Object-Oriented Programming
JUnit Cook’s Tour

CSIE Department, NTUT
Woei-Kae Chen
JUnit: The Goals of JUnit

- To write a framework within which we have some glimmer of hope that developers will actually write tests. The framework has to use familiar tools, so there is little new to learn.

- The second goal of testing is creating tests that retain their value over time. Someone other than the original author has to be able to execute the tests and interpret the results.

- Creating a setup or fixture is expensive and a framework has to enable reusing fixtures to run different tests.
JUnit: The design of JUnit

- The design of JUnit will be presented in a style called **Patterns Generate Architectures**.
  - The idea is to explain the design of a system by starting with nothing and **applying patterns**, one after another, until you have the architecture of the system.
  - We will present the architectural problem to be solved, summarize the pattern that solves it, and then show how the pattern was applied to JUnit.
JUnit: Getting started - TestCase

- Developers often have tests cases in mind, but they realize them in many different ways:
  - print statements, debugger expressions, test scripts.

- If we want to make manipulating tests easy, we have to make them objects.
  - This takes a test that was only implicit in the developer’s mind and makes it concrete.

- The Command pattern fits our needs quite nicely.
  - Encapsulate a request as an object, thereby letting you...
    - queue or log requests...
  - run (execute); decouple invoker and receiver.
JUnit: Getting started - TestCase

public abstract class \texttt{TestCase} implements Test {
    private final String fName;

    public \texttt{TestCase}(String name) {
        fName = name;
    }

    public abstract void \texttt{run}();
    ...
}
JUnit: Blanks to fit in - run()

- The next problem to solve is giving the developer a convenient “place” to put their fixture code and their test code.

- Fortunately, there is a common structure to all tests - they set up a test fixture, run some code against the fixture, check some results, and then clean up the fixture.

- Template Method addresses our problem quite nicely.
JUnit: Blanks to fit in - run()

Here is the template method:

```java
public void run() {
    setUp();
    runTest();
    tearDown();
}
```

The default implementations of these methods do nothing:

```java
protected void runTest() {
}

protected void setUp() {
}

protected void tearDown() {
}
```

Implementation in abstract class
JUnit: Reporting results - TestResult

- If a TestCase runs in a forest, does anyone care about the result? Sure- you run tests to make sure they run. After the test has run, you want a record, a summary of what did and did not work.

- The Smalltalk Best Practice Patterns has a pattern that is applicable. It is called Collecting Parameter.

- It suggests that when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you.
public class TestResult extends Object {
    protected int fRunTests;
    protected Vector fErrors, fFailures;
    public TestResult() {
        fRunTests = 0;
        fErrors = new Vector();
        fFailures = new Vector();
    }
    public synchronized void startTest(Test test) {
        fRunTests++;
    }
    public synchronized void addError(Test test, Throwable t) {
        fErrors.addElement(new TestFailure(test, t));
    }
    public synchronized void addFailure(Test test, Throwable t) {
        fFailures.addElement(new TestFailure(test, t));
    }
}
JUnit: Reporting results - TestResult

```java
public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
        runTest();
    } catch (AssertionFailedError e) { // Failure
        result.addFailure(this, e);
    } catch (Throwable e) { // Error
        result.addError(this, e);
    }
    finally {
        tearDown();
    }
}
```
JUnit: Reporting results - TestResult

- An AssertionError is triggered by the assert method provided by TestCase.
- JUnit provides a set of assert methods for different purposes:
  - `assert(boolean condition)`
    ```java
    protected void assert(boolean condition) {
        if (!condition)
            throw new AssertionFailedError();
    }
    ```
  - `assertEquals(X,Y)`
  - `assertEqual(Iterable actual, Iterable expected)`
JUnit: No stupid subclasses – Test Case Again

- A given test case class may implement many different methods, each defining a single test case.
  - Each test case has a descriptive name like testMoneyEquals or testMoneyAdd.
  - The test cases don’t conform to a simple command interface.
- Therefore our next problem is make all the test cases look the same from the point of view of the invoker of the test.
To reuse fixture: subclass TestCase?

1. Separate fixture and test cases.
2. Implement runTest() for every test case.
3. Disadvantage: too many classes.

Doesn't work: setUp() & tearDown() called only once.
JUnit: No stupid subclasses – TestCase Again

- The Adapter pattern springs to mind.
  - Convert the interface of a class into another interface clients expect.

- Different ways of adapters
  - class adapter: uses subclassing to adapt the interface.

1. Class Adapter
2. Too many classes
JUnit: No stupid subclasses – TestCase Again

Use class adapter

```java
public class TestMoneyEquals extends MoneyTest {
    public TestMoneyEquals() { super("testMoneyEquals"); }
    protected void runTest() { testMoneyEquals(); }
}

or

TestCase test = new MoneyTest("testMoneyEquals ") {
    protected void runTest() { testMoneyEquals(); }
};
```

Java anonymous inner classes
JUnit: No stupid subclasses – TestCase Again

- Smalltalk Best Practice Patterns describes another solution for the problem of different instances behaving differently under the common heading of pluggable behavior.
  - The idea is to use a single class which can be parameterized to perform different logic without requiring subclassing.

- The simplest form of pluggable behavior is the Pluggable Selector. Pluggable Selector stores a Smalltalk method selector in an instance variable.
JUnit: No stupid subclasses – TestCase Again

- The Java **reflection** API allows us to invoke a method from a string representing the method’s name. We can use this feature to implement a pluggable selector in Java.
  - We usually don’t use reflection in ordinary applications. In this case, we are dealing with an infrastructure framework and it is therefore OK to wear the reflection hat.

- **CppUnit** (C++ does not have reflection API)
  
  ```cpp
  CPPUNIT_TEST_SUITE(classNameTest)
  CPPUNIT_TEST( testXXX );
  CPPUNIT_TEST( testYYY );
  CPPUNIT_TEST_SUITE_END();
  ```

A pointer to member function
JUnit: No stupid subclasses – TestCase Again

- JUnit offers both:
  - pluggable selector: as default implementation of runTest method.
  - anonymous adapter class.

protected void runTest() throws Throwable {
    Method runMethod = null;
    try {
        runMethod = getClass().getMethod(fName, new Class[0]);
    } catch (NoSuchMethodException e) {
        assert("Method " + fName + " not found", false);
    }
    try {
        runMethod.invoke(this, new Class[0]);
    } catch (InvocationTargetException and IllegalAccessException) {
    }
}
JUnit: Don’t care about one or many - TestSuite

- JUnit can now run a single test case. Our next challenge is to extend it so that it can run many different tests.
- This problem can be solved easily when the invoker of the tests doesn’t have to care about whether it runs one or many test cases.
- A popular pattern to pull out in such a situation is Composite pattern.
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
JUnit: Don’t care about one or many - TestSuite

- Participants of Composite pattern:
  - Component: declares the interface we want to use to interact with our tests. (Test)
  - Composite: implements this interface and maintains a collection of tests. (TestSuite)
  - Leaf: represents a test case in a composition that conforms to the Component interface. (TestCase)
public interface Test {
    public abstract void run(TestResult result);
}

public class TestSuite implements Test {
    private Vector fTests = new Vector();

    public void run(TestResult result) {
        for (Enumeration e = fTests.elements();
            e.hasMoreElements(); ) {
            Test test = (Test)e.nextElement();
            test.run(result);
    }
    
    public void addTest(Test test) {
        fTests.addElement(test);
    }
}

public static Test suite() {
    return new TestSuite(MoneyTest.class);
}

public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
JUnit: Don’t care about one or many - TestSuite
JUnit: The Final Product
JUnit: JUnit Pattern Story Board

Patterns Generate Architectures.

- Command
- Template Method
- Collecting Parameter
- Adapter
- Pluggable Selector
- Composite
JUnit: Conclusion

- Patterns Generate Architectures
  - Try the same style of presentation for your own system.

- Pattern density
  - There is a high pattern “density” around TestCase.
  - Designs with high pattern density are easier to use, but harder to change.
  - Mature frameworks → high pattern density.

- Eat your own dog food
  - The most challenging application of JUnit was testing its own behavior.

- Intersection, not union
  - JUnit implements only those features absolutely essential to running tests.
  - Extension: a TestDecorator allowing execution of additional code before and after a test.

- Framework writers read their code
  - JUnit authors spent far more time reading the JUnit code than writing it.
  - JUnit authors spent nearly as much time removing duplicate functionality as adding new functionality.