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
  - a framework has to enable reusing fixtures to run different tests.
JUnit: The design of JUnit

- **Patterns Generate Architectures**
  - Used to present the design of JUnit.
  - 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.
  - Presentation flow
    1. Present the architectural problem to be solved
    2. Summarize the pattern that solves it
    3. 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

```java
public abstract class TestCase implements Test {
    private final String fName;

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

    public abstract void run();
}
```

Explained later

Command pattern

Abstract class

JUnit: Blanks to fit in - run()

- The next problem to solve:
  - Give the developer a convenient “place” to put their fixture code and their test code.
  - There is a common structure to all tests
    - they set up a test fixture
    - run some code against the fixture
    - check some results
    - clean up the fixture
- Template Method addresses our problem quite nicely.
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses…
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

- **How to collect test results?**
  - 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 **Collecting Parameter** pattern is applicable.
  - The Smalltalk Best Practice Patterns.
  - 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));
    }
}

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

- **AssertionFailedError**
  - triggered by the assert method provided by TestCase.
  - JUnit provides a set of assert methods
  - `assert(boolean condition)`
    ```java
    protected void assert(boolean condition) {
        if (!condition)
            throw new AssertionFailedError();
    }
    ```
  - `assertEquals(X,Y)`

JUnit: No stupid subclasses – TestCase 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.
JUnit: No stupid subclasses – Test Case Again

- To reuse fixture: subclass Test Case?

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

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

Fixture

Interface

Don't work: setUp() & tearDown() called only once

13
JUnit: No stupid subclasses – TestCase Again

Use class adapter

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

or

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

Smalltalk Best Practice Patterns describes another solution 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

- **Implement a pluggable selector in Java**
  - Use the Java reflection API.
  - Invoke a method from a string representing the method’s name.
  - 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
  - Use Macros instead
    ```cpp
    CPPUNIT_TEST_SUITE(classNameTest)
    CPPUNIT_TEST(testXXX);
    CPPUNIT_TEST(testYYY);
    CPPUNIT_TEST_SUITE_END();
    ```

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

```java
protected void runTest() throws Throwable {
    Method runMethod = null;
    try {
        runMethod = getClass().getDeclaredMethod(name, new Class[0]);
    } catch (NoSuchMethodException e) {
        throw new RuntimeException("Method " + name + " not found", e);
    }
    try {
        runMethod.invoke(this, new Object[0]);
    } catch (InvocationTargetException e) {
        throw new RuntimeException("Method " + name + " not found", e.getCause());
    } catch (IllegalAccessException e) {
        throw new RuntimeException("Method " + name + " not found", e);
    }
}
```
JUnit: No stupid subclasses – TestCase Again

- 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.

JUnit: Don’t care about one or many - TestSuite

- 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)
JUnit: Don’t care about one or many - TestSuite

```java
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);
    }
}
```

JUnit: Don’t care about one or many - TestSuite

Here is an example of creating a TestSuite:

```java
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: Don’t care about one or many - TestSuite

Using pluggable selector

Class: all testXXX()
JUnit: The Final Product

Patterns Generate Architectures
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.
  - High pattern density → easier to use; 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 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.