JUnit A Cook's Tour

Kent Beck and Erich Gamma
JUnit advice

- "Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead." Martin Fowler
- At first create a new fixtures all the time
- Soon, reusing your library of fixtures and new tests as simple as adding a method to an existing TestCase subclass.
You can always write more tests!

However, you will quickly find that only a fraction of the tests you can imagine are actually useful. What you want is to write tests that fail even though you think they should work, or tests that succeed even though you think they should fail. Another way to think of it is in cost/benefit terms. You want to write tests that will pay you back with information.
A reasonable return on your testing investment:

- During Development- When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.

- During Debugging- When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.
One word of caution about your tests.

- Once you get them running, make sure they stay running.
- There is a huge difference between having your suite running and having it broken.
- Ideally, you would run every test in your suite every time you change a method. Practically, your suite will soon grow too large to run all the time.
- Try to optimize your setup code so you can run all the tests.
- Or, at the very least, create special suites that contain all the tests that might possibly be affected by your current development.
- Then, run the suite every time you compile. And make sure you run every test at least once a day: overnight, during lunch, during one of those long meetings....
JUnit Cook’s Tour

The Patterns
Getting started - TestCase

- If we want to make manipulating tests easy, we have to make them objects.
- The Command pattern fits our needs quite nicely. Quoting from the intent, "Encapsulate a request as an object, thereby letting you... queue or log requests..." Command tells us to create an object for an operation and give it a method "execute".
Notation – run a TestCase

```
TestCase
run()
```

Command
The next problem to solve is giving the developer a convenient "place" to put their fixture code and their test code.

Template Method addresses our problem quite nicely. Quoting from the intent, "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure."
TestCase.run() applies Template Method
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. We create a new object, `TestResult`, to collect the results of running tests.
TestResult applies Collecting Parameter
No stupid subclasses - TestCase again

- 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".

- The simplest form of pluggable behavior is the **Pluggable Selector**.

- The Java reflection API allows us to invoke a method from a string representing the method’s name.
**TestCase**

- run(TestResult)
- runTest()
- setUp()
- tearDown()

**fName**

- Pluggable Selector

- Adapter (Class)

- runTest()
Don’t care about one or many - TestSuite

- 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."
Summary

Notice how TestCase, the central abstraction in the framework, is involved in four patterns. Pictures of mature object designs show this same "pattern density". The star of the design has a rich set of relationships with the supporting players.
Here is another way of looking at all of the patterns in JUnit. In this storyboard you see an abstract representation of the effect of each of the patterns in turn. So, the Command pattern creates the TestCase class, the Template Method pattern creates the run method, and so on.

One point to notice about the storyboard is how the complexity of the picture jumps when we apply Composite. This is pictorial corroboration for our intuition that Composite is a powerful pattern, but that it "complicates the picture." It should therefore be used with caution.
JUnit Pattern Storyboard
Conclusion

- **Patterns**
  We found discussing the design in terms of patterns to be invaluable, both as we were developing the framework and as we try to explain it to others.

- **Pattern density**
  There is a high pattern "density" around TestCase, which is the key abstraction of JUnit. Designs with high pattern density are easier to use but harder to change.
- **Eat your own dog food**
  As soon as we had the base unit testing functionality implemented, we applied it ourselves.

- **Intersection, not union**
  There is a temptation in framework development to include every feature you can. However, the fewer features the framework has, the easier it is to learn, the more likely a developer will use it. JUnit is written in this style. It implements only those features absolutely essential to running tests—running suites of tests, isolating the execution of tests from each other, and running tests automatically.
Framework writers read their code
We spent far more time reading the JUnit code than we spent writing it, and nearly as much time removing duplicate functionality as we spent adding new functionality. We experimented aggressively with the design, adding new classes and moving responsibility around in as many different ways as we could imagine. We were rewarded (and are still being rewarded) for our monomania by a continuous flow of insights into JUnit, testing, object design, framework development, and opportunities for further articles.