

JUnit A Cook's Tour

: JUnit 3.8.x .

1.

```

    (Test Infected : Programmers Love Writing Tests, Java Report, July
    1998, Volume 3, Number 7 )
    가
    가
    JUnit
    , 가 가
    가
    ( , )
    가
  
```

2.

```

JUnit ?
, 가 가 가
가 가 가
가 , , 가
가 ? ,
가 가
가
1 5
가 가
  
```



```

tearDown ();
}

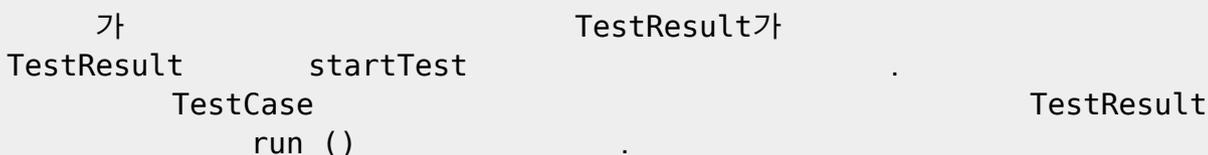
```

TestResult

```

startTest ( ) {
fRunTests ++;
}

```



```

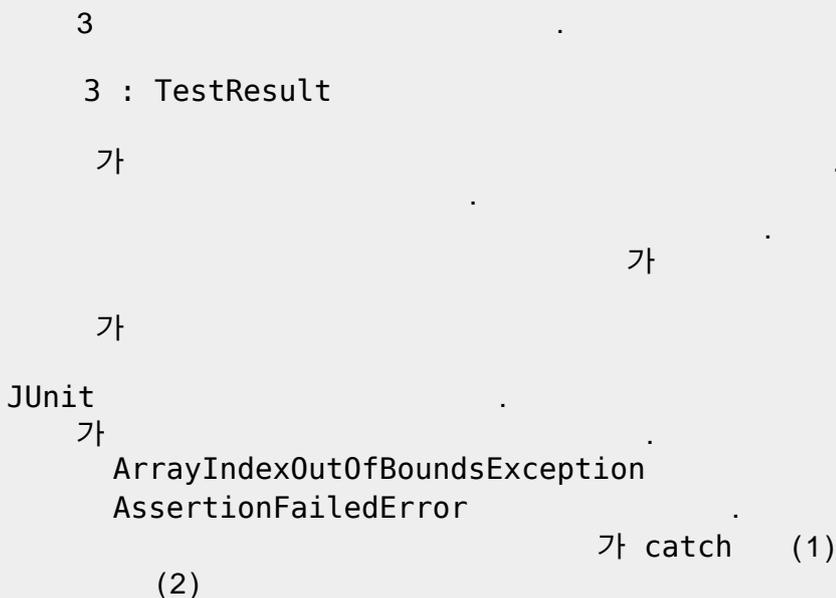
public TestResult run () {
    TestResult result = createResult ();
    result.startTest ();
    return result;
}

```

```

protected TestResult createResult () {
    return new TestResult ();
}

```



```

public void run (TestResult result) {
    result.startTest (this);
    try {
        runTest ();
    }
    catch (AssertionFailedError e) { // 1
        result.addFailure (this, e);
    }
    catch (Throwable e) { // 2

```

```

        result.addError (this, e);
    }
    {
        tearDown ();
    }
}

```

AssertionFailedError TestCase assert
 JUnit assert
 가 .

```

protected void assertTrue (boolean condition) {
    if (! condition)
        throw new AssertionFailedError ();
}

```

AssertionFailedError (TestCase)가
 TestCase.run () AssertionFailedError .

```

AssertionFailedError {
public AssertionFailedError () {}
}

```

. TestResult .

```

addError ( , Throwable t) {
fErrors.addElement (new TestFailure (test, t));
}

```

```

addFailure ( , Throwable t) {
fFailures.addElement (new TestFailure (test, t));
}

```

TestFailure

```

TestFailure Object {
protected Test fFailedTest
    Throwable fThrownException;
}

```

TestResult 가 " " .
 TestResult throw
 MoneyTest TestResult
 가 .

```

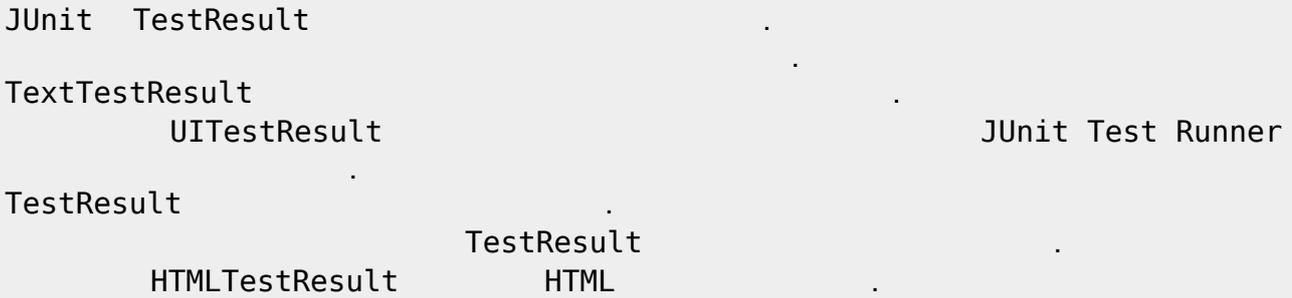
public void testMoneyEquals () {

```

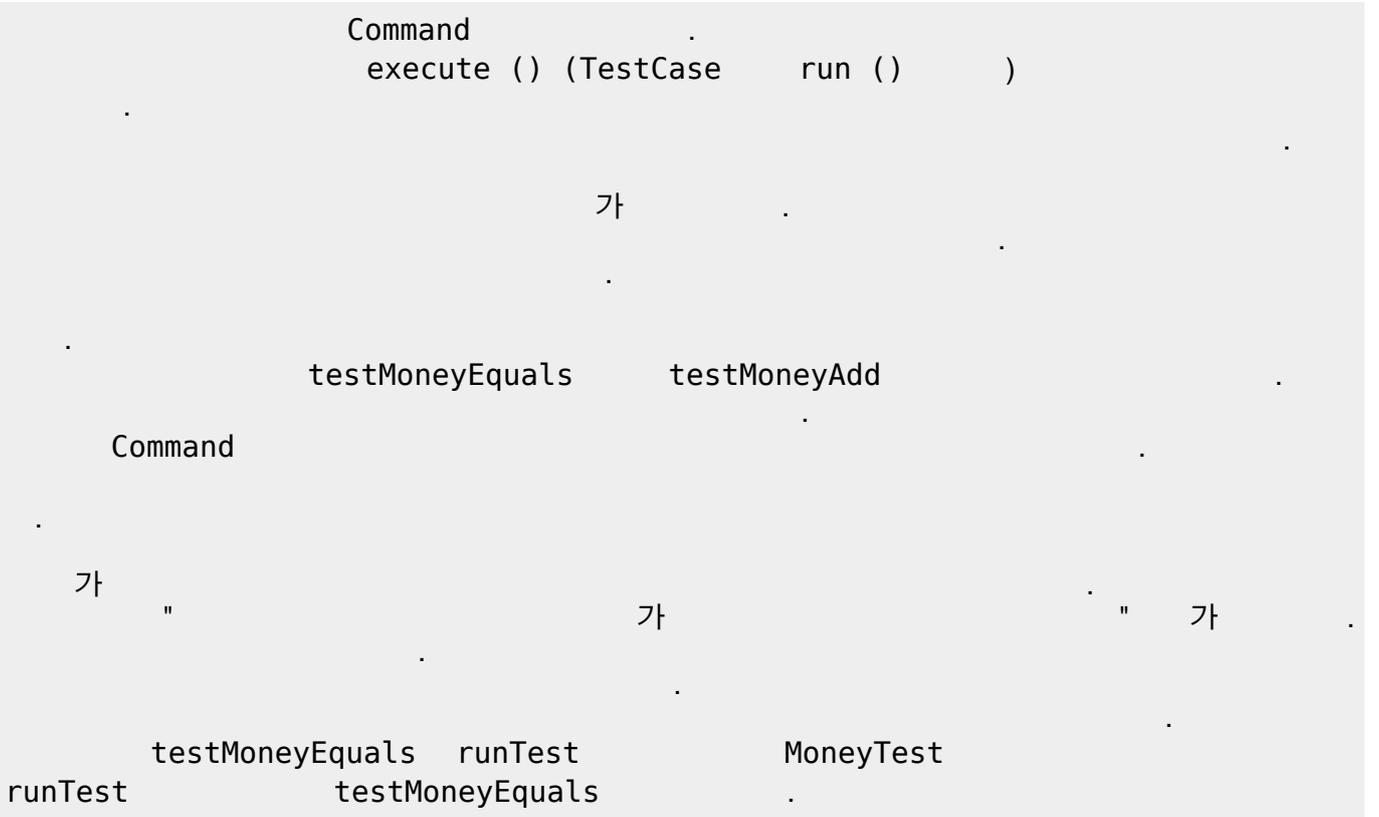
```

assertTrue (! f12CHF.equals (null));
assertEquals (f12CHF, f12CHF);
assertEquals (f12CHF, new Money (12, "CHF"));
assertTrue (! f12CHF.equals (f14CHF));
}

```



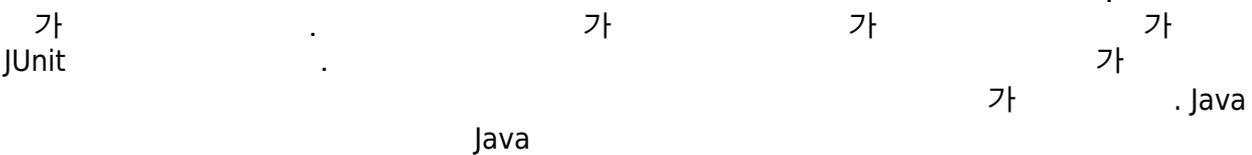
3.4 -TestCase



```

TestMoneyEquals MoneyTest {
public TestMoneyEquals () {super ( "testMoneyEquals"); }
runTest () {testMoneyEquals (); }
}

```

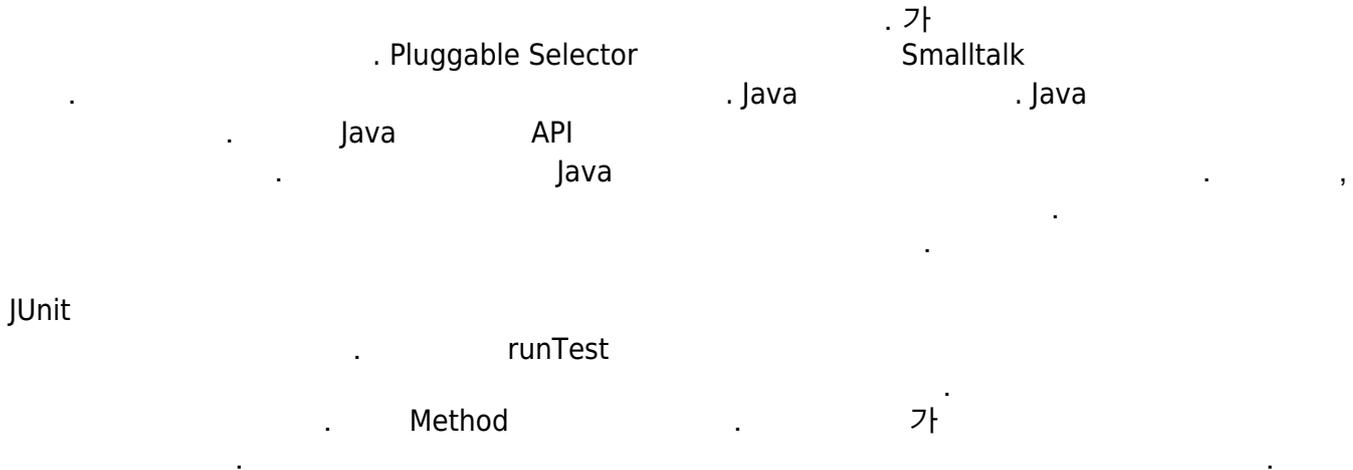


```

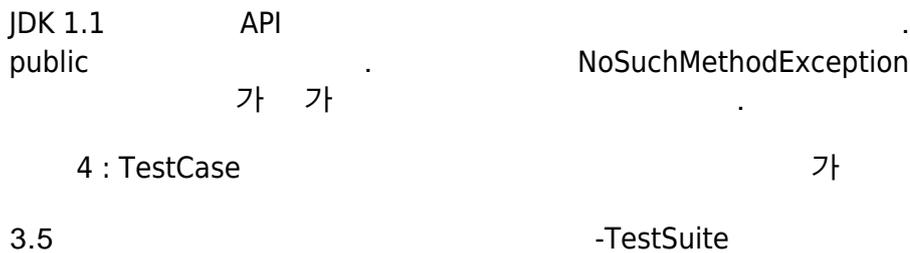
TestCase test = new MoneyTest ( "testMoneyEquals" ) {

```

```
protected void runTest () {testMoneyEquals (); }
};
```



```
protected void runTest () throws Throwable {
    Method runMethod = null;
    try {
        runMethod = getClass (). getMethod ( fName, new Class [0]);
    } catch (NoSuchMethodException e) {
        assertTrue ( "Method \" " + fName + "\" " " ", false);
    }
    try {
        runMethod.invoke (this, new Class [0]);
    }
    // InvocationTargetException IllegalAccessException
}
```





```
public interface Test {
```

```
    public abstract void run (TestResult result);
```

```
} TestCase Composite Leaf
   가                      TestSuite      . TestSuite
```

```
    TestSuite Test {
```

```
        private Vector fTests = new Vector ();
```

```
} run () . public void run (TestResult result) {
```

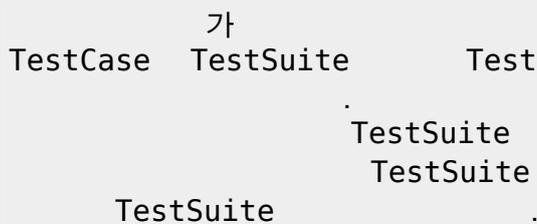
```
    for (Enumeration e = fTests.elements (); e.hasMoreElements ();) {
        Test test = (Test) e.nextElement ();
        test.run ( );
    }
```

```
}
```

```
5 : TestSuite Composite
```

```
    가 addTest
```

```
public void addTest ( ) {
    fTests.addElement (test);
}
```



```
public static Test suite()
{
    TestSuite = new TestSuite ();
    suite.addTest (new MoneyTest ( "testMoneyEquals"));
}
```

```
suite.addTest (new MoneyTest ( "testSimpleAdd"));  
}
```

```
가 . JUnit  
static suite ()  
가 .  
TestSuite 가 .  
"test"  
가 .
```

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

The original way is still useful when you want to run a subset of the test cases only.

3.6 Summary

We are at the end of our cook's tour through JUnit. The following figure shows the design of JUnit at a glance explained with patterns.

Figure 6: JUnit Patterns 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. (The notation of the storyboard is the notation of figure 6 with all the text deleted).

Figure 7: JUnit Pattern Storyboard

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.

4.

To conclude, let's make some general observations:

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. You are now in a perfect position to judge whether describing a framework with patterns is effective. If you liked the discussion above, try the same style of presentation for your own system.

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. We have found that such a high pattern density around key abstractions is common for mature frameworks. The opposite should be true of immature frameworks - they should have low pattern density. Once you discover what problem you are really solving, then you can begin to "compress" the solution, leading to a denser and denser field of patterns where they provide leverage.

Eat your own dog food

As soon as we had the base unit testing functionality implemented, we applied it ourselves. A TestTest verifies that the framework reports the correct results for errors, successes, and failures. We found this invaluable as we continued to evolve the design of the framework. We found that the most challenging application of JUnit was testing its own behavior.

Intersection, not union

There is a temptation in framework development to include every feature you can. After all, you want to make the framework as valuable as possible. However, there is a counteracting force- developers have to decide to use your framework. 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. Sure, we couldn't resist adding some features but we were careful to put them into their own extensions package (test.extensions). A notable member of this package is a TestDecorator allowing execution of additional code before and after a test.

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.
The latest version of JUnit can be downloaded from <http://www.junit.org>.

5. Acknowledgements

Thanks to John Vlissides, Ralph Johnson, and Nick Edgar for careful reading and gentle correction.

From: <http://125.132.25.164/dokuwiki/> -

. - 2023.12

Permanent link: http://125.132.25.164/dokuwiki/doku.php?id=wiki:java:junit:junit_a_cook_s_tour&rev=1598933358

Last update: **2022/03/10 19:52**

