First Iteration - A command-line Application - Part 2

Test-Driven Development Continued

Here is a more fleshed-out version of our JUnitTest (from previous article), including some Spring context features.

JUnitTest:

```
package com.eli.calc.shape.tests;


import static org.junit.Assert.*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.eli.calc.shape.config.AppConfig;

import com.eli.calc.shape.model.CalcType;

import com.eli.calc.shape.model.ShapeName;

import com.eli.calc.shape.service.ShapeCalculatorService;

import com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpl;


public class JUnitTest {
```

```
    private ApplicationContext ctx;

    private ShapeCalculatorService calculator;



    @Before

    public void setUp() throws Exception {

        ctx = new AnnotationConfigApplicationContext(AppConfig.class);

        ((AbstractApplicationContext)ctx).registerShutdownHook();

        calculator = ctx.getBean(ShapeCalculatorService.class);     //by the interface

    }



    @After

    public void tearDown() throws Exception {

        ((AbstractApplicationContext)ctx).close();

    }



    @Test

    public void testDeleteAllResults() {

        calculator.deleteAllResults();

    }



    @Test

    public void testQueueCalculationRequest() {

        double dimension = 0;

        calculator.queueCalculationRequest(ShapeName.CIRCLE, CalcType.CALC_AREA,
dimension);

    }



    @Test
```

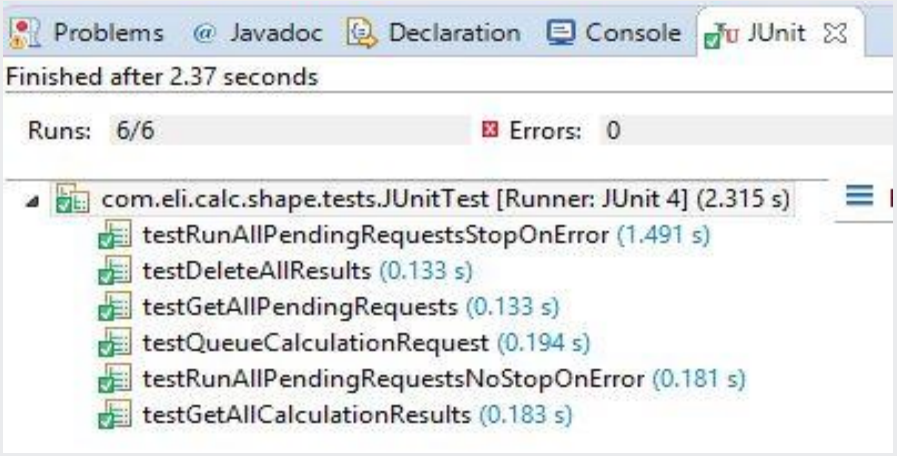
```
    public void testGetAllPendingRequests() {
        calculator.getAllPendingRequests();
    }

    @Test
    public void testGetAllCalculationResults() {
        calculator.getAllCalculationResults();
    }

    @Test
    public void testRunAllPendingRequestsStopOnError() {
        calculator.runAllPendingRequestsStopOnError();
    }

    @Test
    public void testRunAllPendingRequestsNoStopOnError() {
        calculator.runAllPendingRequestsNoStopOnError();
    }
}
```

Run the test...



And they all "pass". These first tests have no assertions, but we at least checked there were no exceptions thrown.

Next, let's add more useful tests.

Note: With JUnit 4, the default behavior seems to be that test-order is not sequential (top-down).

Here are some things we could do.

--Add an annotation to the JUnitClass:

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

--Lump all the tests into one.

--Insure each test stands alone, independent of all the others.

Let's choose the last option.

I came up with a few more tests (that need to be implemented):

```
@Test
    public void testQueueRequestWithNullShapeName() {
}


@Test
    public void testQueueRequestWithNullCalcType() {
}
```

```
@Test
    public void testQueueRequestWithNegativeDimension() {
}

@Test
    public void testQueueRequestAndRetrievePendingRequest() {
}
```

These tests compel us to go back to our requirements, analysis, and design. We need to think about how we want our Shape Calculator Service to behave.

What to do when passed a null or invalid ShapeName or CalcType?

What to do when passed a negative dimension?

Let's add some Javadoc to our interface , to specify behavior.

ShapeCalculatorService:

```
package com.eli.calc.shape.service;

import java.util.List;
import com.eli.calc.shape.domain.CalculationRequest;
import com.eli.calc.shape.domain.CalculationResult;
import com.eli.calc.shape.model.CalcType;
import com.eli.calc.shape.model.ShapeName;

public interface ShapeCalculatorService {

    void deleteAllResults();

    /**
```

```
     *
     * @param shapeName - must not be null;
     * @param calcType  - must not be null;
     * @param dimension - must be greater than / equal to zero;
     *
     * Attempting to queue another request with the same
     * param values will have no effect.
     *
     * An IllegalArgumentException will be thrown
     * if any of the params do not meet criteria
     */
    void queueCalculationRequest( ShapeName shapeName, CalcType calcType, double
dimension);


    /**
     * @return - the list of pending requests
     * (not yet run)
     */
    List<CalculationRequest> getAllPendingRequests();


    /**
     * @return - the list of results
     */
    List<CalculationResult> getAllCalculationResults();


    /**
     * Runs the calculations of all pending requests
     * in a multi-threaded manner.
```

```
     *
     * Not all request queue attempts will make it
     * into the queue. Any previously queued, or
     * previously run request, will not be re-run.
     *
     * Pending/Queued requests are thrown away once
     * they have been run.  Thus, an invocation of
     * this operation removes all pending requests
     * that existed at that instance in time.
     *
     * @return - the number of Requests run
     */
    int runAllPendingRequestsStopOnError();
    int runAllPendingRequestsNoStopOnError();
}
```

Based on the above requirements, let's update our new tests.

```
@Test
public void testQueueRequestWithNullShapeName() {
    try {
        double dimension = 0;
        calculator.queueCalculationRequest(null, CalcType.CALC_AREA, dimension);
    } catch (IllegalArgumentException e) {
        return;
    }
```

```
    fail("Null ShapeName should have caused an exception");
}
@Test
public void testQueueRequestWithNullCalcType() {
    try {
        double dimension = 0;
        calculator.queueCalculationRequest(ShapeName.CIRCLE, null, dimension);
    } catch (IllegalArgumentException e) {
        return;
    }
    fail("Null CalcType should have caused an exception");
}
@Test
public void testQueueRequestWithNegativeDimension() {
    try {
        double dimension = -0.01;
        calculator.queueCalculationRequest(
          ShapeName.CIRCLE, CalcType.CALC_AREA, dimension);
    } catch (IllegalArgumentException e) {
        return;
    }
    fail("Negative dimension should have caused an exception");
}
```

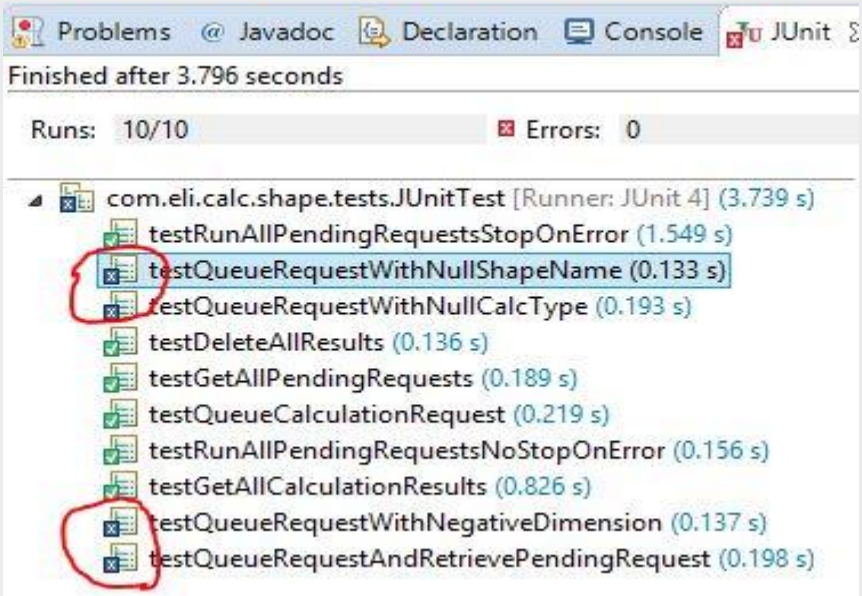
Our last new test, "testQueueRequestAndRetrievePendingRequest()", caused an addition to the ShapeCalculatorService:

```
void deleteAllPendingRequests();
```

```
@Test
public void testQueueRequestAndRetrievePendingRequest() {
    calculator.deleteAllPendingRequests();
    double dimension = 0;
    calculator.queueCalculationRequest(ShapeName.CIRCLE, CalcType.CALC_AREA, dimension);
    List<CalculationRequest> requests = calculator.getAllPendingRequests();
    assertNotNull(requests);
    assertEquals(1,requests.size());
}
```

While the new method may not be used in actual instances... it comes in handy for this test, by guaranteeing the desired return size of list.

Let's run the JUnitTest.....



Three of the tests failed because they did NOT throw the expected exceptions, and the fourth one failed because it DID return a null list.

It is time to flesh out some of our ShapeCalculatorServiceImpl, so these tests will pass.

The first three tests are easily handled by:

ShapeCalculatorServiceImpl:

```
@Override

public void queueCalculationRequest(ShapeName shapeName, CalcType calcType, double
dimension) {

if (null==shapeName) { throw new IllegalArgumentException('ShapeName can not be null'); }

if (null==calcType) { throw new IllegalArgumentException('CalcType can not be null'); }

if (0>dimension) { throw new IllegalArgumentException('dimension must be zero or positive'); }

}
```

Making the fourth test pass is not so trivial. We now have to think deeper about our implementation class' internal functionality.

Continued in next article....