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.

Where can I get Aeryn?

Aeryn can be downloaded from: http://www.paulgrenyer.dyndns.org/aeryn.

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 3.2.3
GNU G++ 3.2.3 & 3.4.3
Xcode 1.5

It *may* be possible to get Aeryn to compile on Microsoft Visual C++ 6.0.

How do I build Aeryn?

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.

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, which yield 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 `TestFailire`, is thrown.

Tests are intended to pass. Therefore a test failing is an exceptional condition and therefore justifies the throwing of an exception. Throwing an exception does of course prevent the test conditions following a failure from being tested, but if a failure has occurred this should be fixed before proceeding to further test conditions, which are quite likely to fail anyway.

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 compilers implementation of the `__FILE__` macro.

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

`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:

```
Expected '4' but got '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 statment contained in the `statment` 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 `THROWS_EXCEPTION` test condition throws if the statement contained in the statement parameter throws a 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.

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",
          Incaranate( &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 calulcatorTests[] =
{
 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, reformats it by capitalising the first letter and inserting spaces prior to each following capital letter, 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:

```
Test For Life The Universe And Everything
```

being used as the test fixture name. `USE_NAME` works in the same way for class based test fixtures removing `Incarnate&` and the class name 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 `testrunner.h` and is default constructible:

```
#include <aeryn/testrunner.h>

…

using namespace Aeryn;

TestRunner testRunner;
```

Once an instance of `TestRunner` has bee 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:

```
Calculator Tests
```

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 two overloads of the `TestRunner` `Run` function. One takes no arguments and the other takes a report object. The version which takes no arguments uses the minimal report (which is described later) and therefore writes the results of the tests to `cout`. Both versions of the `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.

Complete Example

```
#include <aeryn/testrunner.h>

class CalculatorTest
{
    ...

public:
    void TestBasics()
    {
        ...
    }

    void TestVariables()
    {
        ...
    }

    void TestCompound()
    {
        ...
    }

};

using namespace Aeryn;


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

int main()
{
    TestRunner testRunner;
    testRunner.Add( "Calculator", calculatorTests );
    return testRunner.Run();
}
```

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 reports provided with Aeryn. These are discussed below.

Minimal Report

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 (c) Paul Grenyer 2005
http://www.paulgrenyer.dyndns.org/aeryn
------------------------------------------------------------

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

If one or more test cases fail its name is 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 (c) Paul Grenyer 2005
http://www.paulgrenyer.dyndns.org/aeryn
------------------------------------------------------------

Test    : IsTrue( true )

Failure : IS_TRUE( true ) failed.
Line    : 48
File    : c:\sandbox\protected\aeryn2\tests\testfunctest.cpp

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

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

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

Verbose Report

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 (c) Paul Grenyer 2005
http://www.paulgrenyer.dyndns.org/aeryn
------------------------------------------------------------

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 its 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 )

Failure : IS_TRUE( true ) failed.
Line    : 48
File    : c:\sandbox\protected\aeryn2\tests\testfunctest.cpp


 - 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.
```

GCC Report

The GCC report reports failures in a way that is compatible with GCC IDEs (Integrated Development Environment) such as KDevelop [kDevelop], so that if the tests are run as part of the build process failures can be double clicked on to go straight to the point of failure:

```
Aeryn 2 (c) Paul Grenyer 2005
http://www.paulgrenyer.co.uk/aeryn

testfunctest.cpp:48: error: Test Failure - 'IsTrue( true )' in
'Test Function Tests' - IS_TRUE( true ) failed.
```

XCode Report

The XCode [XCode] report is the same as the GCC report, but uses a file based to cookie to stop the tests begin run if the code has not been changed.

Controlling Reports with Command Line Arguments

Aeryn includes an easy way to run different reports depending on the first command line argument passed to the test application at runtime. This is achieved using the `TestRunner::CreateReport` function. For example:

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

    TestRunner testRunner;
    Aeryn::AddTests( testRunner );

    TestRunner::IReportPtr
              report( TestRunner::CreateReport( argc, argv) );

    return testRunner.Run( *report.get() );
}
```

The command line argument options are:

Command Line Argument	Report
	Minimal
`verbose`	Verbose
`gcc`	GCC
`xcode`	XCode

Creating a Custom Report

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

```
#ifndef AERYN_IREPORT_H
#define AERYN_IREPORT_H

#include <aeryn/noncopyable.h>
#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 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 ) = 0;
    };
}

#endif // AERYN_IREPORT_H
```

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

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` objectcontaining 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.

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.

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

References

[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/