# Research Report

for

# Automatic Random Regression Testing with Human Oracle

*Prepared by: Ivan Maguidhir, C00002614, December 2010*

## Table of Contents

## 1. Introduction

The project is titled Automatic Random Regression Testing with Human Oracle. The aim of the project is to take code given to it by the user, to generate tests using the xUnit testing framework and for these tests to generate sets of random values. For each set of random values generated for each test the project must run the test with those values and present the result of the test to the user[1]. The user will then interact with the project to accept or reject the outcome of the test. If the test result is rejected by the user it indicates a problem with the system under test (i.e. a bug in application being tested) and the project must query the user for expected result. Regardless of whether or not the test outcome is accepted or rejected by the user the project must save the test, its randomly generated values and the actual result (if the user accepted the test result) or the expected result (if the user rejected the test result) as the expected result of the test going forward. The consultation with the user to get them to accept or reject the test result is referred to in the project title as “Human Oracle”. The word random in the project title refers to both the nature of generated test values and the way in which previously saved tests are applied going forward. Regression testing is simply testing something that worked at one point in time to make sure that it still works[2]. The project will be written using the C++ programming language and will use the xUnit framework for creating and running tests which it generates. For this reason, most of the research in this document relates to xUnit and in particular CppUnit (an implementation of xUnit for the C++ language).

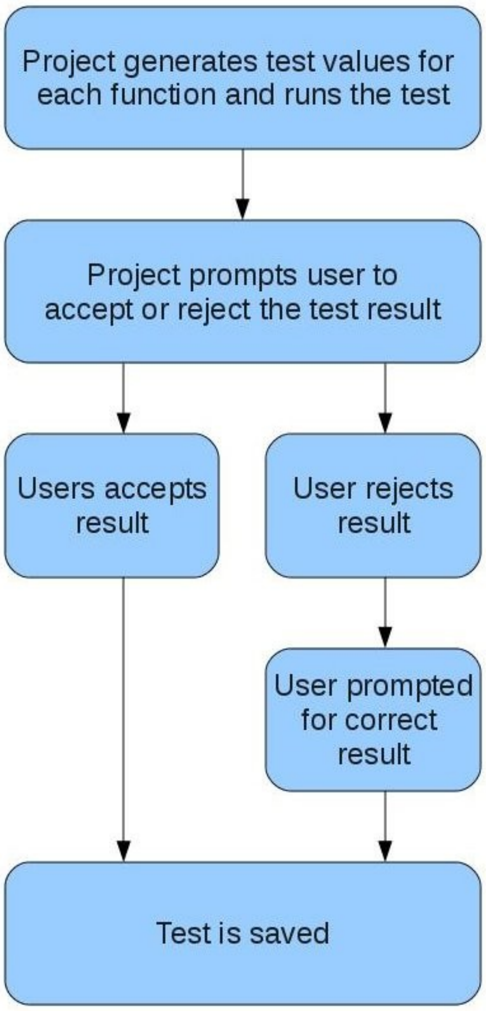

Project generates test values for each function and runs the test
Project prompts user to accept or reject the test result
Users accepts result
User rejects result
User prompted for correct result
Test is saved

*Illustration 1: Outline of test creation*

# *2. xUnit*

## 2.1 Unit Testing

The term “unit” when used in relation to software means the smallest collection of code which can be treated as a whole. Examples of units are classes, components or sets of related functions[2][3].The developer of a particular unit is responsible for testing it to ensure that it functions exactly as designed. This type of testing is known as unit testing. It improves the quality of code and causes the developer to focus on the functional requirements of each unit[4]. Both of these help to reduce errors which are inadvertently added during coding. The psychology of the developer with regard to testing carried out by the Independent Test Group outlined in Pressman's book[2] should provide a huge incentive for developers to write thorough test cases for their work. Developers are protective of their creative work[2] and as a result, given the right tools, should be interested in finding and fixing problems before anyone else has the opportunity “to pick holes in it”. Unit testing seems to have originated from and around use of the Smalltalk programming language during the 1970s. Kent Beck was one of its earliest proponents[4]. It is noteworthy that although Pressman's book[2] specifies unit testing as a responsibility of the developer, not all software companies enforce this and in some cases all responsibility with regard to testing ends up being delegated to the ITG.

## 2.2 Kent Beck's Paper

In 1989 Kent Beck published a paper entitled “Simple Smalltalk Testing: With Patterns”[5]. In it he outlines a strategy and framework for testing Smalltalk code where the unit tests are written, executed and results checked using Smalltalk code. He writes that tests done like this are much more stable than user interface script tests – where a small cosmetic change in the user interface can create false positives[6]. In introducing the framework he also writes that one of its aims is to allow creation of a common set of data on which each test operates but that is created and thrown away for each test as opposed to an approach where single set of test data is created and all of the tests are run using it. According to Beck the latter approach can cause problems because tests interact, that is part of a future test depends on the result of a previous test[6], so that the failure of one test can have a cascading effect. The paper provides a set of patterns (a pattern is a recurring solution to a recurring problem in a context[7]) for writing and running tests[6]:

| Pattern | Purpose |
|---|---|
| Fixture | Create a common set of test data to be used by some tests. |
| Test Case | Stimulate the test fixture and make some predictions about how it should react. |
| Check | Test the fixture to make sure your predictions came true |
| Test Suite | A set of test cases compiled with a view to running them with one command |

*Table 1: Kent Beck's Patterns*

The framework described should distinguish between errors (errors that the developer did not anticipate e.g. unhandled exceptions) and failures (errors that were covered by a test case written by the developer)[6]. The implementation of this framework for Smalltalk was called SUnit however unit testing was not widely adopted until Kent Beck and Erich Gamma collaborated on implementing the framework in the Java language as JUnit[7][8]. Today, Kent Beck's outline of a unit testing framework has been implemented for nearly every language, in fact, you can usually find more than one implementation for a given language. Collectively all of these tools are know as xUnit.

## 2.3 CppUnit

## 2.3.1 Introduction

CppUnit is a port of JUnit for the C++ programming language and is therefore included in the xUnit family. There are several versions available written by different people. The version of CppUnit available on SourceForge.Net is cross-platform, that is it can be compiled on many different systems. It is an extension of work done by Michael Feathers and Jerome Lacoste who both individually created ports which were operating system specific[9].

### *2.3.2 Implementation of the xUnit Patterns*

**Test Case**
The Test Case pattern is implemented in the `CppUnit::TestCase` class. In order to use it it needs to be sub-classed[10], for example:

```
class MyTest : public CppUnit::TestCase {
…
};
```

The `TestCase` class includes a virtual[1] method `runTest()` which should be overridden and defined to contain code for whatever test needs to be carried out. The test should make one or more calls to `CPPUNIT_ASSERT(bool)`. The `CPPUNIT_ASSERT(bool)` function is provided by the CppUnit library to provide a mechanism for making tests fail conditionally. Without a call to `CPPUNIT_ASSERT(bool)` a test will never fail and will be therefore useless[10].

**Fixture**
The Fixture pattern is implemented in the `CppUnit::TestFixture` class. In order to use it it needs to be sub-classed[10], for example:

```
class MyTests : public CppUnit::TextFixture {
…
};
```

Inside the declaration for the `TestFixture` derived class some variables can be defined. These variables can be initialized to hold values which will aid in the execution of tests and will be common to all tests carried out using this fixture. A systematic way of initializing the variables to a state, that the developer decides on, before each test is required. Additionally, a way of tidying up will be required in the event that memory was allocated during initialization. Any memory allocated needs to be deallocated once a test has finished running in order to avoid a memory leak. The `TestFixture` class provides exactly both of these functionalities in the form of two virtual methods `setUp()` and `tearDown()[10]`. The virtual `setUp()` method is designed to be overridden in classes derived from `TestFixture`. Within the definition for the overridden `setUp()` method values can be assigned to variables, memory can be allocated and the addresses

1 A method declared `virtual` in C++ means that any derived class can override the definition of that method. There is an exception to this in the case of methods declared `virtual` when `=0` is appended to the declaration e.g. `virtual void doStuff() =0;` this means that the class is abstract, that no implementation is provided for the method and the method <u>must</u> be overridden for the derived class not to be considered abstract. An abstract class cannot be instantiated into an object.

assigned to pointers etc. The framework calls this method once before each and every test is run[10]. The virtual `tearDown()` method is again designed to be overridden. The purpose of this method is to provide an opportunity to deallocate any memory which was allocated during `setUp()`. For example, any use of the `new` operator in `setUp()` will have a corresponding `delete` statement in `tearDown()` and likewise every call to `malloc()` in `setUp()` will have a call to `free()` in `tearDown()`. The framework calls `tearDown()` after each and every test is run[10]. To add tests to a `TestFixture` derived class the developer simply adds a method to the derived class for each test she wishes to add[10]. For example `myTest1()`, `myTest2()` and `myTest3()` below:

```
class MyTests : public CppUnit::TextFixture {
public:

void setUp();
void tearDown();

void myTest1();
void myTest2();
void myTest3();

};
```

To run individual tests belonging to the created `TestFixture` derived class the developer needs to instantiate a `CppUnit::TestCaller<TestFixture>` object to which a string representing the name of the test and a pointer to the actual method of the class (e.g. `&myTest1()`) are passed using the constructor[10]. For example:

```
CppUnit::TestCaller<MyTests> myTestCaller("My first test",
&MyTests::myTest1);
```
[2]

Following this the developer must instantiate a `CppUnit::TestResult` object which will hold the results of the test after it has been run[10] as follows:

```
CppUnit::TestResult result;
```

The developer can then run the test by calling the run() method of the `TestCaller` (which is inherited from the `TestCase` class) object with a pointer to the test method passed into it[10][11]:

```
myTestCaller.run(&result);
```

**Test Suite**

The Test Suite pattern is implemented in the `CppUnit::TestSuite` class. To use it the developer simply instantiates a `TestSuite` object[10]:

```
CppUnit::TestSuite mySuite;
```

2 Notice the angled brackets used with the `TestCaller` class. This means that the `TestCaller` class needs a template parameter in order for it to be instantiated. In this case the parameter expected is a class. The compiler will actually regenerate the source-code of the `TestCaller` class at compile time for each different class parameter that the developer uses throughout her code. The reason it is being used here is to accommodate the different class names assigned by the developer to derived classes of `TestFixture`.

and begins to add tests in the form of `TestCaller` objects (described above) using the `addTest()` method, which the class provides, and passing in a pointer to a new `TestCaller` object, for example:

```
mySuite.addTest(new CppUnit::TestCaller<MyTests>("My first test",
&MyTests::myTest1));
```

It is important to note that the `new` operator must be used here because the `TestSuite` class manages memory allocation for its constituent tests. During destruction, the `TestSuite` class will attempt to use the `delete` operator on all of the pointers to `TestCaller` objects that it contains. In this way the class assumes both that it allocated the memory for each test itself and that the memory was allocated using the `new` operator. Passing in pointers to `TestCaller` objects which were created elsewhere as a parameter will cause problems[10][11]. Finally and importantly, the `TestSuite` class can be used to manage not only `TestCase` and `TestFixture` objects but may also manage further suites of tests. Specifically any class which implements the `CppUnit::Test` interface can be added using the `addTest()` method[10].

**Check**
The Check pattern is implemented in the `CppUnit::TestResult` class. A pointer to a `TestResult` object is a required parameter of the `TestCaller::run()` and `TestSuite::run()` methods and therefore is used to store the results of any test within the scope of CppUnit. A `TestResult` object keeps track of both errors and failures encountered with a specific test[10].

### *2.3.3 Additional classes*

**TestRunner**
The `TestRunner` class provides a mechanism for the developer to run a `TestSuite` and display its results. The alternative is for the developer to write code to process a `TestResult` object and create her own messages for each result[10]. The suggested way to use the `TestRunner` class is to add a static method to a `Fixture` which creates and returns a suite of tests[10], in the header file something similar to the following would be used:

```
class MyTests : public CppUnit::TextFixture {
public:

static CppUnit::Test* suite();

void setUp();
void tearDown();

void myTest1();
void myTest2();
void myTest3();

};
```

and in the .cpp file something similar to the following would be added :

```
CppUnit::Test* MyTests::suite() {
     CppUnit::TestSuite* newSuite =
new CppUnit::TestSuite("MyTests");
     newSuite->addTest(new CppUnit::TestCaller<MyTests>("My first
test", &MyTests::myTest1));
     newSuite->addTest(new CppUnit::TestCaller<MyTests>("My second
test", &MyTests::myTest2));
     newSuite->addTest(new CppUnit::TestCaller<MyTests>("My third
test", &MyTests::myTest3));
     return newSuite;
}
```

A TestRunner used as follows will display the results of the running the tests on the console:

```
int main(int argc, char** argv) {
  CppUnit::TextUi::TestRunner runner;
  runner.addTest(MyTests::suite());
  runner.run();
  return 0;
}
```

### *2.3.4 Worked examples*

In order to provide a walk-through of how CppUnit is used we can define some functions first, write CppUnit tests and verify that they work according to our requirements. Following this we will look at a more advanced example by implementing a class and testing its methods in the same way (i.e. a complete unit test). I'm using the file names example.h and example.cpp to hold the implementations of functions / classes and test.h and test.cpp to hold the implementation of the CppUnit tests. This is for the sake of simplicity, file names and class names are arbitrary and any can be used in practice.

**Higher and Lower functions**
We create two functions both of which take a pair of integers as parameters and return an integer. In the header file (example.h) we have:

```
int higher(int a, int b);
int lower(int a, int b);
```

and in the .cpp file (example.cpp) we have:

```
int higher(int a, int b) {
     return (b > a) ? b : a;
}
int lower(int a, int b) {
     return (b < a) ? b : a;
}
```

We can see that `higher()` compares the integer values in `a` and `b`. If the value in `b` is greater than the value in `a` then the function returns the value stored in `b`. If the value in `a` is greater than the value in `b` or both values are equal then the function returns the value stored in a. We can see that `lower()`, likewise, compares the integer values in `a` and `b`. If the value in `b` is less than the value in `a` then the function returns the value stored in `b`. If the value in `a` is less than the value in `b` or the two values are equal then the function returns the value stored in `a`. Now that we have our example functions implemented we can write a few tests to make sure they work.

| Function | Tests |
|---|---|
| `higher` | • Create two integer variables `a` and `b` with values `5` and `10` respectively.<br>• Call `higher(a,b)` and check that `10` is the result.<br>• Call `higher(b,a)` and check that `10` is the result.<br>• Call `higher(b,b)` and check that `10` is the result |
| `lower` | • Create two integer variables `a` and `b` with values `5` and `10` respectively.<br>• Call `lower(a,b)` and check that `5` is the result.<br>• Call `lower(b,a)` and check that `5` is the result.<br>• Call `lower(a,a)` and check that `5` is the result. |

*Table 2: Tests for the higher and lower functions*

We notice that the tests formulated for `higher()` and `lower()` have some things in common. They both require the same variables and values to carry out the tests. We can use the Fixture pattern mentioned earlier in the document to create this common set of variables and values. We implement our Fixture as follows, in our header file (test.h) we have:

```
#include <cppunit/TestFixture.h>

class HigherLowerTests : public CppUnit::TestFixture {
public:
     // CppUnit methods
     void setUp();
     void tearDown();
     // Our tests
     void higherTest();
     void lowerTest();
private:
     // Variables which will assist our testing
     int a_;
     int b_;
};
```

and in our source code file (test.cpp) we have:

```
#include "test.h"
#include "example.h"
#include <cppunit/TestAssert.h>

void HigherLowerTests::setUp() {
     a_ = 5;
     b_ = 10;
}
void HigherLowerTests::tearDown() {
     // nothing to be done no memory allocated in setUp()
}
void HigherLowerTests::higherTest() {
     CPPUNIT_ASSERT(higher(a_,b_) == b_);
     CPPUNIT_ASSERT(higher(b_,a_) == b_);
     CPPUNIT_ASSERT(higher(b_,b_) == b_);
}
void HigherLowerTests::lowerTest() {
     CPPUNIT_ASSERT(lower(a_,b_) == a_);
     CPPUNIT_ASSERT(lower(b_,a_) == a_);
     CPPUNIT_ASSERT(lower(a_,a_) == a_);
}
```

In order to run these tests we need to create `TestCaller` objects for both and run them using a `TestRunner` object to get the results. We could use a `TestSuite` either but because there are only two tests and for simplicity's sake we won't. We write the code to run the two tests as follows:

```
#include "example.h"
#include "test.h"
#include <cppunit/TestCaller.h>
#include <cppunit/ui/text/TestRunner.h>

int main()
{
     CppUnit::TextUi::TestRunner runner;
     runner.addTest(new CppUnit::TestCaller<HigherLowerTests>
("Tests for higher() function", &HigherLowerTests::higherTest));
     runner.addTest(new CppUnit::TestCaller<HigherLowerTests>
("Tests for lower() function", &HigherLowerTests::lowerTest));
     runner.run();
     return 0;
}
```

When we run this program we get the following output on the console from the `TestRunner` object: “`OK (2 tests)`”. Our code successfully passed the tests that we wrote. To see what happens when a test fails, we can introduce an error into our code. We replace the line **`return`**`(b < a) ? b : a;` in `lower()` with: **`return`**` (b < a) ? b : b;`.Running our tests again we now get the following output from the `TestRunner` object on the console:

```
1) test: Tests for lower() function (F) line: 18 ../src/test.cpp
assertion failed
- Expression: lower(a_,b_) == a_
```

The `TestRunner` not only tells us that one of the tests failed but gives us the test name we assigned to it, the line number of the failed assertion and its expression. If we had a lot of tests this information would make it much easier to find the failed test in our code.

**String class**

A more advanced CppUnit example can be illustrated using a String class which manages and performs simple operations on strings. This will demonstrate how tests can be written for an entire unit/class. An implementation of such a class with a limited set of features is as follows. The header file (example.h) contains this code:

```
class String {
public:
     // constructors
     String();
     String(const String &str);
     String(const char* str);
     // destructor
     ~String();
     // assignment operators
     String& operator=(String rhs);
     String& operator=(const char* rhs);
     // equality operators
     bool operator==(String& other) const;
     bool operator!=(String& other) const;
     // string operations
     unsigned long length();
     char getAt(unsigned long index);
     String& reverse();
private:
     // the actual NULL terminated string
     char* string_;
     void assign(const char* string);
     char* get();
     // access to private members for class doing testing
     friend class StringTests;
};
```

The .cpp file (example.cpp) contains the following code:

```
#include "example.h"
#include "string.h"
#include "stddef.h"

String::String() {
     // assign the empty string
     string_ = NULL;
     assign("");
}
String::~String() {
     // deallocate memory
     if (string_ != NULL) {
          delete[] string_;
          string_ = NULL;
     }
```

```
}
String::String(const String& str) {
     // construct a String object from an existing String object
     string_ = NULL;
     assign(str.string_);
}

String::String(const char* str) {
     // construct a String object from a char* string
     string_ = NULL;
     assign(str);
}
String& String::operator =(String& rhs) {
     // assign the value of another String object to this object
     char* string = rhs.get();
     assign(string);
     return *this;
}

String& String::operator =(const char* rhs) {
     // assign a char* string to this object
     assign(rhs);
     return *this;
}
bool String::operator ==(String& other) const {
     // == operator
     if (string_ != NULL) {
          char* with = other.get();
          if (with != NULL) {
               if (strcmp(string_, with) == 0)
                    return true; // true if strings are equal
          }
     }
     // default, return false if either string is NULL
     // or if strings not equal
     return false;
}
bool String::operator !=(String& other) const {
     // != operator. Uses the == operator already implemented.
     return !(*this == other);
}
void String::assign(const char* string) {
     if (string != NULL) // can't assign NULL
     {
          // delete the existing string if any
          if (string_ != NULL) {
               delete[] string_;
               string_ = NULL;
          }
          // copy the string parameter to the string_ member
          unsigned long len = strlen(string);
          string_ = new char[len+1];
          strcpy(string_, string);
     }
}
char* String::get() {
     // return a pointer to the string
     return string_;
}
unsigned long String::length() {
     unsigned long len = 0;
     // get the length of the string
     if (string_ != NULL) // string_ should never be NULL
          len = strlen(string_);
     return len;
}
char String::getAt(unsigned long index) {
     // get the length of the string
     unsigned long len = length();
     if (index >= len)
          return -1; // return -1 if index doesn't exist
     else
          return string_[index]; // otherwise return the character
}
String String::reverse()
{
     // create a String object to hold the result
     String reversed;
     if (string_ != NULL) // string_ should never be NULL {
          // copy characters to the new string in reverse order
          unsigned long oldChar = length();
          unsigned long newChar = 0;
          char* result = new char[oldChar+1];
          while (oldChar)
               result[newChar++] = string_[--oldChar];
          reversed = result;
     }
     return reversed;
}
```

Given that each method in the String class exists because it performs some function, we know also that it must be possible to write a test to check that this function has been performed correctly after it has been called. By reading through the code of each method and understanding what work each method, by itself, is meant to do we can start to formulate tests:

| Method | Tests |
|---|---|
| **String**() | Check that the new string is equal to the empty string “”. |
| **String**(**String&** str) | Check that the new string is equal to str. |
| **String**(**const char*** str) | Check that the new string is equal to String(str). |
| **operator =**(**String&** rhs) | Check that the string is equal to rhs. |
| **operator =(const char*** rhs) | Check that the string is equal to String(rhs). |
| **operator ==** | Check that two String objects with equivalent values return true when compared using this operator. |
| **operator !=** | Check that two String objects with different values return true when compared using this operator. |
| **length**() | Create a string of any length and check that this method returns the correct length |
| **getAt**() | 1. Create a string of any length and use this method to request a character at a valid index. Check that the character is correct.<br>2. Create a string of any length and use this method to request a character at an invalid index. Check that -1 is returned. |
| **reverse**() | Create a String object with a string of any length. Create a second String object and assign to it the result of reverse() called on the first String object. Assign the second string to the value returned by reverse() called on itself. Check that the first and second strings are now equal. |

*Table 3: Tests for the String class*

The tests can be implemented using CppUnit as follows, the test.h header file will look like this:

```
#include <cppunit/TestFixture.h>
#include <cppunit/TestSuite.h>
#include <cppunit/TestCaller.h>
class StringTests : public CppUnit::TestFixture {
public:
     // CppUnit methods
     void setUp();
     void tearDown();
     static CppUnit::Test* suite();
     // Our tests
     // tests for constructors
     void constructorTest1();
     void constructorTest2();
     // tests for the operator = method
     void assignmentTest1();
     void assignmentTest2();
     // tests for the operator == method
     void equalityTest1();
     // tests for the operator != method
     void inequalityTest1();
     // tests for the length method
     void lengthTest1();
     // tests for the getAt method
     void getAtTest1();
     void getAtTest2();
     // tests for the reverse method
     void reverseTest1();
private:
     // Variables which will assist our testing
     char* testString1_;
     char* testString2_;
};
```

and the test.cpp source code file will look like this:

```
#include "test.h"
#include "string.h"
#include "example.h"
#include <cppunit/TestAssert.h>
void StringTests::setUp() {
     const char* string1 = "Test String 1";
     const char* string2 = "Test String 2";
     testString1_ = new char[strlen(string1)+1];
     testString2_ = new char[strlen(string2)+1];
     strcpy(testString1_, string1);
     strcpy(testString2_, string2);
}
void StringTests::tearDown() {
     if (testString1_ != NULL) {
          delete[] testString1_;
          testString1_ = NULL;
     }
     if (testString2_ != NULL) {
          delete[] testString2_;
          testString2_ = NULL;
     }
}
CppUnit::Test* StringTests::suite() {
     CppUnit::TestSuite* newSuite = new CppUnit::TestSuite("String
test suite");
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Constructor test 1",
                &StringTests::constructorTest1));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Constructor test 2",
                     &StringTests::constructorTest2));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Assignment test 1",
                     &StringTests::assignmentTest1));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Assignment test 2",
                     &StringTests::assignmentTest2));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Equality test 1",
                          &StringTests::equalityTest1));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Inequality test 1",
                          &StringTests::inequalityTest1));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Length test 1",
                          &StringTests::lengthTest1));
     newSuite->addTest(new CppUnit::TestCaller<StringTests>("GetAt
test 1",
                          &StringTests::getAtTest1));
     newSuite->addTest(new CppUnit::TestCaller<StringTests>("GetAt
test 2",
                          &StringTests::getAtTest2));
     newSuite->addTest(new
CppUnit::TestCaller<StringTests>("Reverse test 1",
                          &StringTests::reverseTest1));
     return newSuite;
}
void StringTests::constructorTest1() {
     // this test covers the default constructor
     String test;
     String test2("");
     CPPUNIT_ASSERT(test == test2);
}
void StringTests::constructorTest2() {
     // this test covers the char* and String& constructors
     String test(testString1_);
     String test2(test);
     CPPUNIT_ASSERT(test == test2);
}
void StringTests::assignmentTest1() {
     // this test covers assignment of String
     String test(testString1_);
     String test2 = test;
     CPPUNIT_ASSERT(test == test2);
}
void StringTests::assignmentTest2() {
     // this test covers assignment of char*
     String test(testString1_);
     String test2 = testString1_;
     CPPUNIT_ASSERT(test == test2);
}
void StringTests::equalityTest1() {
     String test(testString1_);
     String test2(testString1_);
     CPPUNIT_ASSERT(test == test2);
}
void StringTests::inequalityTest1()
{
     String test(testString1_);
     String test2(testString2_);
     CPPUNIT_ASSERT(test != test2);
}
void StringTests::lengthTest1() {
     String test(testString1_);
     unsigned long actualLength = strlen(testString1_);
     unsigned long reportedLength = test.length();
     CPPUNIT_ASSERT(actualLength == reportedLength);
}
void StringTests::getAtTest1() {
     // test that the right character is returned by getAt
     // when the index is valid
     String test(testString1_);
     char actualCharacter = test.string_[0];
     char reportedCharacter = test.getAt(0);
     CPPUNIT_ASSERT(actualCharacter == reportedCharacter);
}
void StringTests::getAtTest2() {
     // test that -1 is returned by getAt
     // when the index is out of range
     String test(testString1_);
     unsigned long len = test.length();
     char reportedCharacter = test.getAt(len+1);
     CPPUNIT_ASSERT(reportedCharacter == -1);
}
void StringTests::reverseTest1() {
     // test that the reverse of the reverse of a string
     // is equal to that string
     String test(testString1_);
     String test2 = test.reverse();
     test2 = test2.reverse();
     CPPUNIT_ASSERT(test == test2);
}
```

We can run the test suite which our Fixture returns as follows:

```
int main()
{
	CppUnit::TextUi::TestRunner runner;
	runner.addTest(StringTests::suite());
	runner.run();
	return 0;
}
```

Cheered by our previous success with our tests on `higher()` and `lower()` we run the above program which runs our test suite sure that nothing can go wrong, after all we're very good programmers. The TestRunner outputs the following information on the console:

```
1) test: Reverse test 1 (F) line: 127 ../src/test.cpp
assertion failed
- Expression: test == test2
```

Oh no, our test on the `reverse()` method has failed. The output from the `TestRunner` doesn't help in identifying why the `String` object `test` was not equal to the `String` object `test2` but at least we know where to look for the error – in the `reverseTest1()` method of `StringTests` and in the `reverse()` method of `String`. We insert a breakpoint on the first line in `reverseTest1()` and run the debugger. We step over the code and notice something unusual about the `String` object value returned from the first call to `String::reverse()`. It contains the reversed string al-right, but there is a garbage/unexpected character at the end of the string. We stop the debugger and run it again this time stepping into the first call to `String::reverse()`. We take one look at the main code in this method:

```
// copy characters to the new string in reverse order
unsigned long oldChar = length();
unsigned long newChar = 0;
char* result = new char[oldChar+1];
while (oldChar)
	result[newChar++] = string_[--oldChar];
reversed = result;
```

and we can see our mistake straight away - we didn't initialize the last character in the memory allocated for `result` to the null terminator `0`. We could have achieved this either by specifically setting the last character to `0` or calling `memset()` on the allocated memory. While we are looking at the code we notice we have made a further mistake in not deallocating the memory we allocated for `result`. We update the code for `reverse()` to the following with modifications underlined:

```
String String::reverse()
{
	// create a String object to hold the result
	String reversed;

	if (string_ != NULL) // string_ should never be NULL
	{
		// copy characters to the new string in reverse order
		unsigned long oldChar = length();
```

```
        unsigned long newChar = 0;
        char* result = new char[oldChar+1];
        memset(result, 0, oldChar+1);
        while (oldChar)
            result[newChar++] = string_[--oldChar];
        reversed = result;
        delete[] result;
    }

    return reversed;
}
```

We run our tests again after the modifications to `reverse()` and see the following information from the `TestRunner` on the console: “`OK (10 tests)`”. All of our tests have now passed.

#### *2.3.5 Notes*

Some things I learned while creating the examples:

- That an object which represents a string shouldn't have a `NULL` value associated with a pointer representing its data at any stage, it should always have some value to start with – a pointer to a string containing some characters or a pointer to an empty string.
- Methods which return some revision of the object the method is being called on should return an object value containing the revision. This is so method chaining can take place and so that recursion doesn't take place. For example it allows the following to happen: `string = string.reverse().reverse();`
- Configuring applications to use libraries in Linux is much simpler than on Windows. This is because libraries are stored in the /usr/lib folder when installed. This means that to link to a specific library all that is required is the name of the library, as opposed to the same process on Windows where the full path of the library must be known and specified.

## *3. Similar existing software*

### 3.1 Introduction

The title of this section was originally Similar existing applications. I decided upon this name following a search of the internet for applications which perform a similar function to what is required of this project. That section title would be a misnomer now as I have discovered from actually using QtTestRunner and wxTestRunner that they are libraries requiring some implementation on the part of the developer as opposed to stand-alone applications. Both applications implement a functionality similar to that found in the JUnit testing framework for showing test results graphically, specifically the `junit.ui.TestRunner` class[12]. Although they are not applications in their own right QtTestRunner and wxTestRunner are to the best of my knowledge, the two pieces of software which are most closely related to this project. I was unable to find software which performs the entire set of functions outlined in the project proposal.

### 3.2 QtTestRunner

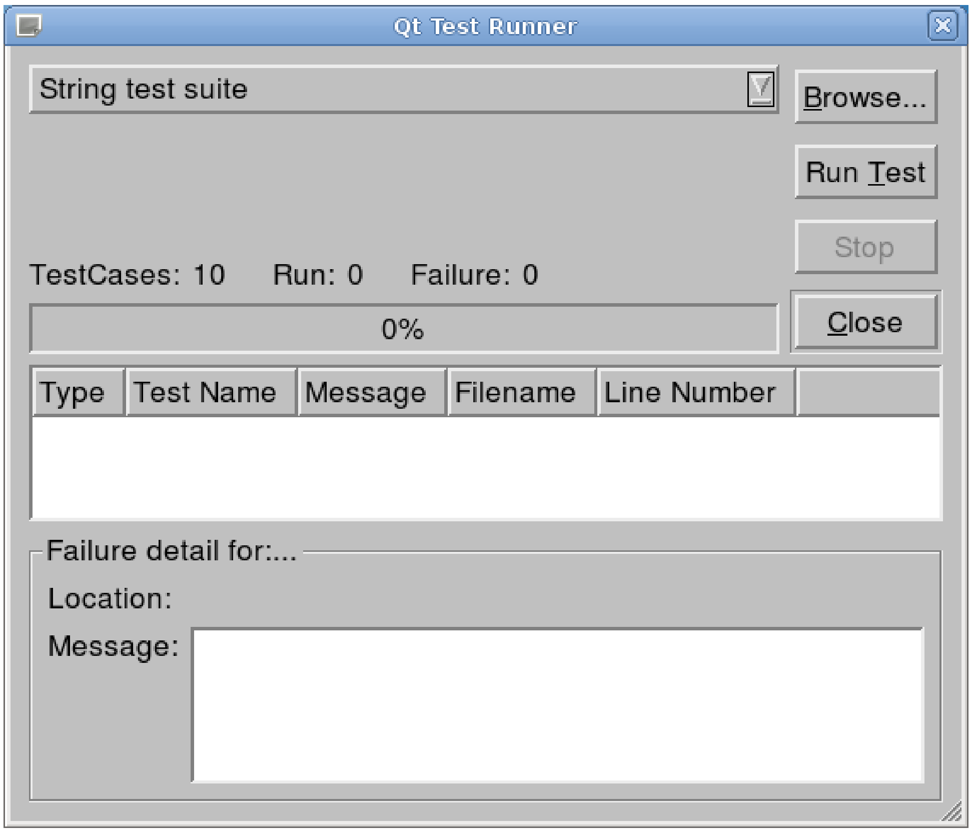
QtTestRunner is a GUI test runner for the CppUnit testing framework or more specifically is a library containing a graphical implementation of the text-based CppUnit::TestRunner class. It can be used to test any kind of C++ software not just applications written using Qt[3]. QtTestRunner is written using Qt therefore requires Qt to be installed before it can be used. The library was written using Qt version 2 and works with Qt version 3 but does not work with Qt version 4. This is due to the fact that Qt4 is not backwards compatible with Qt3. QtTestRunner is included with CppUnit versions 1.10.2 and greater. The library works on Linux/Unix and Windows[14]. As already mentioned the current version of QtTestRunner does not work with Qt version 4. I spent a few unhappy hours figuring this out the hard way. The result of attempting to link an application with both the Qt4 libraries and QtTestRunner library was, in my case, a segmentation fault before `main()` was called. In order to get the QtTestRunner library up and running on Debian Squeeze (the version of Debian Linux currently in testing), I needed to have the libqt3-mt, libqt3-mt-dev and qt3-dev packages installed and link my application with the libqt3-mt library by adding the qt-mt library name to the linker parameters in Eclipse. To put it simply, I needed to install and link my application with Qt3 (the previous version of Qt). I was then able to link my application with the QtTestRunner library and run it without issue. I created the following simple application to leverage some of the tests created earlier in this document as a demonstration of QtTestRunner:

```
#include "example.h"
#include "test.h"
#include <cppunit/ui/qt/QtTestRunner.h>
#include <qapplication.h>

int main(int argc, char* argv[])
{
     QApplication app(argc, argv);
     CppUnit::QtUi::TestRunner runner;
     runner.addTest(StringTests::suite());
     runner.run();
     return app.exec();
}
```

It is necessary, as can be seen from the code above, to initialize a Qt `QApplication` object in order to be able to use the Qt GUI components which QtTestRunner requires. When `runner.run()` is called the following appears on the screen:

3 “Qt is a cross-platform application and UI framework. Using Qt, you can write web-enabled applications once and deploy them across desktop, mobile and embedded operating systems without rewriting the source code.”[13]

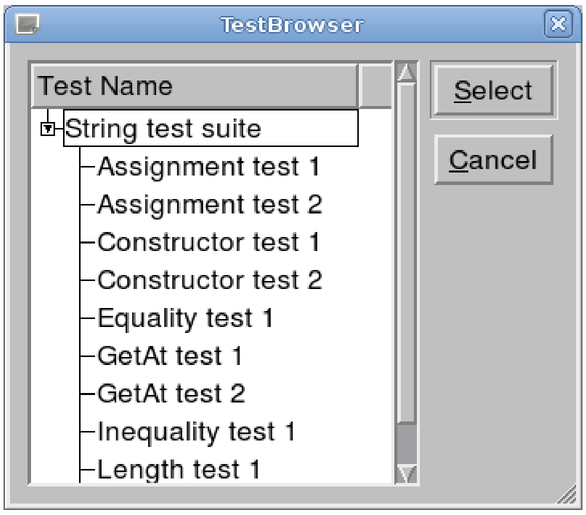


TestCases: 10 Run: 0 Failure: 0

0%

*Illustration 2: Main QtTestRunner dialog*

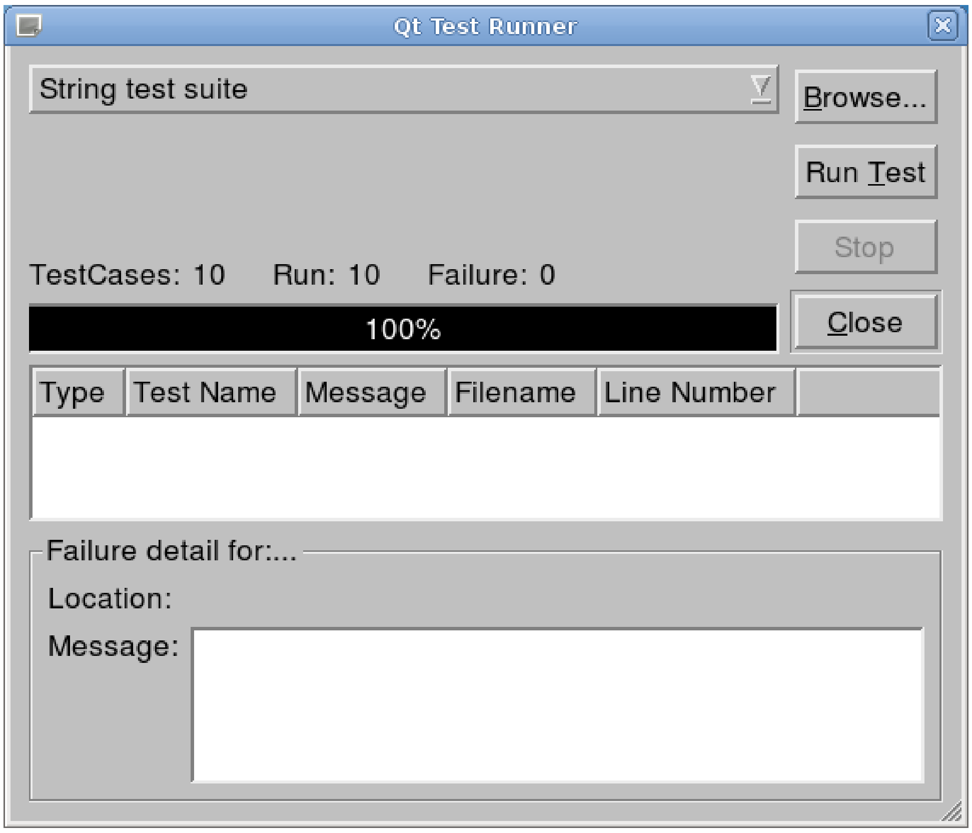
The drop down list at the top of the window shows the name of the selected test. In our case the TestSuite which we added is selected initially. We can select individual tests to run from within this suite by clicking the "Browse..." button and selecting a specific test from the list:



String test suite
- Assignment test 1
- Assignment test 2
- Constructor test 1
- Constructor test 2
- Equality test 1
- GetAt test 1
- GetAt test 2
- Inequality test 1
- Length test 1

*Illustration 3: QtTestRunner TestBrowser dialog*

Clicking the "Select" button returns us to the initial screen however if we selected a specific test from the "Test Name" list in the "TestBrowser" window that test will now be displayed in the combo-box on the initial screen. To run whatever test we have selected we use the "Run Test" button and the screen is updated:

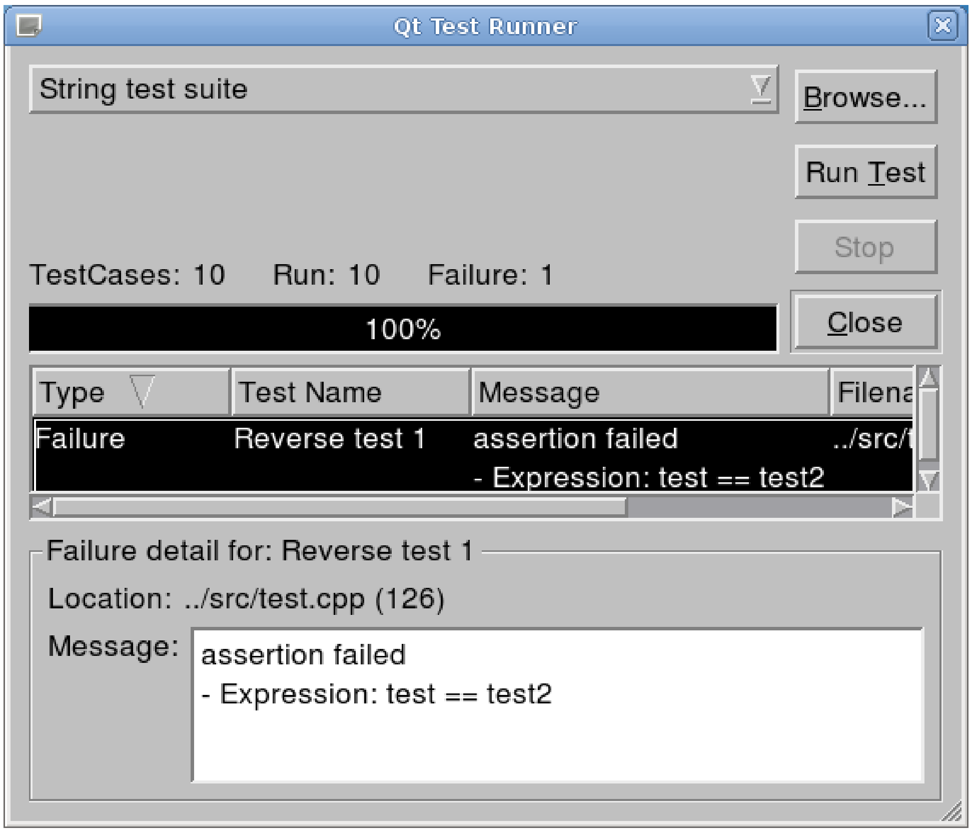


TestCases: 10 Run: 10 Failure: 0

100%

*Illustration 4: Main QtTestRunner dialog updated following a test run*

All of the tests contained in our test suite have been run and have passed. The GUI tells us the total number of test cases, the number of test cases run and the number of failures as well as providing a progress bar to demonstrate visually the progress of running the tests. This is would be an important feature if there were a large number of tests involved. Additionally the GUI provides a list of failures and a space for the details of the failure which we select from the list. In order to look at this more closely I have commented out the solution to the problem encountered with String::reverse() which was solved in the previous chapter. When we recompile the QtTestRunner sample application (including the String class), run the application again and click “Run Test” to run the suite of tests the GUI is updated as follows:


Qt Test Runner

String test suite

TestCases: 10 Run: 10 Failure: 1

100%

| Type | Test Name | Message | Filena |
|---|---|---|---|
| Failure | Reverse test 1 | assertion failed - Expression: test == test2 | ../src/t |

Failure detail for: Reverse test 1

Location: ../src/test.cpp (126)

Message: assertion failed
- Expression: test == test2

*Illustration 5: Main QtTestRunner dialog updated following a test failure*

It is noteworthy that the error we identified in the `String::reverse()` method does not always cause the test to fail. It only fails when the memory allocated for the result `char*` happens not to have a `0` value in its last offset. We can see easily now that the test doesn't always fail even when the problem is present as we can run the test many times in quick succession using the “Run Test” button.

## 3.3 wxTestRunner

WxTestRunner like QtTestRunner is a GUI test runner for the CppUnit testing framework. QtTestRunner and wxTestRunner have another similarity in that they are both implemented using cross-platform frameworks. In the case of wxTestRunner the framework in question is wxWidgets. Again, like QtTestRunner wxTestRunner is a graphical implementation of the CppUnit::TestRunner class and works with C++ applications not only those using wxWidgets. The software has been tested with Borland C++ Builder 6 and Microsoft Visual C++ 7 both of these are products which run on Microsoft Windows. We can deduce from this that wxTestRunner has not been tested on Unix/Linux. The software has not been updated since July 2004[15].

I had difficulty in demonstrating this software library. The problems I encountered were as follows:

- No makefiles for Unix-like systems
- All of the code is written using multi-byte strings and multi-byte/ANSI versions of wxWidgets are not readily available for my OS.
- The version of wxWidgets required is 2.5.2. This version of wxWidgets was released in May 2004[16] and from experimenting with the source code I could see that it requires old version of GTK+ (1.2) which I cannot install on the OS I'm using.

As a workaround to all of these issues and because I at least wanted to see what wxTestRunner looked like, I decided to create a new project for the wxTestRunner source code in Eclipse. This was simply a case of creating a new project, specifying that the project was a static library and adding all of the .cpp, .h and program icons from the archive downloaded from the wxTestRunner website[15]. I then needed to get my hands on a multi-byte/ANSI version of the wxWidgets library in order to use the wxTestRunner library and I already knew I wouldn't find one in the Debian repositories pre-built. So, I downloaded the oldest stable version of wxWidgets from the official website with the intention of building it from scratch. I extracted the source code to an arbitrarily named folder and completed these steps:

- I ran ./configure to generate a Makefile. At this point I was told that I was building a Unicode version, but I ignored this for the moment because I wanted to see it build successfully first before changing anything).
- I ran make. At this point I got a compile error relating to the current version of glib (which I had) and some of the source code in this old archive. I did a Google search and found a patch (which involved changing some `#include` statements) on the bug tracking system on the official wxWidgets site[17]. I applied this.
- I ran `make` again. This time with success.
- I ran `make install` successfully.

In order to install a non-Unicode build I ran `./configure --disable-unicode` to generate a Makefile for the multi-byte version of wxWidgets. I ran `make` and `make install` successfully. Following this I used `wx-config (`a tool for displaying information and required compiler/linker flags for installed versions of wxWidgets) to determine that the multi-byte build had been installed. I ran `wx-config --list` and received the following output: `Default config is gtk2-ansi-release-2.6`. I ran `wx-config --cppflags` and was provided with the following information:

```
-I/usr/local/lib/wx/include/gtk2-ansi-release-2.6
-I/usr/local/include/wx-2.6 -DGTK_NO_CHECK_CASTS -D__wxGTK__
-D_FILE_OFFSET_BITS=64 -D_LARGE_FILES -DNO_GCC_PRAGMA
```

These are the compiler flags and include folders which were required to build a project with my newly minted version of wxWidgets 2.6. I copied these faithfully into the settings for my wxTestRunner project in Eclipse. I performed a "Project->Build Project" in Eclipse and the library was built successfully. I implemented the following test application to make use of the wxTestRunner library and leverage our String class tests from the previous section by making reference to the official wxWidgets tutorials online:

```
#include <wx/wx.h>
#include <cppunit/ui/wx/WxTestRunner.h>

class MyApp: public wxApp
{
    virtual bool OnInit();
};

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
```

```
{
	static CppUnit::WxTestRunner runner;
	runner.addTest(StringTests::suite());
	runner.run();
	return true;
}
```

In order to get the test application to work I ran `wx-config --libs` and got the following information which specified what libraries were required:

```
-L/usr/local/lib -pthread   -lwx_gtk2_xrc-2.6 -lwx_gtk2_qa-2.6
-lwx_gtk2_html-2.6 -lwx_gtk2_adv-2.6 -lwx_gtk2_core-2.6
-lwx_base_xml-2.6 -lwx_base_net-2.6 -lwx_base-2.6
```

I added the library search path `/usr/local/lib` and the individual library names to the linker configuration in Eclipse for the test application. In order to be able to debug/run the test application I needed to set the `LD_LIBRARY_PATH` environment variable to include the `/usr/local/lib` folder where the shared libraries from my wxWidgets 2.6 build were located. The test application refused to run without this environment variable set correctly. This is what appears on the screen when `OnInit()` returns:

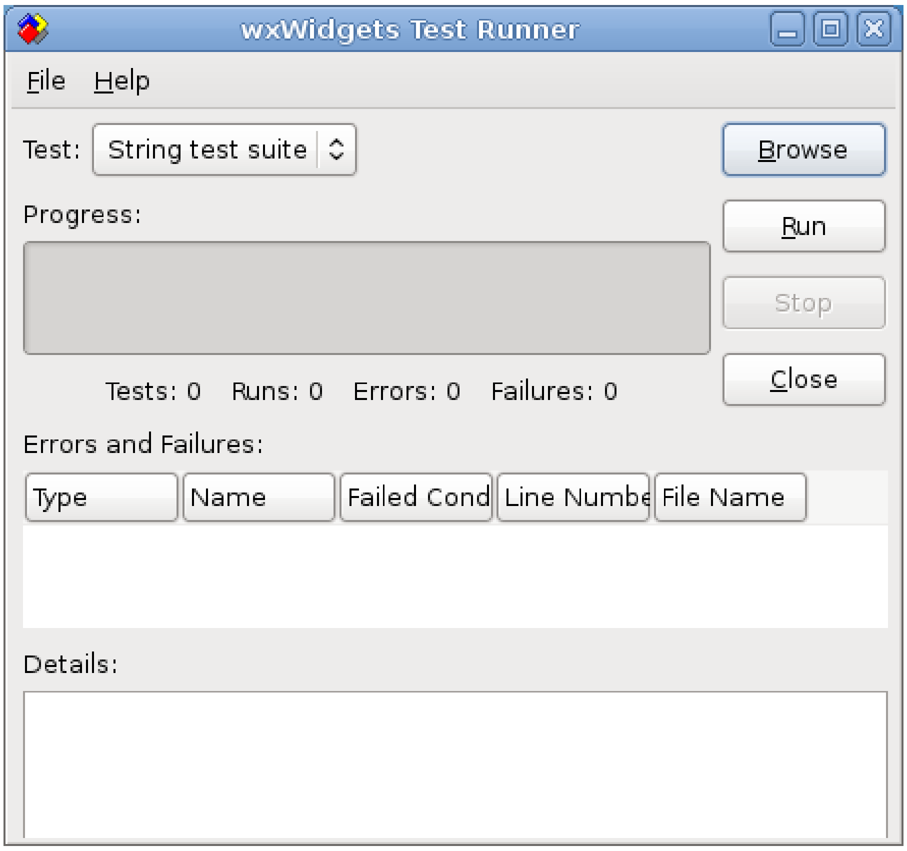
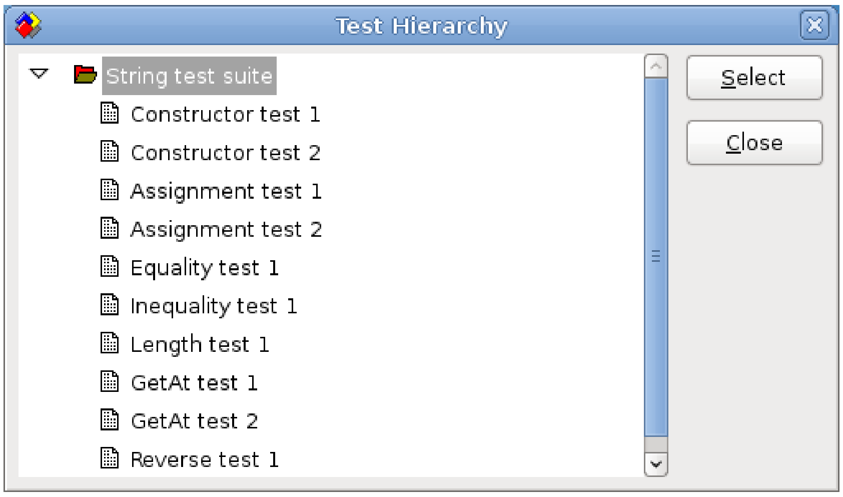


*Illustration 6: Main wxTestRunner dialog*

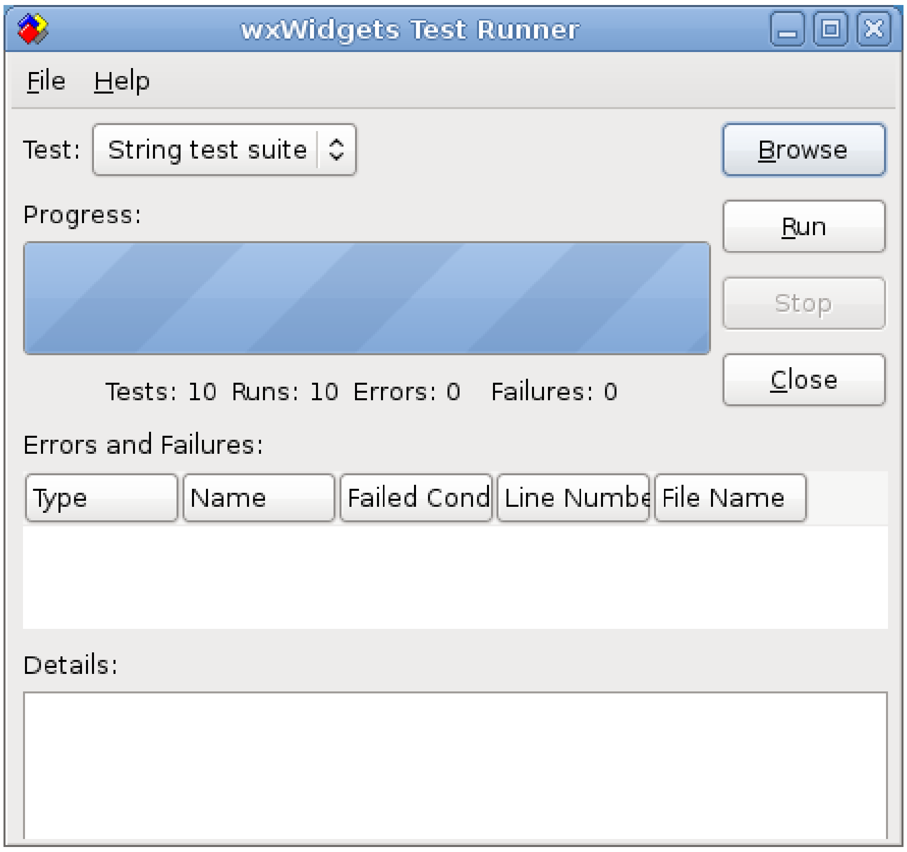
We can see that the interface is much the same as the QtTestRunner interface. By clicking on the “Browse” button we summon up the following dialog:



String test suite
- Constructor test 1
- Constructor test 2
- Assignment test 1
- Assignment test 2
- Equality test 1
- Inequality test 1
- Length test 1
- GetAt test 1
- GetAt test 2
- Reverse test 1

*Illustration 7: wxTestRunner Test Hierarchy dialog*

We can now select a test or test suite from the list and then click “Select” or “Close”. The initial screen then reappears. If we chose a new test it will now be displayed on the initial screen in place of our test suite name. When we click the “Run” button the GUI is updated to the following:

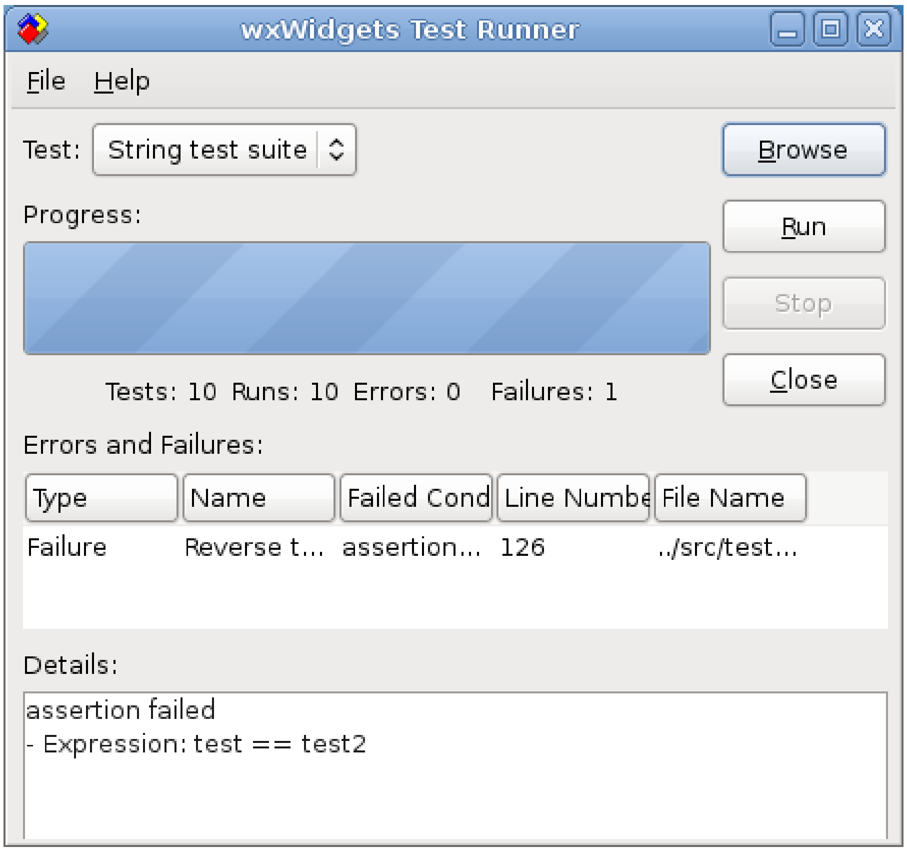


Test: String test suite

Tests: 10 Runs: 10 Errors: 0 Failures: 0

*Illustration 8: Main wxTestRunner dialog updated following a test run*

When we reintroduce our `String::reverse()` error as before with QtTestRunner, rebuild and click the “Run” button again the GUI is updated as follows:



Test: String test suite

Progress:

Tests: 10 Runs: 10 Errors: 0 Failures: 1

Errors and Failures:

| Type | Name | Failed Cond | Line Numbe | File Name |
|---|---|---|---|---|
| Failure | Reverse t... | assertion... | 126 | ../src/test... |

Details:

```
assertion failed
- Expression: test == test2
```

*Illustration 9: Main wxTestRunner dialog updated following a test failure*

I encountered the following issues while running wxTestRunner:

- wxTestRunner should contain the feature of green and red coloured progress bars as visual "hints" for test success and failure. We can see from the screen-shots that this feature is not working.
- I experienced two crashes in wxTestRunner when I ran it initially. These were both caused by the SetDefault() method called on a wxButton control. I simply commented out the two offending calls to SetDefault() to get a chance to see the various screens/dialogs in the application.

I suspect both of these issues were directly related to the fact that the version of GTK, and as a result, version of wxWidgets that I ended up using is newer than the required version specified by the authors[15].

## 3.4 Notes

- Both QtTestRunner and wxTestRunner are merely graphical extensions of the text/console based TestRunner class included with CppUnit. They don't provide any information in addition to what it provides. They have a progress bar but it only updates between tests not during them. We could easily write our own text based progress bar for the CppUnit::TestRunner class.
- An advantage of using a GUI version of the TestRunner class instead of the text based one provided with CppUnit is that it is possible to re-run the tests several times in quick succession by clicking the "Run" or "Run Test" buttons. The ability to re-run tests in quick succession could prove useful for tests which don't fail consistently due to memory related

bugs coupled with the state of memory allocated for the application etc.

## 4. Random test generation

"A random number generator can be used to generate test cases when writing a regression test[18]." A random number generator is a function that returns a random looking number each time it is called. Random number generators are typically implemented using pseudo random number algorithm. Pseudo random number algorithms are entirely dependent on a seed value which they require to generate their output, as a result the outputs from a pseudo random number algorithm is deterministic given a particular seed value[18]. There also exist hardware generators which generate truly random numbers in a non-deterministic sequence. In order to use random numbers to generate test values we can use pseudo random number algorithms and capitalise on their determinism in order to be able to repeat tests[18]. The advantage of doing this rather than choosing values is that initially the test values are random or appear to be. A very simple example of a pseudo random number generator is:

```
class rand
{
     static int seed;
     rand() {
          seed = 123;
     }
     static int gen() {
          seed = seed * 1234;
          return seed;
     }
};
```

This pseudo random number generator starts with an initial seed value of 123. The `gen()` method returns an `int` value which is a pseudo-random number. Each time the `gen()` method is called the seed value is updated to the most recently generated random number which is then used in any subsequent call to `gen()`. There are drawbacks to using random numbers for testing purposes[18]:

- Using random values means that are a lots of values which aren't being tested
- Random values are not smart in terms of white-box testing and boundary value analysis

This project will be dealing with many different data-types other than the `int` data-type used in the code example above. It will be possible, however, to generate random values for all of these by being aware of their maximum and minimum values and generating random values between 0 and 1. For example, for the `unsigned char` data-type, we know that its maximum value is 255 so we can generate a random value for a variable of this data-type by multiplying a random value between 0 and 1 by 255. I have used the example of an unsigned data-type here for simplicity. Data-types which can have both positive and negative values can have random values generated for them in the same way but with additional steps.

## 5. Technology

### 5.1 Linux

Linux is an operating system created by a Finnish student named Linus Torvalds in 1991. The operating system was inspired by his use of Minix[19], a small UNIX system created by Andrew

Tanenbaum for teaching purposes[20]. The difference between it and other operating systems is that its kernel (core part of the operating system) was released using the GNU license making it and its source-code freely available to everyone. Many companies and individuals have since then released their own versions of Linux (Novell and Corel, to provide one successful and one failed example[21 ]). Due to its reliability and adaptability it is regularly found running on internet servers and embedded in devices (routers, modems etc.)[19]. The distribution of Linux which I am using to develop the project is called Debian. The Debian project was set up by Ian Murdock in 1993 and named after himself and his wife Debra[22]. The project is the result of many developers around the world volunteering their spare time. Communication is done using a variety of mailing lists as well as email. Although the project is based on voluntary contributions it does have a carefully organised structure[22]. Other Linux distributions such as Ubuntu are based largely on the Debian distribution[23]. Most software available for Linux can be installed as packages which are pre-built by the Debian developers for installations of the OS running on different architectures. These packages are stored in online and offline (CD/DVD installation media) databases known as repositories. The Debian operating system contains a set of tools called the “Advanced Package Tool (APT)” and a command-line front-end called “aptitude”[24] which can be used to easily search for and install any software located in Debian repositories.

## 5.2 Eclipse

Eclipse is a software development application which consists of an IDE (integrated development environment) and an “extensible plug-in system”[25]. It is written mostly using the Java programming language. The plug-in system allows it to be used to develop in many other languages for example, C, C++ and PHP. The original source-code is based on VisualAge a Smalltalk development environment. Eclipse started life as an IBM project but eventually it was released as open-source software and an organisation called the Eclipse Foundation was set up to manage and continue its development[25]. The default version of Eclipse (“Classic”) allows development using the Java programming language. Plug-ins developed either by the Eclipse Foundation itself or by other companies (as Eclipse is open source) allow the application to be used for development in other languages. CDT is an example of an extended version of Eclipse for C and C++ development which is developed by the Eclipse Foundation. Eclipse CDT (as Eclipse with CDT is known) is the environment which I am using to develop the project. Eclipse CDT uses the GCC (GNU Compiler Collection) by default for compiling and linking C and C++ code however, any compilers and linkers available for the operating system on which Eclipse is being run can be used instead[26]. An example of a commercial product which uses Eclipse is Zend Studio for PHP. This is a PHP development environment from the company who invented the language[27].

## 5.3 GNU Compiler Collection

The GNU Compiler Collection is a suite of compilers for several programming languages including C, C++, Objective C, Objective C++, Java, Fortran and Ada[28]. The project was founded by Richard Stallman, also the founder of the GNU Project which was started in 1984[29]. The goal of the GCC project was to provide a free set of compilers. The abbreviation GCC is used to refer to both the “GNU Compiler Collection” and also the “GNU C Compiler” when the context is compilation of C programs. Most of the compilers have there own names, G++ is the name of the compiler for C++ and GNAT is the name of the compiler for Ada[28]. None of the compilers work by converting source code to C code and then compiling the C code. All of them compile to machine code directly. GCC supports three versions of the C standard, C89, C94 and a version of C99 with corrections[30]. The first version of this optimizing compiler was released  in 1987. GCC supports most of the ISO C++ standard (C++98) with the exception of the export keyword “proposed as a way of separating the interface of templates from their implementation”. Most C++

compilers do not support this keyword. The ability to compile C++ code with the GNU Compiler Collection was added in 1992[29]. A significant feature of the GCC is its portability. It has been implemented for and runs on most platforms available today, including embedded chips.

## Bibliography

[1] Meudec, C (2010). Project Proposals 2010-2011.
[2] Pressman, R. S (2010). Software Engineering: A Practitioner's Approach. The McGraw-Hill Companies, Inc.
[3] Perry, W. E (2000). Effective Methods for Software Testing. John Wiley & Sons, Inc.
[4] Osherove, R (2009). The Art of Unit Testing. Manning Publications Co.
[5] Kent Beck Publications. http://www.servinghistory.com/topics/Kent_Beck::sub::Publications. Accessed: 29/11/2010.
[6] Beck, K (1989). Simple Smalltalk Testing: With Patterns.
[7] Meszaros, G (2007). xUnit Test Patterns. Pearson Education, Inc.
[8] Beck, K & Gamma, E. Test Infected: Programmers Love Writing Tests. http://junit.sourceforge.net/doc/testinfected/testing.htm. Accessed: 15/12/2010.
[9] CppUnit Documentation. http://cppunit.sourceforge.net/doc/lastest/index.html. Accessed: 08/12/2010.
[10] CppUnit Cookbook. http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html. Accessed: 08/12/2010.
[11] CppUnit Source Code. http://downloads.sourceforge.net/cppunit/cppunit-1.12.1.tar.gz. Accessed: 15/12/2010.
[12] Hammell, T. Extreme Java GUI Testing. http://www.developer.com/java/other/article.php/1016841. Accessed: 12/12/2010.
[13] Nokia. Products - Qt. http://qt.nokia.com/products/. Accessed: 14/12/2010.
[14] QtTestRunner - cppunit. http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=QtTestRunner. Accessed: 14/12/2010.
[15] Pang, A & Lepilleur, B. wxTestRunner: a CppUnit (C++) Test Runner for wxWidgets. http://www.softwaredevelopment.ca/wxtestrunner.shtml. Accessed: 14/12/2010.
[16] SourceForge.Net. wxWidgets Browse/wxAll. http://sourceforge.net/projects/wxwindows/files/wxAll. Accessed: 14/12/2010.
[17] wxWidgets Developers. Conflict with glib 2.21 in GSocket. http://trac.wxwidgets.org/ticket/10883. Accessed: 14/12/2010.
[18] Thomas Wang. Random Number Based Test Case Generation. http://www.concentric.net/~ttwang/tech/rndtest.htm. Accessed: 15/12/2010.
[19] Linux.org. What is Linux. http://www.linux.org/info/. Accessed: 15/12/2010.
[20] MINIX. http://en.wikipedia.org/wiki/MINIX. Accessed: 15/12/2010.
[21] Corel Linux. http://en.wikipedia.org/wiki/Corel_Linux. Accessed: 15/12/2010.
[22] About Debian. http://www.debian.org/intro/about. Accessed: 15/12/2010.
[23] About Ubuntu. http://www.ubuntu.com/project/about-ubuntu. Accessed: 15/12/2010.
[24] Apt. http://wiki.debian.org/Apt. Accessed: 16/12/2010.
[25] Eclipse (software). http://en.wikipedia.org/wiki/Eclipse_(software). Accessed: 15/12/2010.
[26] CDT FAQ. http://wiki.eclipse.org/CDT/User/FAQ. Accessed: 15/12/2010.
[27] Zend Studio. http://www.zend.com/products/studio/. Accessed: 15/12/2010.
[28] Programming Languages Supported by GCC. http://gcc.gnu.org/onlinedocs/gcc-4.5.1/gcc/G_002b_002b-and-GCC.html. Accessed: 16/12/2010.
[29] Gough, B. An Introduction to GCC: For the GNU Compilers GCC and G++. http://www.network-theory.co.uk/docs/gccintro/index.html. Accessed: 16/12/2010.
[30] Language Standards Supported by GCC. http://gcc.gnu.org/onlinedocs/gcc-4.5.1/gcc/Standards.html. Accessed: 16/12/2010.

## Illustration Index

## Index of Tables