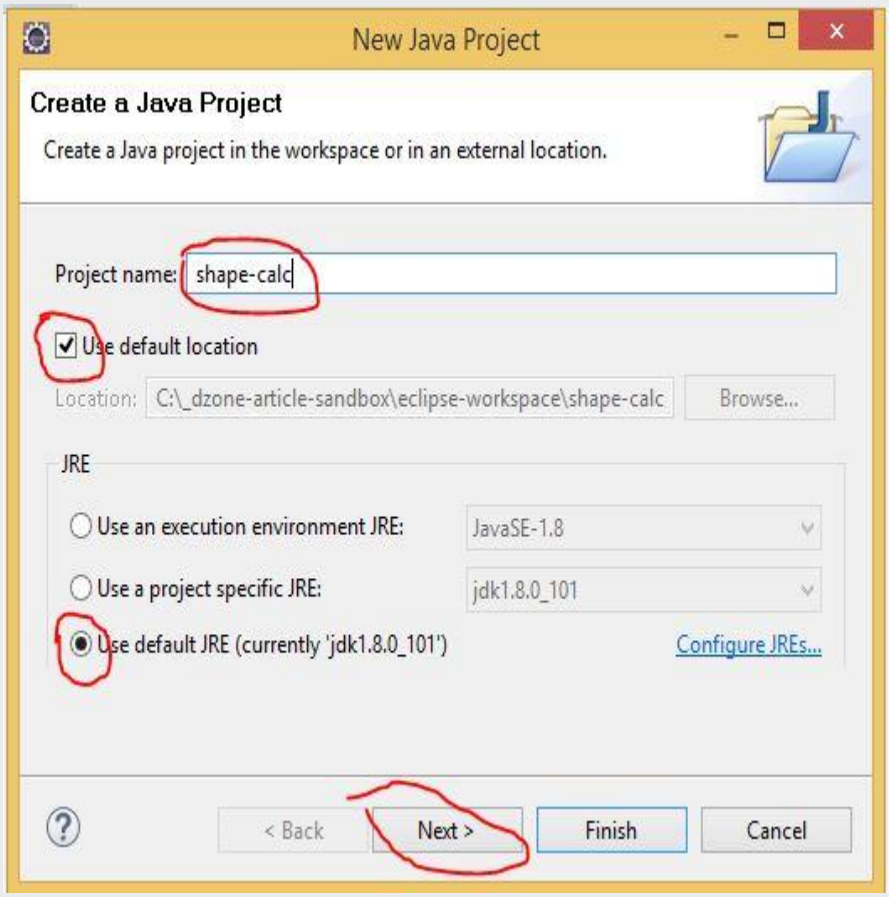
First Iteration - A command-line Application (a possible future component or service)

New Project

(This article may seem a bit drawn-out or slow to some readers. Hopefully we can speed things up in upcoming articles of this series)

Create the Project Structure

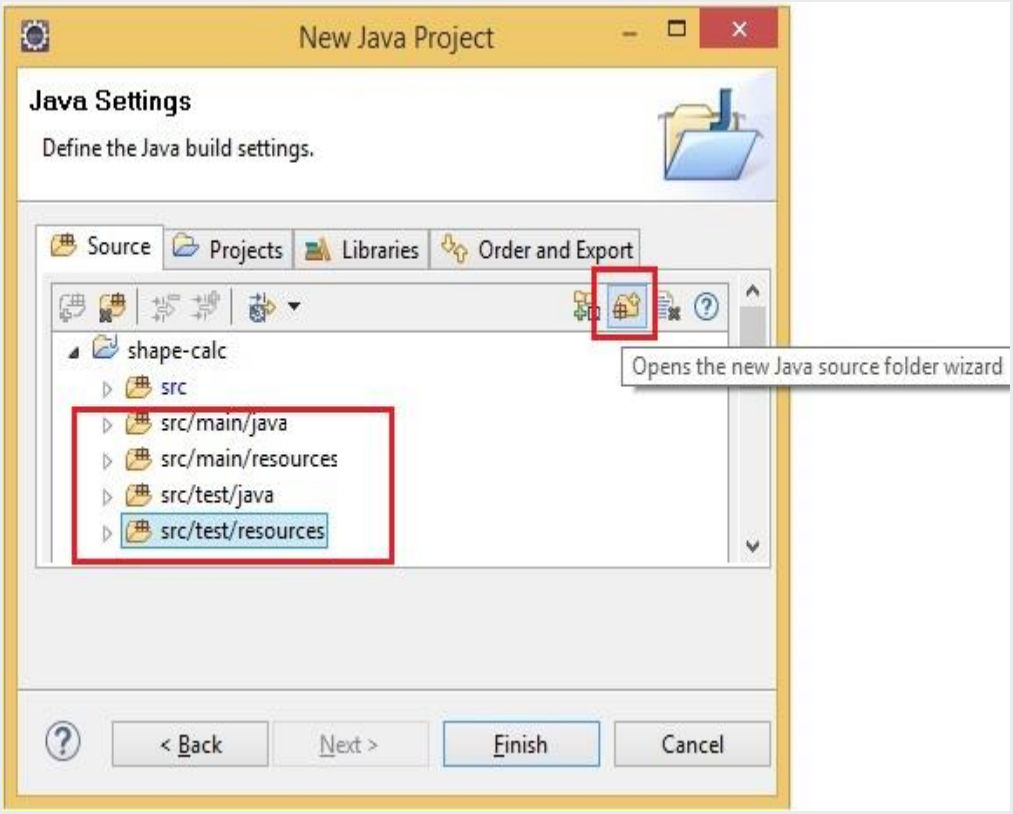
Let's create a new project. I elected to create a Java Project, and named it "shape-calc" (for 'Shape Calculator'). In same dialog where you enter the project name, below, make sure you select "Use default JRE (currently 'jdk1.8.0_101')". Click "Next".



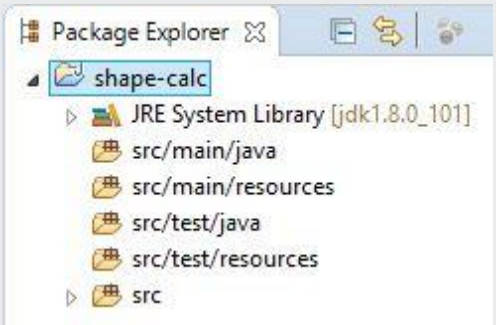
The next part has to do with "source" folders. We want to add some source folders.

Since we will be using Maven, it has a notion of expected source folders.

See https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html for the list of standard folders.



As my personal preference, I went to the "Order and Export" tab (see above image) and re-arranged the top-down order of the folders. And I clicked "Finish". Here is the result:



Analyze Requirements

Here again are the requirements from the first article in this series:

"Given a single number (either it is the radius or length of an edge), write a program that will calculate the area of a circle, a square, and an equilateral triangle. Make the program multi-threaded to handle many calculations."

So we possibly need:

- a calculator
- calculations

I think we should keep those separate.

Also, rather than jump into a calculator implementation, we should use start with an interface, since we will probably go through iterations of calculators, and we will most likely use Spring later on, and for other reasons.

Initial Design

Let's define our calculator contract.

Let's call it ShapeCalculatorService, because in this case we saw into the future and knew we would be offering this as a service. :D

From the requirements, a calculation involves:

- a shape
- what we want to calculate (example: area, volume, or yet-to-be-determined)
- a dimension (single dimension, only certain shapes are applicable)

And with the above three parameters, we will arrive at a calculation result.

So our ShapeCalculatorSerivce interface could look somewhat like so:

-- deleteAllResults()

--queueCalculationRequest(ShapeName, CalcType, dimension)

--getPendingRequests()

--getAllResults()

--runAllPendingRequestsStopOnError()

--runAllPendingRequestsNoStopOnError()

This is not a real project, not even a worthy one - our purpose here is just to review / use various technologies.

I chose to queue and run separately, in order to attempt to demonstrate the "multi-threadedness" of this application.

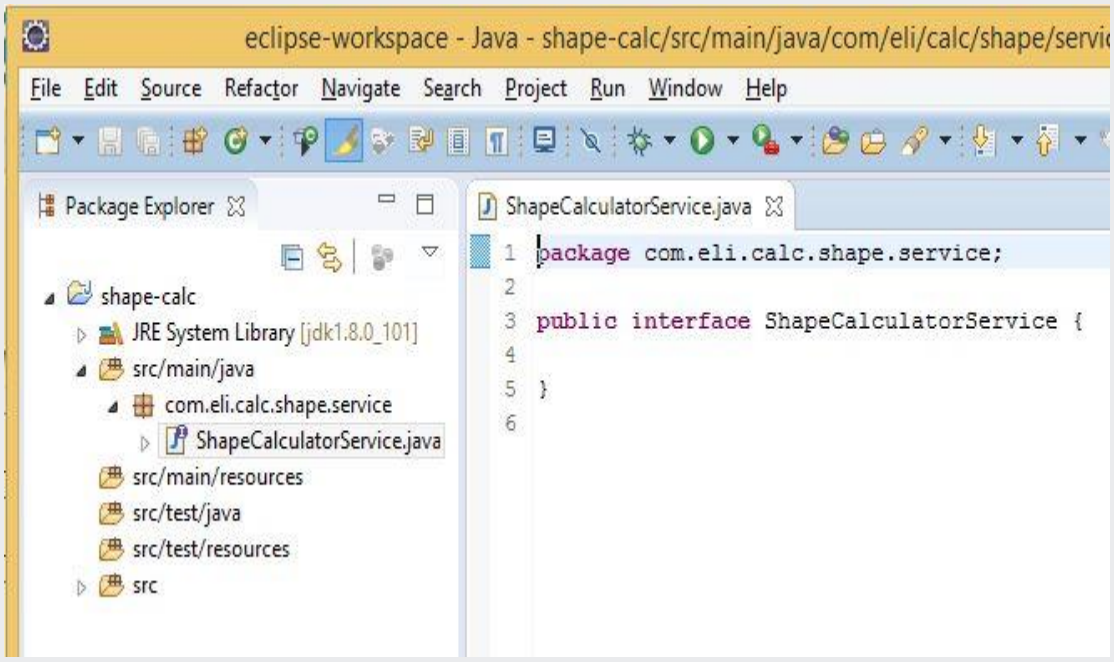
Others will have their own opinions of how the above should be. For example, what if there are a million calculation results? Do we want to return all of them at once?

Since the main purpose of this article series is to employ various tools and technologies, and not to produce the absolute best ShapeCalculatorService. I think what we have will suffice.

Initial Implementation (from OOA, OOD, OOP)

Interface

With what we have let's create a new Java interface....



Rabbit Trail into Vi Editor (optional)

When the new file appears in the editor, I realize that I want to use vi keystrokes, so...

(please skip if not interested)

Go to Help->Eclipse Marketplace and install "vrapper".

Why do I like vi commands? The main reason is I hate to use the mouse. I am much faster with my fingers. (Yes, I also use standard window-style key-sequence short-cuts. That is what is nice about vrapper - it tries to allow as much of both worlds as possible.)

But granted, vi commands are very cryptic.

Continue With Interface

After I re-started my Eclipse, we're back at our newly-created java interface... we add what we think is our contract ....

*ShapeCalculatorService.java

```
1  package com.eli.calc.shape.service;
2
3  public interface ShapeCalculatorService {
4
5      void deleteAllResults();
6
7      void queueCalculationRequest(
8              ShapeName shapeName,
9              CalcType calcType,
10             doule dimension
11             );
12
13     List<CalculationRequest> getAllPendingRequests();
14
15     List<CalculationResult> getAllCalculationResults();
16
17     int runAllPendingRequestsStopOnError();
18
19     int runAllPendingRequestsNoStopOnError();
20
21 }
```

It seems we have some work to do. Let's import java.util.List (of course)...

Now, thinking ahead, we probably will want to persist our data. Otherwise what good is it.

We will be using Hibernate (later), and as I recall, it has some issues with interfaces.

Add Supporting Classes

So we will begin with concrete classes CalculationRequest and CalculationResult.

In creating the classes, Eclipse indicates an error:

"Syntax error, parameterized types are only available if source level is 1.5 or greater"

If you recall, we set our Java compliance during the Eclipse set up. And yet, upon checking our "project-specific" settings, we find that it is NOT set to Java 1.8 compliance.

We fix that and move on.

So far we have:

ShapeCalculatorService.java

```
8  public interface ShapeCalculatorService {
9
10     void deleteAllResults();
11
12     void queueCalculationRequest(
13             ShapeName shapeName,
14             CalcType calcType,
15             doule dimension
16             );
17
18     List<CalculationRequest> getAllPendingRequests();
19
20     List<CalculationResult> getAllCalculationResults();
21
22     int runAllPendingRequestsStopOnError();
23
24     int runAllPendingRequestsNoStopOnError();
```

CalculationResult.java

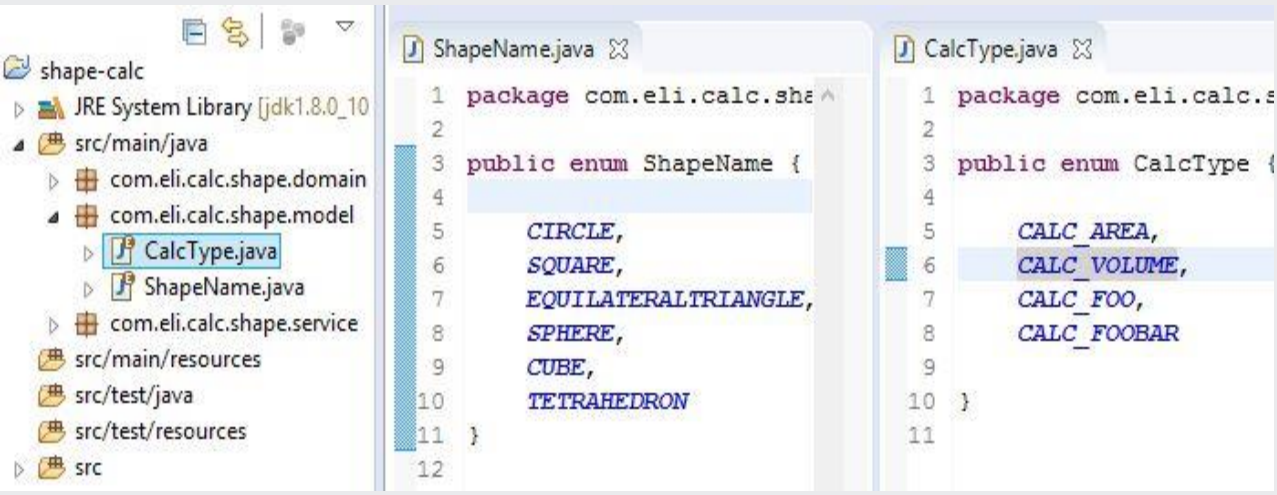
```
1  package com.eli.calc.shape.domain;
2
3  public final class CalculationResult {
```

CalculationRequest.java

```
1  package com.eli.calc.shape.domain;
2
3  public final class CalculationRequest {
4
```

We now need a ShapeName, a CalcType, and to fix a typo (yikes!).

My thinking here is that these should be enums. Let's do it.



More Detailed Analysis

Take a look at this article: Further Analysis of Shape Calculator - Calculation Requests & Calculation Results

as an explanation why I chose CalculationRequest and CalculationResult.

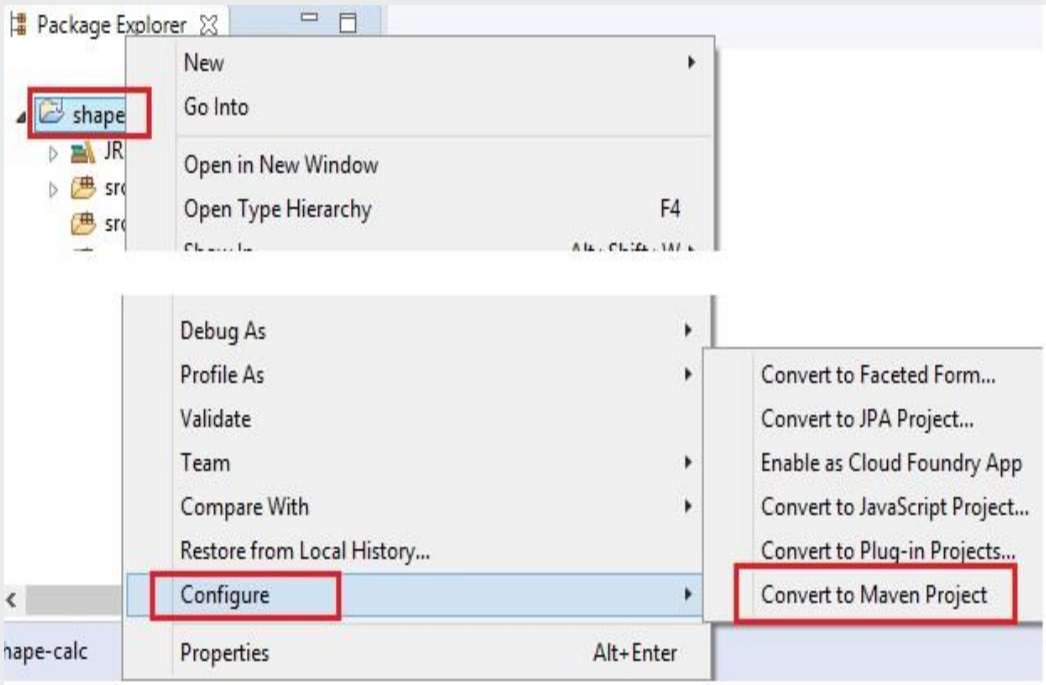
Those two entities are the basis for the entire service.

Setup ForTest-Driven Development (TDD)

Let's attempt to create our first test of what we have so far... I am thinking to use JUnit.

We could download it and set it up in Eclipse under Preferences. Or, we can use Maven.

We convert our project to a Maven project...



Eclipse will pop a POM-creation wizard. Just accept the defaults, click "Finish".

Since we setup our preferences in Eclipse, the new POM appears in XML view.

Let's add a "<dependencies></dependencies>" section to our POM, placing it between the <version> and the <build> sections. (Just my preference).

Go to https://mvnrepository.com/, search for JUnit, and you will arrive at this dependency:

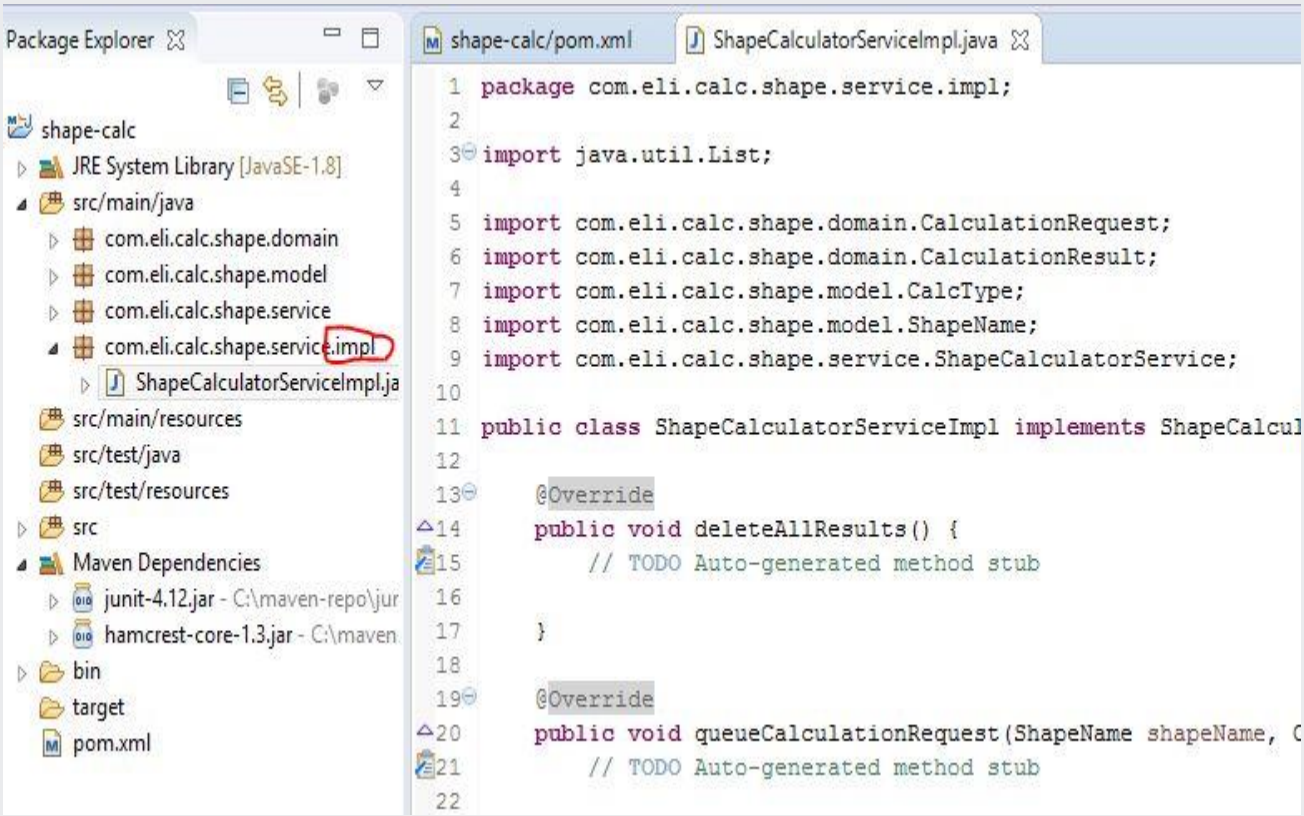
```
<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>
```

Since "automatic build" is usually the Eclipse default, as soon as I plopped in the dependency and saved the POM, Eclipse went out and retrieved the necessary jars. You can view them in the new "Maven Dependencies" section that is now part of our new project.

Back to Implementation (from OOA, OOD, OOP)

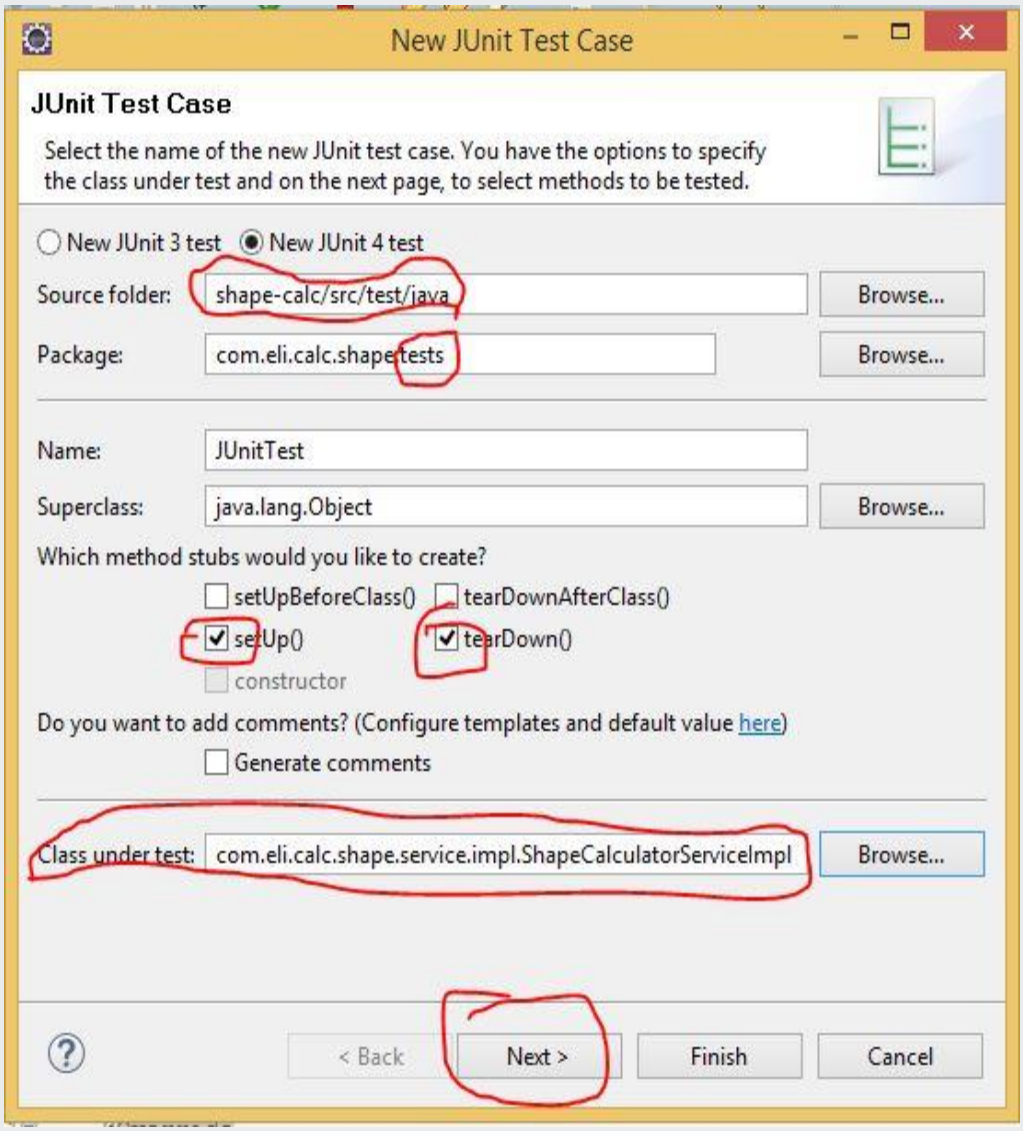
One approach to software development can be iterative and incremental, jumping back and forth to analysis, design, implementation, testing, and so forth, as needed. This is in contrast, for instance, to a waterfall approach.

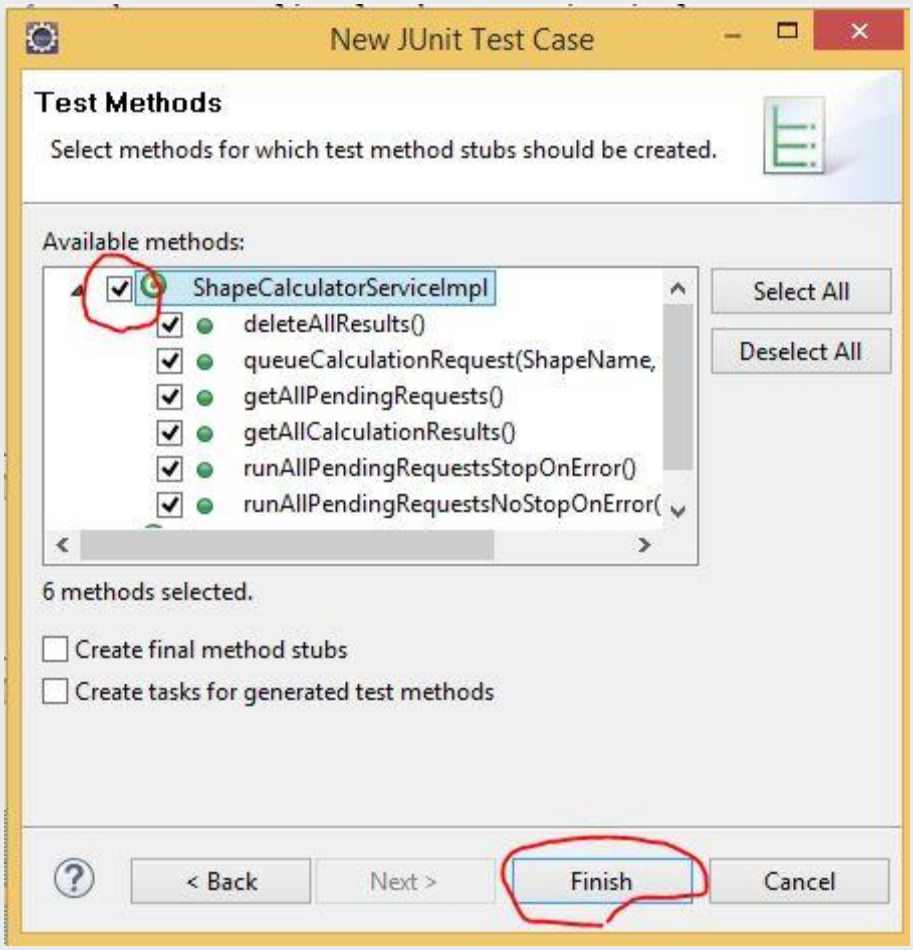
We now need at least a basic implementation class for our ShapeCalculatorService interface.



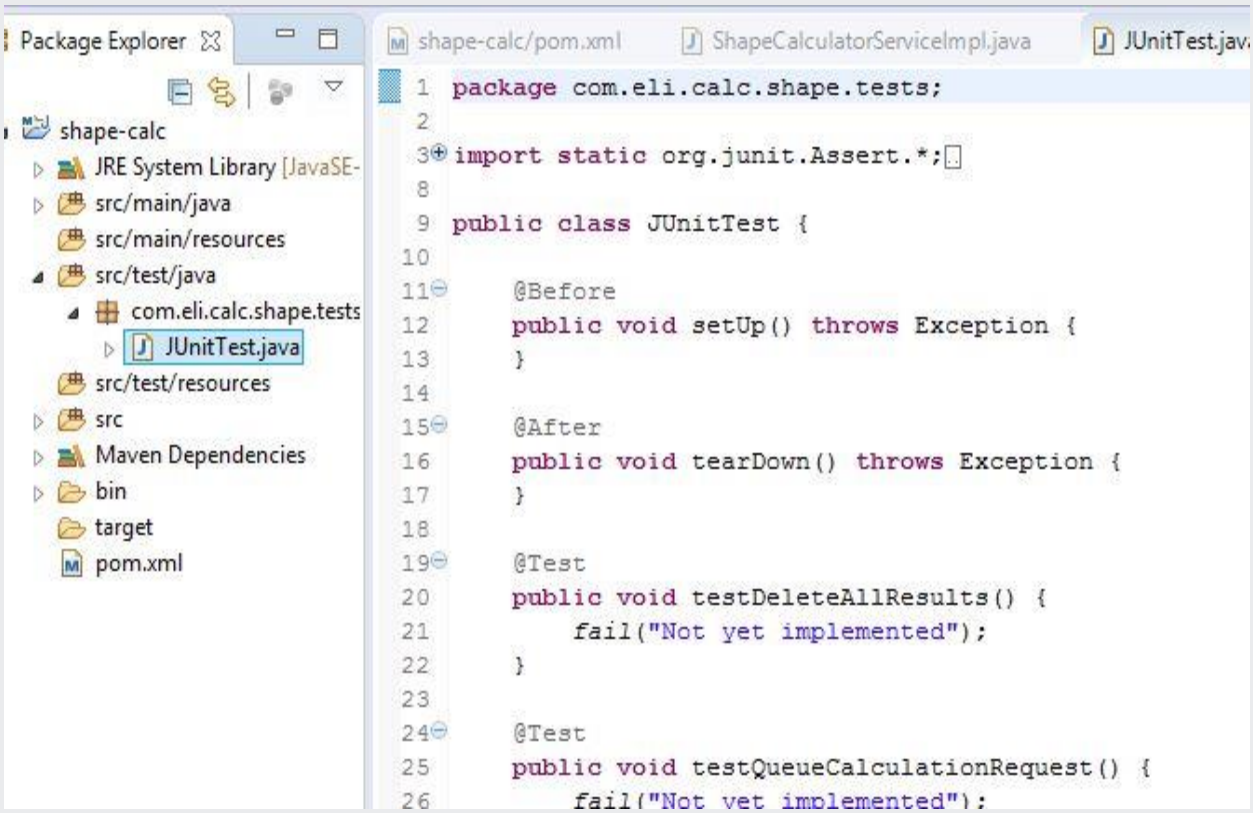
We create the new class in an "impl" sub-package, making sure to select the interface class, and we have Eclipse add our method stubs.

Continue with TDD


New JUnit Test Case
JUnit Test Case
Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.
New JUnit 3 test
New JUnit 4 test
Source folder:
shape-calc/src/test/java
Browse...
Package:
com.eli.calc.shape.tests
Browse...
Name:
JUnitTest
Superclass:
java.lang.Object
Browse...
Which method stubs would you like to create?
setUpBeforeClass()
tearDownAfterClass()
setUp()
tearDown()
constructor
Do you want to add comments? (Configure templates and default value here)
Generate comments
Class under test:
com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpl
Browse...
< Back
Next >
Finish
Cancel



The above results in the following:



We may or may not use all of the created tests, and we may create our own, but Eclipse seemed at least a bit helpful here. (another reason I grudgingly let go somewhat of my old command-line development ways).

Introduce Spring into Project

Since we will very likely be using Spring, might as well start here, with this new JUnit test.

Why use Spring? We could just do the following to our JUnit class:

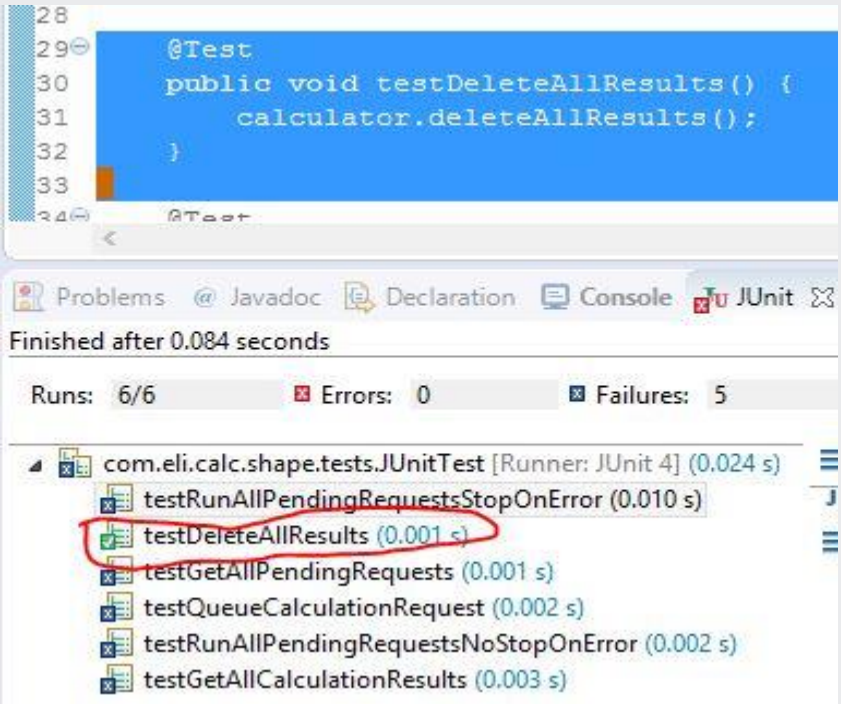
```
public class JUnitTest {

        private ShapeCalculatorService calculator;

@Before

public void setUp() {



        calculator = new ShapeCalculatorServiceImpl();
```

```
}

  // ....more code here....

}
```

Let's implement one test:

```
@Test

public void testDeleteAllResults() {

        calculator.deleteAllResults();

}
```

Do a 'Run as.... " JUnit test. Below are the results.



Notice that the one implemented test does pass.

However, the test class, or any subsequent code that uses our "calculator", is closely-coupled to the calculator implementation.

It means it knows what is the exact package, and exact class. What if we want to change the implementation to something better or different?

It would be nice not to have to worry about where, what, and how so much.

We can do that (de-coupling) using Spring.

Spring XML Configuration

Get Spring Framework

First we need to get some spring jars.

Go to https://mvnrepository.com/, search for spring-context, and you will arrive at this dependency:

```
<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>
```

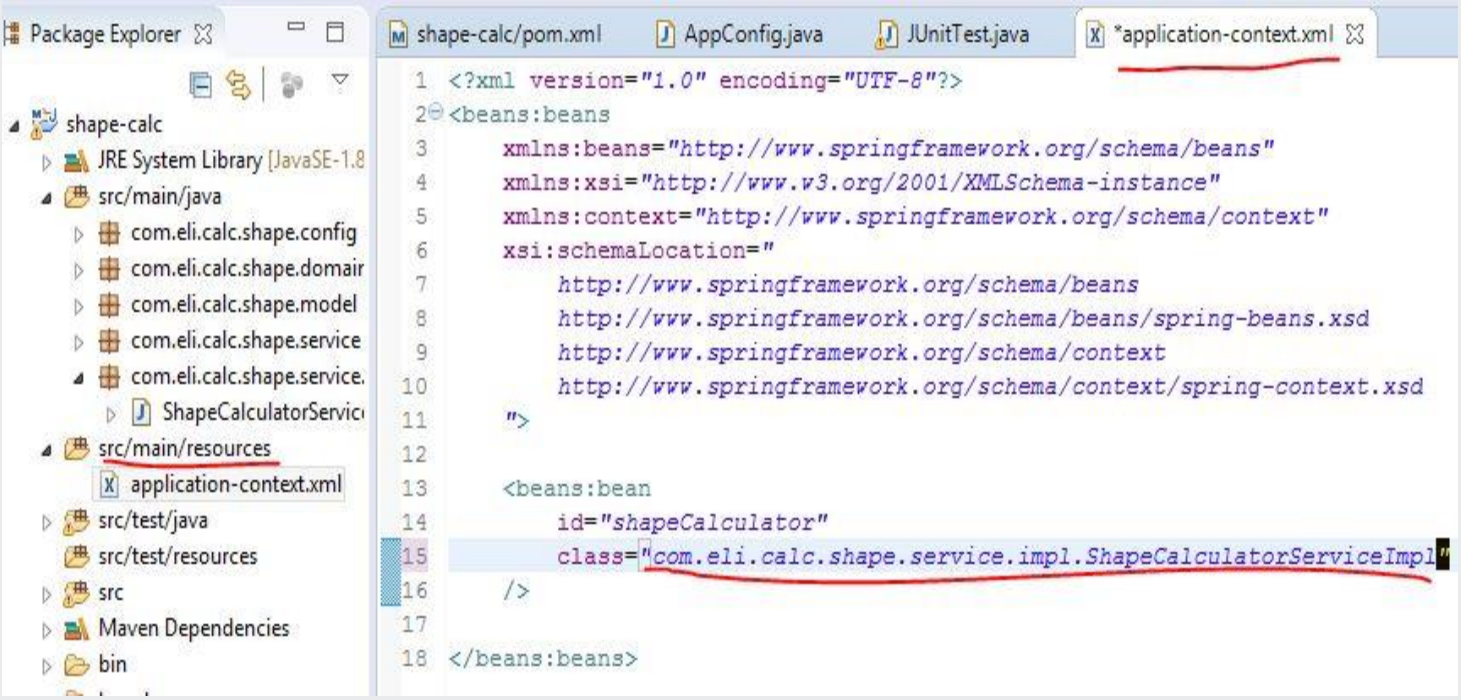
We need to add some properties to our POM:

```
<properties>

<spring.version>4.3.2.RELEASE</spring.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>
```

When you save, you will notice new spring jars in the "Maven Dependencies" folder.

Specifying A Bean (an implementation)

Now we need to configure Spring context. One way to do this is to create an XML configuration file. Create an application-context.xml file in src/main/resources, and create a <bean> , which is our ShapeCalculatorServiceImpl class.



Then, in the JUnitTest class, change the code in the setUp() method.

```
public class JUnitTest {

private ShapeCalculatorService calculator;

@Before

public void setUp() throws Exception {

    ApplicationContext ctx =

      new ClassPathXmlApplicationContext("application-context.xml");

    //find by the implementation class

    calculator = ctx.getBean(ShapeCalculatorServiceImpl.class);

    // find by the id

    //calculator = (ShapeCalculatorService)ctx.getBean("shapeCalculator");

    // find by the interface

    //calculator = ctx.getBean(ShapeCalculatorService.class);

}
```

If you run the test using any one of the above lines of code, it will work.

However, using the first (uncommented) line is pointlesss - it is no better than how we started (by creating a new ShapeCalculatorServiceImpl()).

The second (commented) line isn't all that great, since it is using a string to associate with the desired class, and then we also have to do a cast.

The third (commented) line is much better, since it specifying what we already have in the code anyway - the interface.

By using Spring, and configuring outside of our code, we have somewhat de-coupled the service from our application (or test) code.

However, we could de-couple it even further. Right now, in the application-context.xml file, we had to specify the fully qualified class name.

It would be better to, say if in the future we want a completely different implementation, including a different package name, that we not have to back into application-context.xml file to make the change.

This can be done by having Spring "scan" packages for implementations of the ShapeCalulatorService interface.

Try This Exercise (optional)

Before we introduce the "scan" feature, go into the application-context.xml file, and change the spelling of the class very slightly.

From this: com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpl

To this: com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpll

Notice the extra "l" in the "Impl". So we have a typo.

```
WARNING: Exception encountered during context initialization - cancelling
refresh attempt:
org.springframework.beans.factory.CannotLoadBeanClassException: Cannot
find class [com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpll] for
bean with name 'shapeCalculator' defined in class path resource [application-
context.xml]; nested exception is java.lang.ClassNotFoundException:
com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpll
```

That is one short-coming of using an XML-based configuration for the Spring context, if the beans are explicitly declared.

We can improve on that by using "component-scanning".

Using Package Scanning / Annotations

To setup for scanning, add this line to the application-context.xml file:

```
<context:component-scan base-package="com.eli.calc.shape" />
```

And comment out the explicit <bean> definition. Your file should look like so:

shape-calc/pom.xml | AppConfig.java | JUnitTest.java | *application-context.xml | FakeU

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans:beans
3      xmlns:beans="http://www.springframework.org/schema/beans"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xmlns:context="http://www.springframework.org/schema/context"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd
11     ">
12
13     <context:component-scan base-package="com.eli.calc.shape" />
14
15 <!--
16     <beans:bean
17         id="shapeCalculator"
18         class="com.eli.calc.shape.service.impl.ShapeCalculatorServiceImpl"
19     />
20 -->
21
22 </beans:beans>
```

Now Spring knows where to scan for candidate bean. But, we need also configure such a bean. As things are now, Spring will find nothing if you run the test.

Open and edit the implementation class, and at the class level, specify the @Component annotation. Why not @Bean? Notice the line in the XML file says "component-scan", not "bean-scan". A Component is somewhat like a Bean for our purposes , but I am being fast and loose with that statement. You can google for more information.

ShapeCalculatorServiceImpl:

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

import java.util.List;

import org.springframework.stereotype.Component;

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;

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

@Component

public class ShapeCalculatorServiceImpl implements ShapeCalculatorService
{

    @Override

    public void deleteAllResults() {

    // TODO Auto-generated method stub

    }

  // ....more code here...
```

Spring will be scanning packages for any "Component" that "implements" the desired interface, ShapeCalculatorService.

Eureka. If you run the JUnitTest now, our single implemented method will pass, because compononent-scanning found a qualifying implementation for our interface.

That's pretty de-coupled.

There is type-checking because the ShapeCalculatorServiceImpl class was already checked by Eclipse when you wrote it - now to declare it as a bean, all we did was a @Component.

Spring Java Configuration

So, Spring, as it we have used it til now, is pretty cool. XML-based configuration is very popular, and by using annotations, it is even better.

There is another way to do the same thing as XML, and it is using Java Config.

We are going to create a Java class that will replace (act the same as) the application-context.xml file.

Create the following class in the new package:

Somehow , we need to inform Spring that this new AppConfig class is now the equivalent of the application-context.xml file.

We again use an annotation to do so. Add @Configuration to it at the class level. And we need another annotation, to scan packages just as was done with the application-context.xml file. Right below the @Configuration, add a :

```
@ComponentScan(basePackages="com.eli.calc.shape")
```

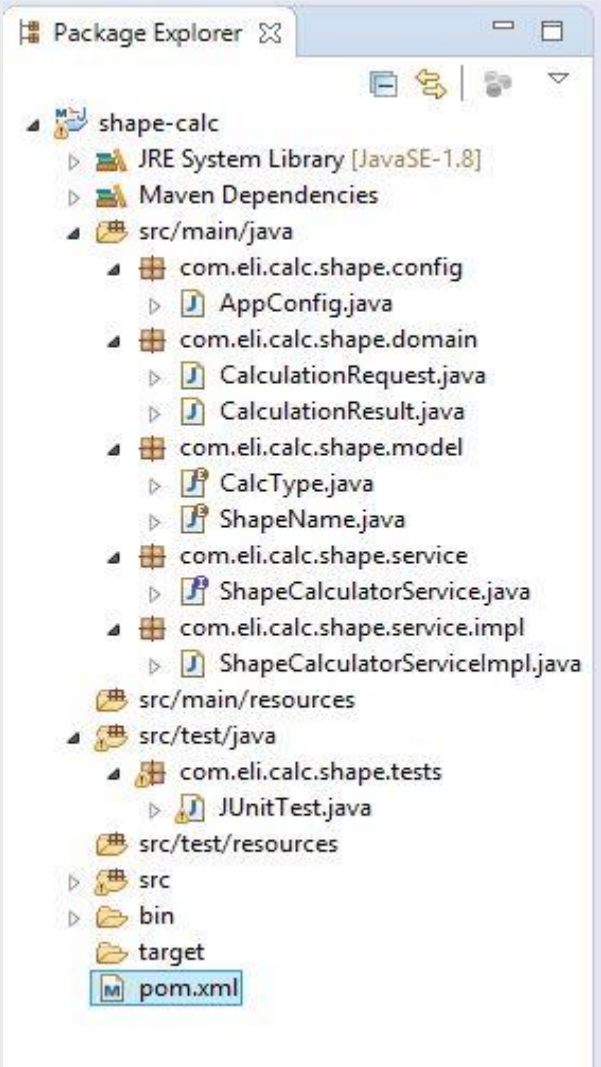
(Notice that the AppConfig class body is empty, and the class containing just two annotations. More about this later)

To complete the changes, comment anything of significance inside the application-context.xml file.

And run the test. And voila - our single implemented test again passes.

Review Of What We Have So Far

You should check that you have the following packages and classes in your project:



Here is the code, in order from top-down:

AppConfig:

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

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;
```

```
@Configuration

@ComponentScan(basePackages="com.eli.calc.shape")

public class AppConfig {

}
```

CalculationRequest and CalculationResult are empty for now.

ShapeName and CalcType are enums.

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();

    void queueCalculationRequest(

        ShapeName shapeName,

        CalcType calcType,

        double dimension

    );

    List<CalculationRequest> getAllPendingRequests();

    List<CalculationResult> getAllCalculationResults();
```

```
    int runAllPendingRequestsStopOnError();

    int runAllPendingRequestsNoStopOnError();

}
```

ShapeCalculatorServiceImpl:

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

import java.util.List;

import org.springframework.stereotype.Component;

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;

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

@Component

public class ShapeCalculatorServiceImpl implements ShapeCalculatorService
{

    @Override

    public void deleteAllResults() {

        // TODO Auto-generated method stub

    }

    @Override

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

        // TODO Auto-generated method stub
```

```
    }

    @Override

    public List<CalculationRequest> getAllPendingRequests() {

        // TODO Auto-generated method stub

        return null;

    }

    @Override

    public List<CalculationResult> getAllCalculationResults() {

        // TODO Auto-generated method stub

        return null;

    }

    @Override

    public int runAllPendingRequestsStopOnError() {

        // TODO Auto-generated method stub

        return 0;

    }

    @Override

    public int runAllPendingRequestsNoStopOnError() {

        // TODO Auto-generated method stub

        return 0;

    }

}
```

And finally, our 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.ClassPathXmlApplicationContext;

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

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

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

public class JUnitTest {

    private ShapeCalculatorService calculator;

    @Before

    public void setUp() throws Exception {

        //ApplicationContext ctx =

            new ClassPathXmlApplicationContext("application-context.xml");

        ApplicationContext ctx =

            new AnnotationConfigApplicationContext(AppConfig.class);

        // find by interface

        calculator = ctx.getBean(ShapeCalculatorService.class);

    }

    @After
```

```
public void tearDown() throws Exception {

}

@Test

public void testDeleteAllResults() {

    calculator.deleteAllResults();

}

@Test

    public void testQueueCalculationRequest() {

    fail("Not yet implemented");

}

@Test

public void testGetAllPendingRequests() {

    fail("Not yet implemented");

}

@Test

public void testGetAllCalculationResults() {

    fail("Not yet implemented");

}

@Test

public void testRunAllPendingRequestsStopOnError() {

    fail("Not yet implemented");

}

@Test

public void testRunAllPendingRequestsNoStopOnError() {
```

```
        fail("Not yet implemented");

    }

}
```

Continued in next article....