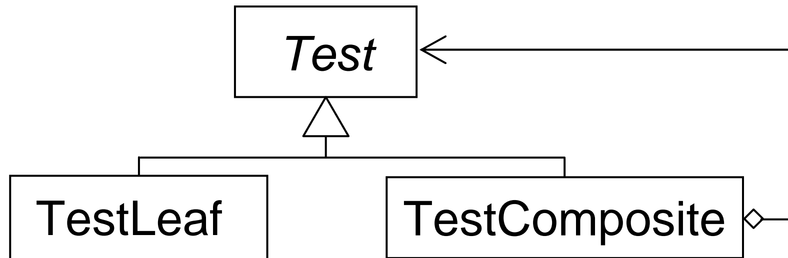
Basic Design of CppUnit

C++ Object Oriented Programming

Pei-yih Ting
NTOUCS

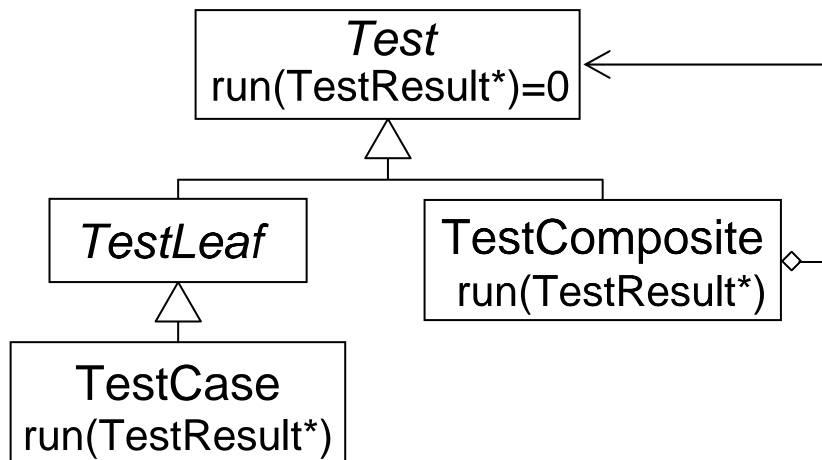
Structurally, this is a **Composite** pattern.



Each **TestCase** is a single test.
Each **TestComposite** contains multiple tests.
This defines a tree of multiple tests as the leave nodes.

http://en.wikipedia.org/wiki/Composite_pattern

Functionally, it acts more like an **Interpreter** pattern.

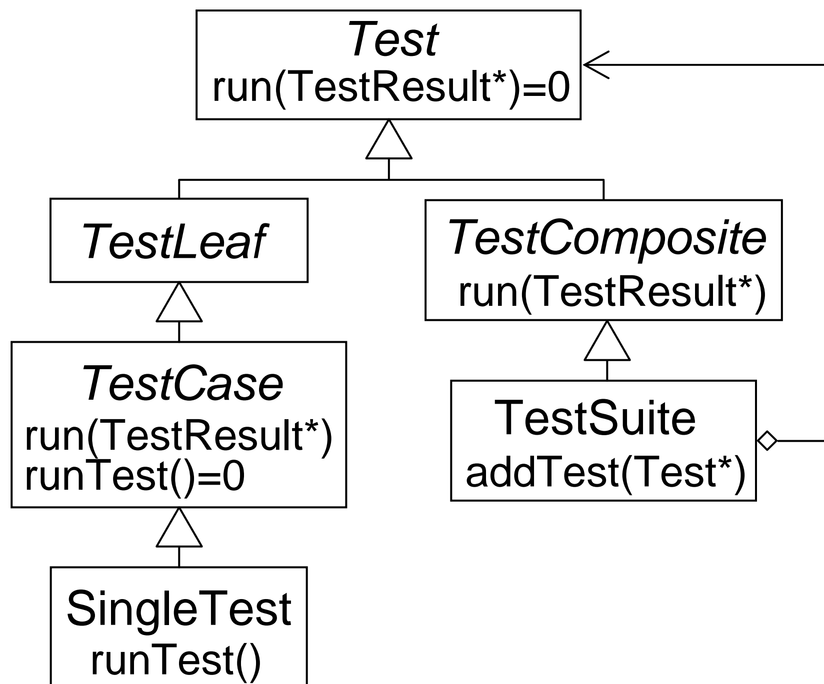


You can send the root **TestComposite** object a “run” message, it then forwards the message recursively down to each leaf of the hierarchy sequentially.

http://en.wikipedia.org/wiki/Interpreter_pattern

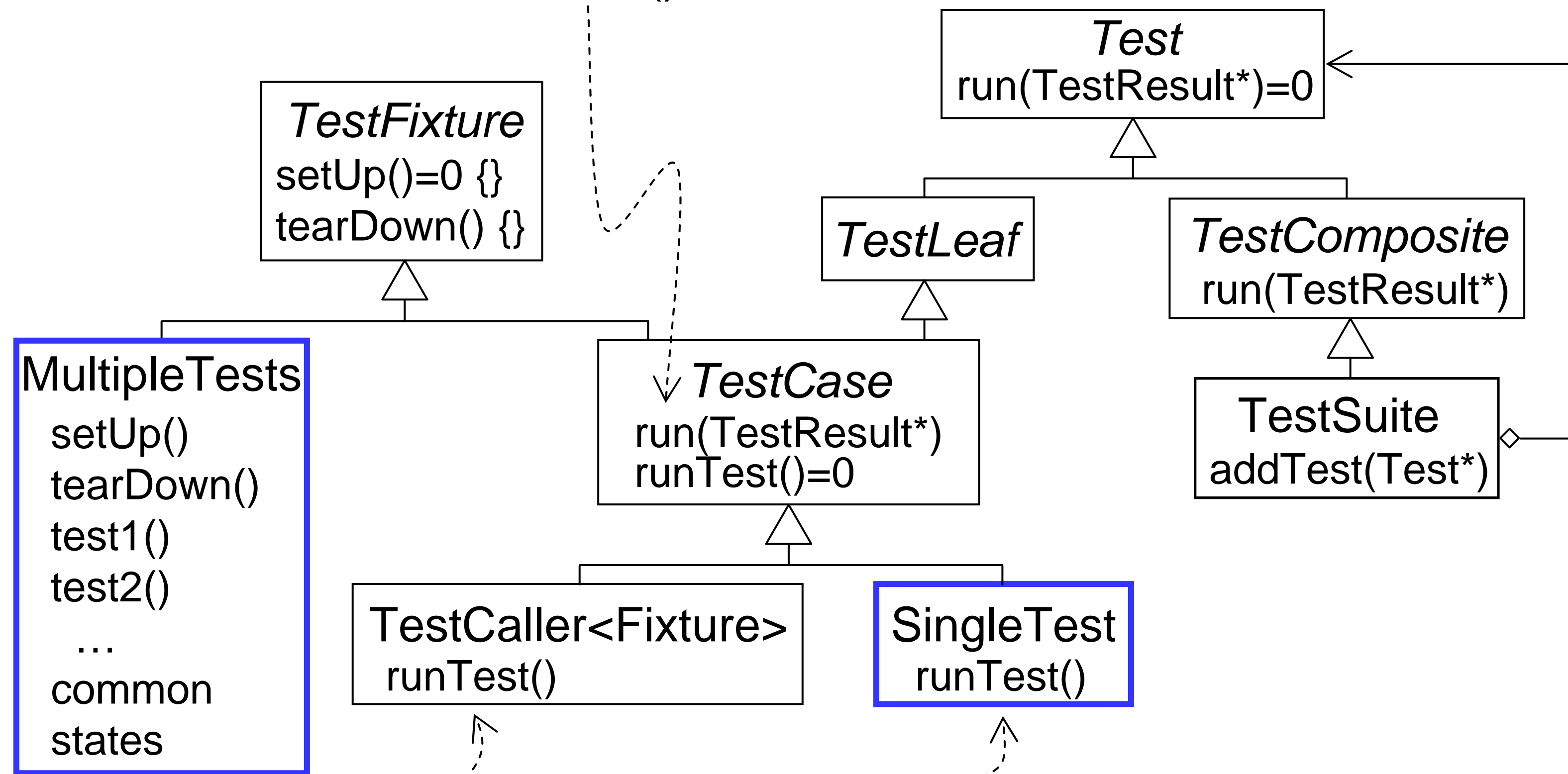
We would like each class to focus on a certain part of functionalities in this design. Thus,

1. defer implementation of the actual test as runTest() to the subclass of TestCase, i.e. SingleTest.
2. defer the implementation of detail management function addTest(Test *) and the hierarchical container to the subclass of TestComposite, i.e. TestSuite.



However, each **SingleTest** classes of related tests might have similar context setup. We would like to combine several related tests into a single class called **TestFixture.**

TestCase::run(TestResult*)
wraps runTest() with Fixture::setUp()
and Fixture::tearDown().



SingleTest::runTest()
implements a single test

TestCaller<Fixture>::runTest()
delegates the “run” message
to a single Fixture::test#()
This template is used to extract each
MultipleTests::test#() as a single test.

As a result, you code your Unit Tests with the following procedure

1. Collect related tests into a MultipleTests class with suitable setUp(), tearDown(), member variables, and independent test1(), test2(), …
2. Use TestCaller<Fixture> template to extract each test#() member function from your MultipleTests class as a single test case.
3. Put solitaire test into a SingleTest class with default implementation of setUp() and tearDown() empty and the test body in runTest()
4. Instantiate a TestSuite object, add the above test case objects or other TestSuite objects to it such that a tree of test cases can be formed.
5. Instantiate a TestResult object *controller* to log the running outputs
6. Send the root TestSuite object of the tree a run(TestResult*) message such that every leaves test case will be executed in order and the results logged into *controller*.
7. If you instantiate a TestResultCollector object *result*, you can use *controller*.addListener(&*result*) to collect all the test results.
8. You can then instantiate a CompilerOutputter *outputter*(&result, CPPUNIT_NS::stdCOut()) to bind with the TestResultCollector
9. Use outputter.write(); to output to the screen.

```
int main( int argc, char* argv[] ) {
  // Create the event manager and test controller
  CPPUNIT_NS::TestResult controller;

  // Add a listener that colllects test result
  CPPUNIT_NS::TestResultCollector result;
  controller.addListener( &result );

  // Add a listener that print dots as test run.
  CPPUNIT_NS::BriefTestProgressListener progress;
  controller.addListener( &progress );

  // Add the top suite to the test runner
  CPPUNIT_NS::TestRunner runner;  // use MFCTestRunner to get GUI
  runner.addTest( CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest() );
  runner.run( controller ); // simply runner.run() would output to the screen

  // Print test in a compiler compatible format.
  CPPUNIT_NS::CompilerOutputter outputter( &result, CPPUNIT_NS::stdCOut() );
  outputter.write();

  return result.wasSuccessful() ? 0 : 1;
}
```

The above procedures can be simplified a lot with the Macros defined in **cppunit/extensions/HelperMacros.h**

ComplexUnitTest.h

```
#include <cppunit/extensions/HelperMacros.h>
class ComplexUnitTest : public CPPUNIT_NS::TestFixture {
    CPPUNIT_TEST_SUITE( ComplexUnitTest );
    CPPUNIT_TEST( testAdd );
    CPPUNIT_TEST( testDivide );
    CPPUNIT_TEST_SUITE_END();
public:
  void setUp(); // will be executed before each test
  void tearDown(); // will be executed after each test
  void testAdd();
  void testDivide();
private:
  …
};
```

ComplexUnitTest.cpp

```
#include <cppunit/config/SourcePrefix.h>
CPPUNIT_TEST_SUITE_REGISTRATION( ComplexUnitTest );
…
```

```
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TestRunner.h>
    ...
void main() {
  CPPUNIT_NS::Test *suite =
    CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest();
  CppUnit::TextUi::TestRunner runner;
  runner.addTest(suite);
  bool wasSucessful = runner.run();
}
```

專案屬性:

a. '專案 > lab31 屬性頁 > 組態屬性 - C/C++ > 程式碼產生 > 執行階段程式庫 > 多執行緒偵錯 DLL/MDd'

b. '專案 > lab31 屬性頁 > 組態屬性 - 連結器 > 輸入'.
'其他相依性' 填入 'cppunit.lib'

c. '專案 > lab31 屬性頁 > 組態屬性 - 建置事件 > 建置後事件'.
'命令列' 填入 '$(TargetPath)'
'描述' 填入 'Unit Tests...'

進階: 你可以嘗試把被測試的程式和 unit test 分成同一個方案裡的兩個專案