright message followed by a full stop (period) for each successful test, an F for failed tests an E for errors and an N for missing tests followed by the number of passes and number of failures. The output from a successfully run set of tests should look like this:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------
.................................
Ran 33 tests, 33 Passed, 0 Failed.
```

If one or more test cases fail, the names of the failed tests are displayed along with the test condition failure message, the line the failure occurred on and the file the failure occurred in *after* all the tests have run. Again, the number of test cases run, the number of passes and the number of failures is displayed:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------
.......F.......................

------------------------------------------------------------
Test    : IsTrue( true )
c:\aeryn2\tests\test_func_test.cpp(49): IS_TRUE( true ) failed.
------------------------------------------------------------
Ran 33 tests, 32 Passed, 1 Failed.
```

If there are any tests missing, identified by using the `MISSING_TEST` test condition macro, it is indicated as follows:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------
......N........................
Ran 33 tests, 32 Passed, 0 Failed, 1 Missing.
```

Minimal Report

Header file:	`minimal_report.hpp`	
Constructor Parameters:	`out`	A reference to an output stream to write the main to.
Author:	Paul Grenyer	

The minimal report does not give any information about the test cases unless one or more fail. The output of a successfully run set of test cases gives only the Aeryn copyright message, the number of test cases run, the number of passes and number of failures (which is obviously zero):

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------

Ran 23 tests, 23 Passed, 0 Failed.
```

If one or more test cases fail, the names of the failed tests are displayed along with the test condition failure message, the line the failure occurred on and the file the failure occurred in. Again, the number of test cases run, the number of passes and the number of failures is displayed:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------
Test    : IsTrue( true )
c:\aeryn2\tests\test_func_test.cpp(49): IS_TRUE( true ) failed.
------------------------------------------------------------

Ran 23 tests, 22 Passed, 1 Failed.
```

If there are any tests missing, identified by using the `MISSING_TEST` test condition macro, it is indicated as follows:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------

Test    : IsTrue( true )
c:\aeryn2\tests\test_func_test.cpp(81): Test missing.
------------------------------------------------------------

Ran 33 tests, 32 Passed, 0 Failed, 1 Missing.
```

The minimal report is used by default if no other report is specified.

Verbose Report

Header file:	`verbose_report.hpp`	
Constructor Parameters:	`out`	A reference to an output stream to write the main to.
Author:	Paul Grenyer	

The verbose report lists all test sets and test cases that are given a name. Those that are not named are not displayed as there is no useful information. Following the copyright message and the list of tests, the number of test cases run, the number of passes and number of failures are displayed:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------

Test Set : Report Tests

 - Minimal Report Test
 - Verbose Report Test
 - Gcc Report Test
------------------------------------------------------------
```

```
Test Set : Test Function Tests

 - IsTrue( true )
 - IsTrue( false )
 - IsFalse( false )
 - IsFalse( true )
 - IsEqual( true )
 - IsEqual( false )
 - IsNotEqual( false )
 - IsNotEqual( true )
 - IsEqual missing operator<<
 - IsNotEqual missing operator<<
 - Failure
 - THROWS_EXCEPTION( throw std::bad_alloc();, std::bad_alloc
 - THROWS_EXCEPTION( "", std::bad_alloc )
------------------------------------------------------------

Test Set : Calculator Test

 - Run
 - All calculator tests
------------------------------------------------------------

Test Set : Context object tests

 - Function
 - Class
------------------------------------------------------------

Ran 23 tests, 23 Passed, 0 Failed.
```

If one or more test cases fail the failure message, the line the failure occurred on and the file the failure occurred in are displayed immediately below the test case name:

```
...

Test Set : Test Function Tests

 - IsTrue( true )
c:\aeryn2\tests\test_func_test.cpp(49): IS_TRUE( true ) failed.
 - IsTrue( false )
 - IsFalse( false )
 - IsFalse( true )
 - IsEqual( true )
 - IsEqual( false )
 - IsNotEqual( false )
 - IsNotEqual( true )
 - IsEqual missing operator<<
 - IsNotEqual missing operator<<
 - Failure
 - THROWS_EXCEPTION( throw std::bad_alloc();, std::bad_alloc
 - THROWS_EXCEPTION( "", std::bad_alloc )
------------------------------------------------------------

...

------------------------------------------------------------

Ran 23 tests, 22 Passed, 1 Failed.
```

If there are any tests missing, identified by using the `MISSING_TEST` test condition macro, it is indicated as follows:

```
Aeryn 2.1.1 (c) Paul Grenyer 2005
http://www.aeryn.co.uk/
------------------------------------------------------------

...

 - IsTrue( true )
c:\..\aeryn2\tes
ts\test_func_test.cpp(81): Test missing.

...

------------------------------------------------------------

Ran 33 tests, 32 Passed, 0 Failed, 1 Missing.
```

XCode Report

Header file:	`xodel_report.hpp`	
Constructor Parameters:	`out`	A reference to an output stream to write the main to.
Author:	Thaddaeus Frogley	

The XCode [XCode] report (constributed by Thaddaeus Frogley) uses a file based cookie to stop the tests being run if the code has not been changed. It is a template and can therefore take on the behaviour of any one of the provided reports or a custom report via its template parameter. For example:

```
int main( int argc, char *argv[] )
{
      using namespace Aeryn;
      TestRunner testRunner;
      Aeryn::AddTests( testRunner );

      XcodeReport< MinimalReport >
                  report( std::cerr, "filename.txt" );
      return testRunner.Run( report );
}
```

Creating a Custom Report

Custom reports can be created by implementing the `IReport` interface which is found in `Aeryn/ireport.hpp`.

```
#ifndef AERYN_IREPORT_H
#define AERYN_IREPORT_H

#include <aeryn/noncopyable.hpp>
#include <string>

namespace Aeryn
{
    class TestFailure;

    class IReport : private Utils::Noncopyable
    {
    public:
        virtual ~IReport
            () = 0;

        virtual void BeginTesting
            ( const std::string& header,
              unsigned long testCount ) = 0;

        virtual void BeginTestSet
            ( const std::string& testSetName ) = 0;

        virtual void BeginTest
            ( const std::string& testName ) = 0;

        virtual void Pass
            ( const std::string& testName ) = 0;

        virtual void Failure
            ( const std::string& testName,
              const TestFailure& failure ) = 0;

        virtual void MissingTest
            (  const std::string& testName,
               const TestMissing& missingTest ) = 0;

        virtual void Error
            ( const std::string& testName,
              const std::string& errorDetails ) = 0;

        virtual void EndTest
            ( const std::string& testName ) = 0;

        virtual void EndTestSet
            ( const std::string& testSetName ) = 0;

        virtual void EndTesting
            ( unsigned long testCount,
              unsigned long failureCount,
              unsigned long missingCount ) = 0;
    };
}

#endif // AERYN_IREPORT_H
```

`aeryn/test_failure.hpp` must also be included, usually in the custom report's cpp file as `TestFailure` is only forward declared in `ireport.hpp`.

To implement the `IReport` interface, simply inherit from and override each of the pure virtual functions. An instance of your custom report can then be passed to `TestRunner::Run`. Each pure virtual function in the `IReport` interface is explained below:

BeginTesting

`header`	A `std::string` object containing the Aeryn header, including the copyright message.
`testCount`	The number of test cases that will be run.

The `BeginTesting` function is called prior to the first test set.

BeginTestSet

`testSetName`	The name of the test set that is about to be run.

The `BeginTestSet` function is called at the start of each test set.

BeginTest

`testName`	The name of the test case about to be run.

The `BeginTest` function is called prior to each test case.

Pass

`testName`	The name the test case that passed.

The `Pass` function is called immediately after each test case that passes.

Failure

`testName`	The name the test case that failed.
`failure`	A `TestFailure` object containing the details of the test case that failed.

The `Failure` function is called immediately after each test case that fails.

MissingTest

testName	The name of the missing test.
failure	A TestMissing object containing the details of the test missing test.

The MissingTest function is called immediately after a missing test is identified by the MISSING_TEST test condition marco.

Error

testName	The name the test case that cause an error.
errorDetails	A std::string describing the error caused by the test case.

The Error function is called immediately after each test case that results in an error.

EndTest

testName	The name of the test case that was just run.

The EndTest function is called after each test case.

EndTestSet

testSetName	The name of the test set that was just run.

The EndTestSet function is called after each test set.

EndTesting

testCount	The number of test cases run.
failureCount	The number of test cases that failed or resulted in an error.
missingCount	The number of missing tests.

The EndTesting function is called after all test sets and their associated test cases have been run.

Command Line Parser

Aeryn comes with a rich command line interface, which is provided by the CommandLineParser class and is used as follows:

```
#include <aeryn/test_runner.hpp>
#include <aeryn/test_name_not_found.hpp>
#include <aeryn/command_line_parser.hpp>
...

int main( int, char *argv[] )
{
	using namespace Aeryn;
	int result = -1;

	try
	{
		TestRunner testRunner;
		// Add tests here
		CommandLineParser commandLineParser( argv );
		result = testRunner.Run( commandLineParser );
	}
	catch( const Exception& e )
	{
		std::cout << e.what() << std::endl;
	}

	return result;
};
```

There are two `TestRunner::Run()` overloads which take a `CommandLineParser` object. These are described in the Running Tests section above.

The `CommandLineParser` class simply takes and processes the arguments specified at the command line and passed into main via the `argv[]` variable. When passed to a `TestRunner::Run()` overload, the `CommandLineParser` object is used to control the behaviour of Aeryn.

Using the `CommandLineParser` class it is possible to specify which tests and test sets to run and which report to use. This can result in `TestRunner::Run()` throwing an exception if, for example, the name of a test, test set or report that doesn't exist is specified. Therefore, when using the `CommandLineParser` a try/catch block, as shown above, must be used. All exceptions thrown by `TestRunner::Run()` inherit from `Aeryn::Exception`, which in turn inherits from `std::runtime_error`.

The available command line arguments are as follows:

`-h` `--help`	Displays the list of available command line arguments.
`-nh` `--noheader`	Suppresses the Aeryn header and copyright message from the standard reports.
`-t <test name>` `--test <test name>`	Runs the test specified by <test name>. If the test does not exist `TestNameNotFound` is thrown. If there is more than one test with the specified name `DuplicateTestNameFound` is thrown.
`-ts <set name>` `-testset <set name>`	Runs the test set specified by <set name>. If the test set does not exist `TestSetNameNotFound` is thrown. If there is more than one test set with the specified name `DuplicateTestSetNameFound` is thrown.
`-r <report name>` `--report <report name>`	Use the report specified by <report name>. If the report does not exist `BadReportName` is thrown. The names for the standard report are: `minimal, verbose, terse, xcode` Custom reports can also be specified from the command line using the `ReportFactory` class, which is explained later.
`-lt` `--list-test-names`	Lists the names of all the tests.
`-lts` `--list-test-set-names`	Lists the names of all the test sets.

Making Custom Reports available from the Command Line

Custom reports, described above in the Creating a Custom Report section, can also be specified from the command line by creating a custom report factory. The `ReportFactory` class is used to create reports specified by the `CommandLineParser` class.

The `ReportFactory` class uses the `Create` member function, along with the name of the required report, to create Aeryn reports. For example:

```
ReportFactory reportFactory;
IReportPtr report =
            reportFactory.Create( VerboseReport::Name() );
```

To add a custom report, you must inherit from `ReportFactory`, add a static function which creates the custom report and register that static function with the report's name. For example:

```
class CustomReport : public IReport
{
public:
	static std::string Name()
	{
		return "custom report";
	}

	...
};

...

class CustomReportFactory : public ReportFactory
{
public:
	CustomReportFactory()
		: ReportFactory()
	{
		// Register custom report.
		Add( CustomReport::Name(), &CreateCustomReport );
	}

	static IReportPtr CreateCustomReport()
	{
		return IReportPtr( new CustomReport );
	}
};
```

You can also specify your custom report, or any other report, as the default which is created when a report isn't specified at the command line, by registering it as the default:

```
class CustomReportFactory : public ReportFactory
{
public:
	CustomReportFactory()
		: ReportFactory()
	{
		// Register custom report as default report.
		// defaultTestName is defined by ReportFactory.
		Add( defaultTestName &CreateCustomReport );
	}

	...
};
```

or:

```
class CustomReportFactory : public ReportFactory
{
public:
	CustomReportFactory()
		: ReportFactory()
	{
		// Register verbose report as default report.
		// defaultTestName and CreateVerboseReport are
		// defined by ReportFactory.
		Add( defaultTestName & CreateVerboseReport);
	}

	…
};
```

Finally the custom report factory must be passed to `TestRunner`'s constructor:

```
...
try
{
	CustomReportFactory reportFactory;
	TestRunner testRunner( reportFactory );
	...
	CommandLineParser commandLineParser( argv );
	result = testRunner.Run( commandLineParser );
}
catch( const Exception& e )
{
	...
}
...
```

Test Registry

The test registry, contributed by Pete Goodliffe, is a singleton [Singleton] test runner wrapper that, along with its associated macros, allows tests to be registered in any source file without having to pass around a `TestRunner` instance by pointer or reference. `TestRegistry` and its associated macros can be found in the `test_registry.hpp` header file.

`TestRegistry` has a single static member function called `GetTestRunner` that is used to access the wrapped `TestRunner` instance and must be used everywhere a `TestRunner` instance would be used. For example in `main`:

```
int main(int /*argc*/, char *argv[])
{
      using namespace Aeryn;
      int result = -1;

      try
      {
            CommandLineParser commandLineParser( argv );
            result =
                  TestRegistry::GetTestRunner().Run(
                                    commandLineParser );
      }
      catch( const Exception& e )
      {
            std::cout << e.what() << std::endl;
      }

      return result;
}
```

There are three test registry macros that are used to add tests to the test runner. Each one calls the appropriate `TestRunner::Add` member function. Each macro creates a uniquely named static variable which causes the specified tests to be added to the test runner wrapped by the test registry.

`REGISTER_TESTS( tests )`

Adds a single test or array of tests to the test runner without specifying a name. The name for the test or test set is generated from the `__FILE__` macro. This means, of course, that if `REGISTER_TESTS` is used twice or more in the same file, then two or more tests or test sets with the same name will be created. If that name is then used from the command line to run the test or test set, a `DuplicateTestName` or `DuplicateTestSetName` exception will be thrown.

`REGISTER_TESTS_WITH_NAME( tests, name )`

Adds a single test or array of tests to the test runner with the specified a name.

`REGISTER_TESTS_USE_NAME( name )`

Adds a single test or array of tests to the test runner and applied the `USE_NAME` macro to generate a name.

An example of the use of the test registry and its associated macros can be found in `simple_calc_test.cpp` in the `extras/testrunner2` directory:

```
// simple_calc_test.cpp

void SimpleCalcTest::AddTest()
{
    IS_EQUAL( 4, SimpleCalc::Add( 2, 2 ) );
    IS_NOT_EQUAL( 3, SimpleCalc::Add( 2, 2 ) );
    IS_NOT_EQUAL( 5, SimpleCalc::Add( 2, 2 ) );
}

void SimpleCalcTest::SubtractTest()
{
    IS_EQUAL( 5, SimpleCalc::Subtract( 10, 5 ) );
    IS_NOT_EQUAL( 4, SimpleCalc::Subtract( 10, 5 ) );
    IS_NOT_EQUAL( 6, SimpleCalc::Subtract( 10, 5 ) );
}

void SimpleCalcTest::MultiplyTest()
{
    IS_EQUAL( 4, SimpleCalc::Multiply( 2, 2 ) );
    IS_NOT_EQUAL( 3, SimpleCalc::Multiply( 2, 2 ) );
    IS_NOT_EQUAL( 5, SimpleCalc::Multiply( 2, 2 ) );
}

void SimpleCalcTest::DivideTest()
{
    IS_EQUAL( 2, SimpleCalc::Divide( 10, 5 ) );
    IS_NOT_EQUAL( 1, SimpleCalc::Divide( 10, 5 ) );
    IS_NOT_EQUAL( 3, SimpleCalc::Divide( 10, 5 ) );

    DOES_NOT_THROW_EXCEPTION( SimpleCalc::Divide( 10, 5 ) );
    THROWS_EXCEPTION( SimpleCalc::Divide( 10, 0 ),
                      DivideByZero );
}

namespace
{
    TestCase addSubtractTests[] =
    {
        TestCase( USE_NAME( SimpleCalcTest::AddTest) ),
        TestCase( USE_NAME( SimpleCalcTest::SubtractTest) ),
        TestCase()
    };

    TestCase multiplyTest[] =
    {
        TestCase( USE_NAME( SimpleCalcTest::MultiplyTest) ),
            TestCase()
    };

    TestCase divideTest[] =
    {
        TestCase( USE_NAME( SimpleCalcTest::DivideTest) ),
        TestCase()
    };
```

```
    REGISTER_TESTS( addSubtractTests );

    REGISTER_TESTS_WITH_NAME( "Multiply Test", multiplyTest );

    REGISTER_TESTS_USE_NAME( divideTest );
}
```

Mainlib

Projects using the test registry will generally all have the same form of `main()` as there is no need for custom code to add tests to the test runner and full control over which tests and reports to run can be achieved using the command line parser. Therefore Aeryn provides a canonical version of `main()`:

```
int main(int /*argc*/, char *argv[])
{
      using namespace Aeryn;
      int result = -1;

      try
      {
            CommandLineParser commandLineParser( argv );
            result =
                  TestRegistry::GetTestRunner().Run(
                                          commandLineParser );
      }
      catch( const Exception& e )
      {
            std::cout << e.what() << std::endl;
      }

      return result;
}
```

in the form of a static library that can be linked into Aeryn test projects. The Mainlib project can be found in `extras/mainlib`. It is built and used as part of the TestRunner2 project (also found in `extras`) by both the Aeryn2 Microsoft Visual C++ solution and Aeryn make files. The libraries can be found in `build/lib` and are named as follows:

Compiler	Library name
Microsoft Visual C++ 7.1	`aerynmain_debug.lib` (debug) `aerynmain.lib` (release)
MinGW	`libaeryn_main.a`
g++	`libaeryn_main.a`

Glossary of Terms

Term	Meaning
Test Condition	A test condition is a piece of code being tested using one of Aeryn's test condition macros such as `IS_EQUAL`. For example: `IS_EQUAL( lifeTheUniverseAndEverything, 42 );`

Test Fixture	A test fixture is a class or a function containing test conditions. For example:
	
	`void TestForTheMeaningOfLife()`
	`{`
	`  IS_EQUAL( lifeTheUniverseAndEverything, 42 );`
	`}`
	
	A test fixture is passed to Aeryn's `TestRunner` in order to run its tests.

Test Case	A test case is a wrapper for a test fixture which enables it to be given a name and run by Aeryn.
Context Object	An object of any type which is passed to a test fixture containing extra information needed by the test.
Test Runner	A test runner runs test cases.
Test Set	A set of one or more test cases added to a test runner at the same time and, therefore, given the same name.

Frequently Asked Questions

1. Why do source and header files have lower case, underscore separated names when Aeryn classes have camel case names?

I have never been a fan of the lower case with underscores (e.g. `auto_ptr`) naming convention used by standard C++ and have always preferred the Microsoft Visual C++ camel case (e.g. `AutoPtr`) naming convention.

I also favour lower case file names and originally all Aeryn source and header file names matched the classes they contained, but in lower case (for example `TestRunner` was declared in test_runner.hpp).

A particular Aeryn user expressed difficulty in reading the file names and asked me to introduce underscores. I was happy to do this and found I preferred the way they looked. I also introduced the more C++ orientated .hpp extension for header files.

References

[SVN] http://subversion.tigris.org/

[Tigris] http://www.tigris.org/

[TortoiseSVN] http://tortoisesvn.tigris.org/

[CPPUnit] http://cppunit.sourceforge.net/cgi-bin/moin.cgi

[Test Crickett] http://www.crickett.co.uk/cpp_unit_testing.php

[EncapsulateContextPattern] http://allankelly.net/patterns/encapsulatecontext.pdf

[Sutter] Item 16 - Exceptional C++ by Herb Sutter. ISBN: 0201615622.

[KDevelop] KDevelop: http://www.kdevelop.org/

[XCode] XCode: http://www.apple.com/macosx/features/xcode/

[Singleton] Design patterns : elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vissides. ISBN: 0201633612.