JUnit – Overview

Summarized by Regev Azriel.

[This summary was taken from:
junit.sourceforge.net/doc/cookstour/cookstour.htm]

1.Goals

What are the goals of JUnit?

Start point : If a program feature lacks an automated test, we assume it doesn't work.

From this perspective, developers aren't done when they write and debug the code, they must also write tests that demonstrate that the program works.

So, our goal are:

- 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. It has to require no more work than absolutely necessary to write a new test. It has to eliminate duplicated effort.

- 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. It should be possible to combine tests from various authors and run them together without fear of interference.

- it has to be possible to leverage existing tests to create new ones. Creating a setup or fixture is expensive and a framework has to enable reusing fixtures to run different tests.

2.The Design of JUnit

- *Design* – 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, supporting our goal of creating tests that retain their value over time. At the same time, object developers are **used to**, well, developing with objects, so the decision to make tests into objects **supports our goal** of making test writing more inviting (or at least less imposing).

- *Run* – The next problem to solve is giving the developer a **convenient "place"** to put their fixture code and their test code. The declaration of TestCase as abstract says that the developer is expected to **reuse** TestCase by subclassing. However, if all we could do was provide a superclass with one variable and no behavior, we wouldn't be doing much to satisfy our first goal, making tests easier to write.

  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. This means that each test will run with a fresh fixture and the results of one test can't influence the result of another. This supports **the goal of maximizing the value of the tests**.

- *Results* – 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 didn't work.

  If tests had equal chances of succeeding or failing, or if we only ever ran one test, we could just set a flag in the TestCase object and go look at the flag when the test completed. However, tests are (intended to be) highly asymmetric- they usually work. Therefore, we only want to **record the failures** and a highly condensed summary of the successes.

  If tests always ran correctly, then we wouldn't have to write them. **Tests are interesting when they fail**, **especially if we didn't expect them to fail**. What's more, tests can fail in ways that we expect, for example by computing an incorrect result, or they can fail in more spectacular ways, for example by writing outside the bounds of an array. No matter how the test fails we want to execute the following tests.

  JUnit distinguishes between *failures* **and** *errors*. The possibility of a failure is anticipated and checked for with assertions. Errors are unanticipated problems like an ArrayIndexOutOfBoundsException. Failures are signaled with an AssertionFailedError error.

- *TestCase as an interface implementation* – **We need an interface to generically run our tests**. However, all test cases are implemented as different methods in the same class. This avoids the unnecessary proliferation of classes. A given test case class may implement many different methods, each defining a single test case. The test cases don't conform to a simple command interface. Different instances of the same Command class need to be invoked with different methods. Therefore our next problem is **make all the test cases look the same** from the point of view of the **invoker** of the test.

  Reviewing the problems addressed by available design patterns, the **Adapter pattern** springs to mind. Adapter has the following intent "Convert the interface of a class into another interface clients expect". This sounds like a good match. Adapter tells us different ways to do this. One of them is a class adapter, which uses subclassing to adapt the interface.

- *Multiple running of TestCases* – To get confidence in the state of a system we need to run many tests. Up to this point JUnit can run a single test case and report the result in a TestResult. **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**. To quote its intent "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." The point about part-whole hierarchies is of interest here. We want to support suites of suites of suites of tests.

  Composite introduces the following participants:

  - Component: declares the interface we want to use to interact with our tests.
  - Composite: implements this interface and maintains a collection of tests.
  - Leaf: represents a test case in a composition that conforms to the Component interface.

  The pattern tells us to introduce an abstract class which defines the common interface for single and composite objects. The primary purpose of the class is to define an interface. When applying Composite in Java we prefer to define an interface and not an abstract class. Using an interface avoids committing JUnit to a specific base class for tests. All that is required is that the tests conform to this interface.