CS2720 Practical Software Development

CPPUNIT Tutorial Spring 2011

Instructor: Rex Forsyth
Office: C-558
E-mail: forsyth@cs.uleth.ca
Tel: 329-2496

Tutorial Web Page: http://www.cs.uleth.ca/~forsyth/cs2720/lab/lab.html

Testing

- Any library or class
  - ensure that it works according to specifications
- project; as libraries or classes are added
  - ensure that everything works together
  - ensure that any component does not break another
- new code or modifications
  - do not break existing code
  - previous tests still work

Up till now this has been done on an ad hoc basis by writing simple test programs and testing various cases.

Problems

- determine the correct cases
- testing all cases with every test
- remembering which cases need to be tested
- recording results for examination later or by other team members

Tests should be written before any code is written. Just use the **design** document.

Consider a **die** class.

Die Design

Create a die which may have between 4 and 12 (use constants) sides and a value on top. Should be able to determine the value on top the die and the number of sides that the die has. Should also be able to display the value on the top of the die. The class should also provide a way to change the number of sides on the die and to roll the die (ie change the top value).

Behaviours

- Constructors
  - 0 parameters
    - create a 6-sided die with 1 to 6 on top
  - 1 parameter(n)
    - if n is valid, create an n-sided die with 1 to n on top
    - otherwise create a 6-sided die with 1 to 6 on top.
  - 2 parameters(n,t)
    - if n and t are valid, create an n-sided with t on top
    - if n is not valid, create a 6-sided die.
    - if t is not valid, set the top to a valid random value

- Mutators
  - setSides(n)
    - if n is not valid, do nothing
    - if n is valid, change the number of sides to n.
    - if the current top value is between 1 and new number of sides then do nothing,
    - otherwise change the top value to a random value between 1 and the new number of sides.
  - roll()
    - randomly change the top value to a value between 1 and number of sides
- Accessors
  - sides() - return the number of sides that this die has
  - top() - return the value on the top of the die
  - display() - display the value on the top of the die

The die class

- constructors – very specific requirements
- mutators – setsides, roll
- accessors – top, sides, display

What things should we test?

CPPUNIT

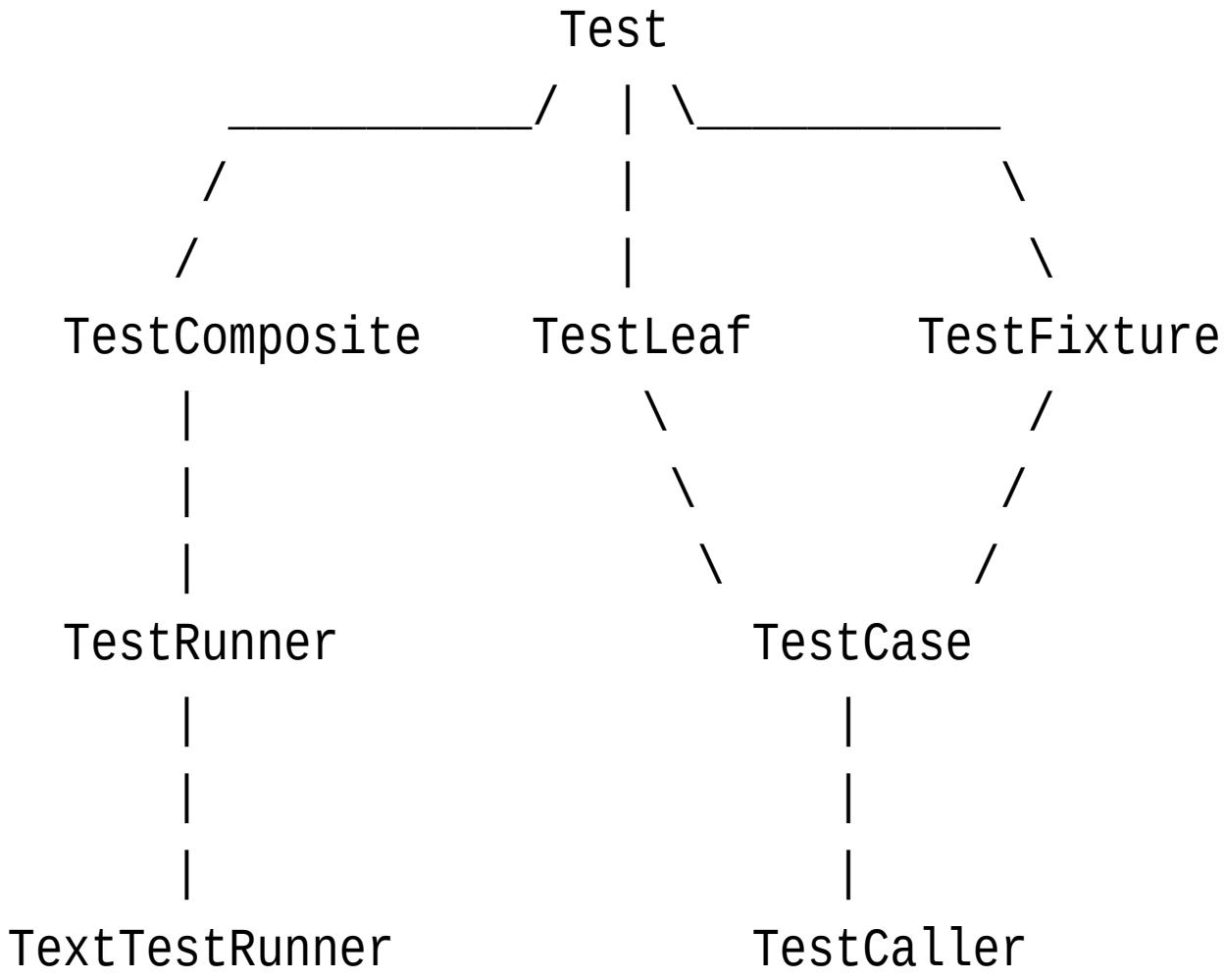
- Provides a way to write tests, add more tests and record results
- A class hierarchy and associated macros
- The base class is called **test**
- It has a *run* method which subclasses should override

To use CPPUNIT

1. Include appropriate header files
2. Create derived classes that inherit from the base classes provided
3. Write a testing program which uses these classes
4. Compile and link these classes
   - `-lcppunit -ldl`
5. run the test program

All the CPPUNIT objects are in the namespace CppUnit

Some of the hierarchy



We will use the **TestFixture** which requires a test suite.

CPPUNIT provides macros to build a test suite and to simplify the process.

The Main program

```
/// \file
/// Main program to run test suites
#include <cppunit/TextTestRunner.h>

#include "dieTester.h"

//////////////////////////////////////////////////////////////////////
/// The simple main function to run the test suites
//////////////////////////////////////////////////////////////////////
int main()
{
   CppUnit::TextTestRunner runner;
   runner.addTest(DieTest::suite());
   runner.run();

   return 0;
}
```

The header file

```
#ifndef DIE_TESTER_H
#define DIE_TESTER_H

#include <cppunit/TestFixture.h>
#include <cppunit/extensions/HelperMacros.h>

#include "die.h"

//**********************************************************************
/// Class to test the die class using a fixture and a test suite using Macros
//**********************************************************************
class DieTest : public CppUnit::TestFixture {

      /// the macros to create the test suite
      CPPUNIT_TEST_SUITE(DieTest);
      CPPUNIT_TEST(testConstructors);
      CPPUNIT_TEST(testSetSides);
      CPPUNIT_TEST(testRoll);
      CPPUNIT_TEST_SUITE_END();
```

```
   public:
      void setUp();             ///< Create variables to be used in testing
      void tearDown();          ///< Clean up any allocated variables
      void testConstructors(); ///< Test the different uses of constructors
      void testSetSides();      ///< Test different uses of \c setSides
      void testRoll();          ///< Test the \c roll method

   private :
      Die* die1;
      Die* die2;
      Die* die3;
      Die* die4;
      Die die5;
      int die2Top;
};

#endif
```

The implementation file

```
#include <cppunit/extensions/HelperMacros.h>
#include "die.h"
#include "dieTester.h"

//**************************************************************************
/// Create a default die, a nine sided die, a six sided die because the
/// parameter is out of range, and an eleven sided die with 8 on top
//**************************************************************************
void DieTest::setUp() {
   die1 = new Die;
   die2 = new Die(9);
   die3 = new Die(3);
   die4= new Die(11,8);
   die2Top = die2->top();
}
//**************************************************************************
/// return memory for all four die
//**************************************************************************
void  DieTest::tearDown() {
   delete die1;
   delete die2;
   delete die3;
   delete die4;
}
```

```
//*************************************************************************
/// Test that a \b default creation has 6 sides and valid value on top; that
/// the creation with \b 9, has nine side and a valid number on top; that the
/// creation with \b 3, in fact has 6 sides with a valid number on top; and
/// that the creation with <b>11,8</b>, has 11 sides with 8 on top.
//*************************************************************************
void  DieTest::testConstructors() {
   CPPUNIT_ASSERT(die1->sides() == 6);
   CPPUNIT_ASSERT(die1->top() >= 1 && die1->top() <= 6);

   CPPUNIT_ASSERT(die2->sides() == 9);
   CPPUNIT_ASSERT(die2->top() >= 1 && die2->top() <= 9);

   CPPUNIT_ASSERT(die3->sides() == 6);
   CPPUNIT_ASSERT(die3->top() >= 1 && die3->top() <= 6);

   CPPUNIT_ASSERT(die4->sides() == 11);
   CPPUNIT_ASSERT(die4->top() == 8);
}
```

```
//*********************************************************************
/// Show that changing 9 sides to 12 changes the sides but not the top;
/// that changing 12 sides to 13 (invalid) leaves the die unchanged;
/// and that changing 11 sides, with 8 on top, to 7 sides
/// leaves 7 sides and also changes the top from 8 to a valid value
//*********************************************************************
void  DieTest::testSetSides() {
   die2->setSides(12);
   CPPUNIT_ASSERT(die2->sides() == 12);
   CPPUNIT_ASSERT(die2->top() ==  die2Top);

   int numSides = die2->sides();
   die2->setSides(13); // invalid number of sides so should be unchanged
   CPPUNIT_ASSERT(die2->sides() == numSides);
   CPPUNIT_ASSERT(die2->top() == die2Top);

   Die die6(11,8);
   die6.setSides(7);
   CPPUNIT_ASSERT(die6.sides() == 7);
   CPPUNIT_ASSERT(die6.top() != 8);
   CPPUNIT_ASSERT(die6.top() >= 1 && die6.top() <= 7);
}
```

```
//*********************************************************************
/// show that rolling a six sided die always leaves a value between 1 and
/// 6 inclusive on the top.
//*********************************************************************
void  DieTest::testRoll(){
   for (int i = 0; i < 100000; i++) {
      die5.roll();
      CPPUNIT_ASSERT(die5.top() >= 1 && die5.top() <= 6);
   }
}
```

The Makefile

```
CC = g++
CCFLAGS = -Wall
OBJS =  tester.o die.o dieTester.o utility.o
macro : $(OBJS)
        $(CC) $^ -o $@ -lcppunit -ldl

# default rule for compiling .cc to .o
%.o: %.cc
        $(CC) -c $(CCFLAGS) $<

## generate the prerequistes and append to the desired file
.prereq : $(OBJS:.o=.cc) $(wildcard *.h) Makefile
        rm -f .prereq
        $(CC) $(CCFLAGS) -MM $(OBJS:.o=.cc) >> ./.prereq

## include the generated prerequisite file
include .prereq

.PHONY : clean
clean:
        rm -f *.o *~ *% *# .#*

.PHONY : clean-all
clean-all: clean
         rm -f macro
```

Ideally the test suite is coded first based on the design.

As you develop functions, classes etc., you run the tester. Of course, cases which have not yet been implemented fail, but this lets you know what has yet to be implemented.

Ideally, the person writing the tester is **NOT** the same person implementing the function or class.

- You may only need **one** test program.
- Test writers create a new test class for each function or class to be tested, including the suite.
- Now add the test suite to the test program
- And then adjust the Makefile to include the new class (add .o to OBJS)

```
/// Main program to run test suites
#include <cppunit/TextTestRunner.h>

#include "integerTest.h"
#include "powerTest.h"
#include "dieTester.h"
#include "timeInputTest.h"
#include "timeOutputTest.h"
#include "timeTester.h"

/// The simple main function to run the test suites
int main()
{
   CppUnit::TextTestRunner runner;
   runner.addTest(DieTest::suite());
   runner.addTest(IntegerTest::suite());
   runner.addTest(PowerTest::suite());
   runner.addTest(TimeTest::suite());
   runner.addTest(TimeTestIn::suite());
   runner.addTest(TimeTestOut::suite());
   runner.run();
   return 0;
}
```

```
CC = g++
CCFLAGS = -Wall
OBJS= tester.o integer.o die.o dieTester.o utility.o time1.o powerTest.o \
      integerTest.o timeTester.o timeInputTest.o timeOutputTest.o
ptest : $(OBJS)
        $(CC) y$^ -o $@ -lcppunit -ldl

%.o : %.cc
        $(CC) $(CCFLAGS) -c $<

## generate the prerequistes and append to the desired file
.prereq : $(OBJS:.o=.cc) $(wildcard *.h) Makefile
        rm -f .prereq
        $(CC) $(CCFLAGS) -MM $(OBJS:.o=.cc) >> ./.prereq

## include the generated prerequisite file
include .prereq

.PHONY : clean
clean :
        rm -f *.o *~ #* .#*

.PHONY : clean-all
clean-all : clean
        rm -f ptest
```