BBM490

ENTERPRISE WEB ARCHITECTURE

Mert ÇALIŞKAN

Hacettepe University Spring Semester ’15

Unit Testing with JUnit & Integration Testing with Spring

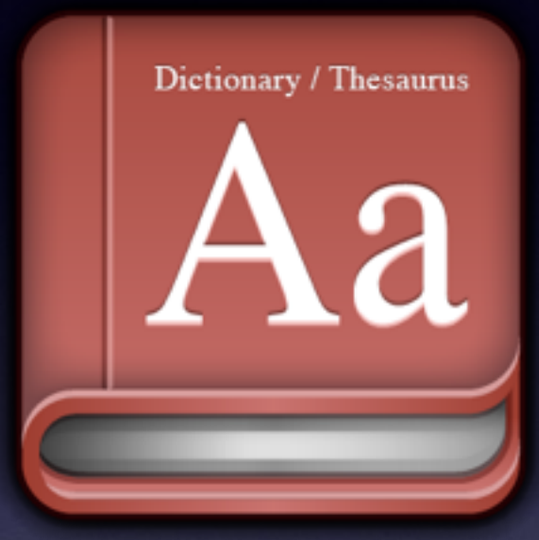
We will define what unit testing is and demonstrate it with JUnit.
We will also get to know the ways of testing Spring beans with integration testing and learn the details of creating mock objects.

Your Software, after some coding…

But we should aim for a ravioli…

Fundamentals of Testing

webster says that...



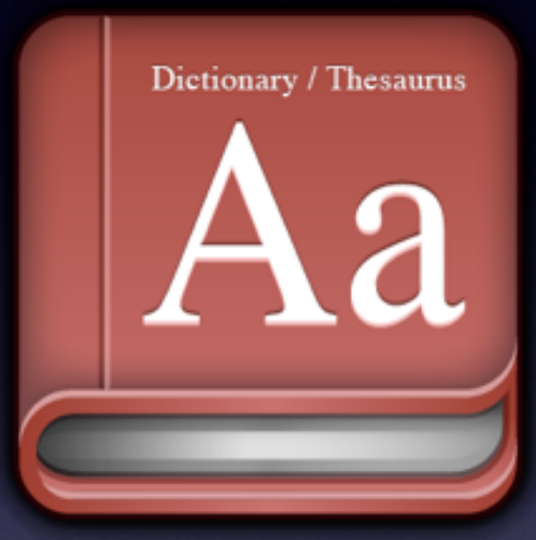
test [1] |tɛstl|

noun

a procedure intended to establish the quality, performance, or reliability of something, especially before it is taken into widespread use...

Fundamentals of Testing

webster says that...



test [1] |tɛstl

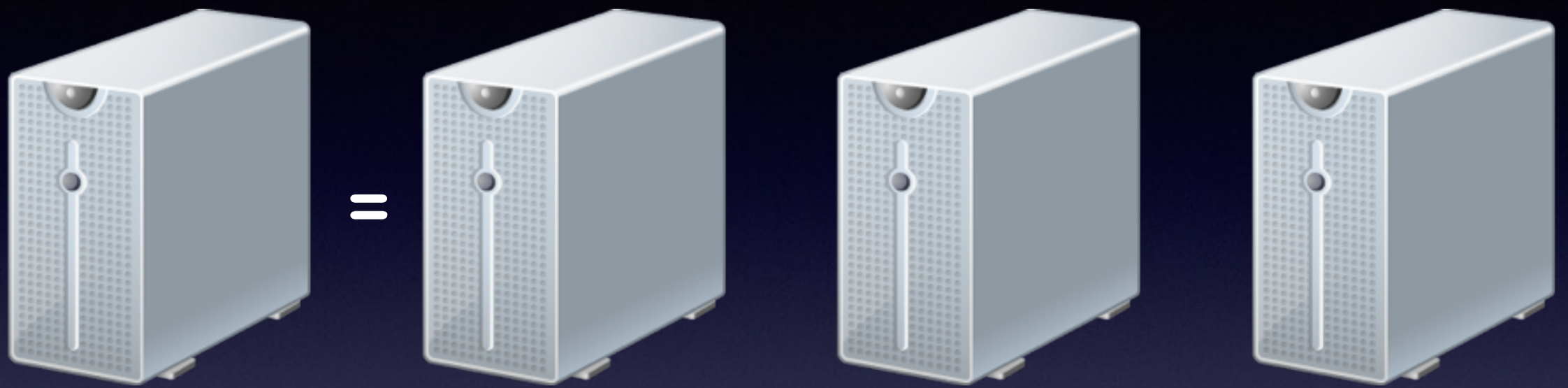
noun

a procedure intended to **establish** the **quality**, **performance**, or **reliability** of something, especially before it is taken into widespread use...

What to Test?

- Unit Tests
  - every public method… with its every line of code…
- Integration Tests
  - DB layer, network layer… the parts that are integrated with.
- Mock Tests
  - Service layer
- Functional Tests
  - UI Testing (Selenium), WS Testing (SoapUI)

Testing Methods/Env's

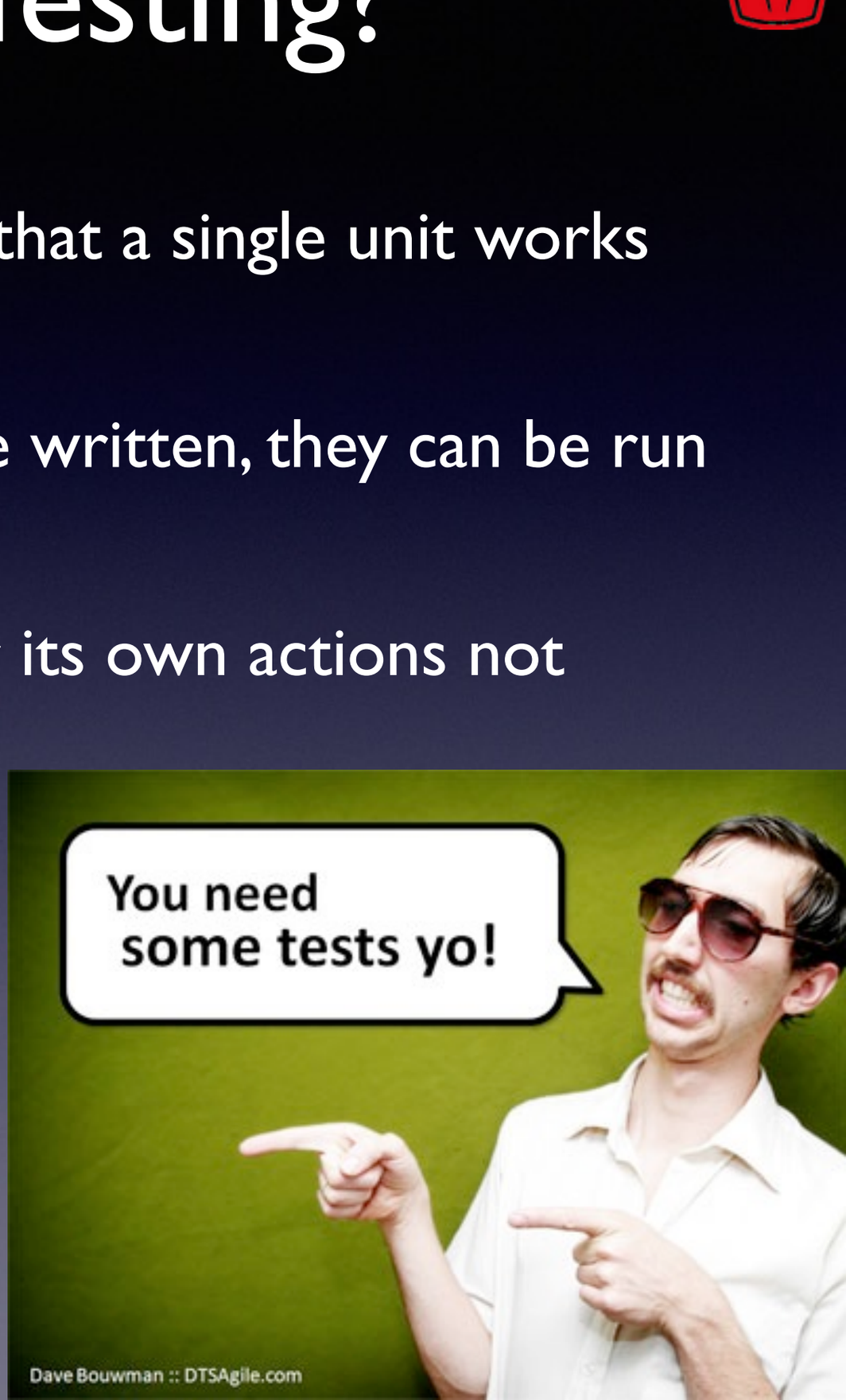


Prod Staging Test Local

- Go for 3 different locations other than local if it's applicable.
- Try to make Prod and Staging identical...

What's Unit Testing?

- In unit testing, you're just verifying that a single unit works according to expectations.
- Unit tests are not interactive. Once written, they can be run without further input.
- Each test should be responsible for its own actions not interfere with each other.



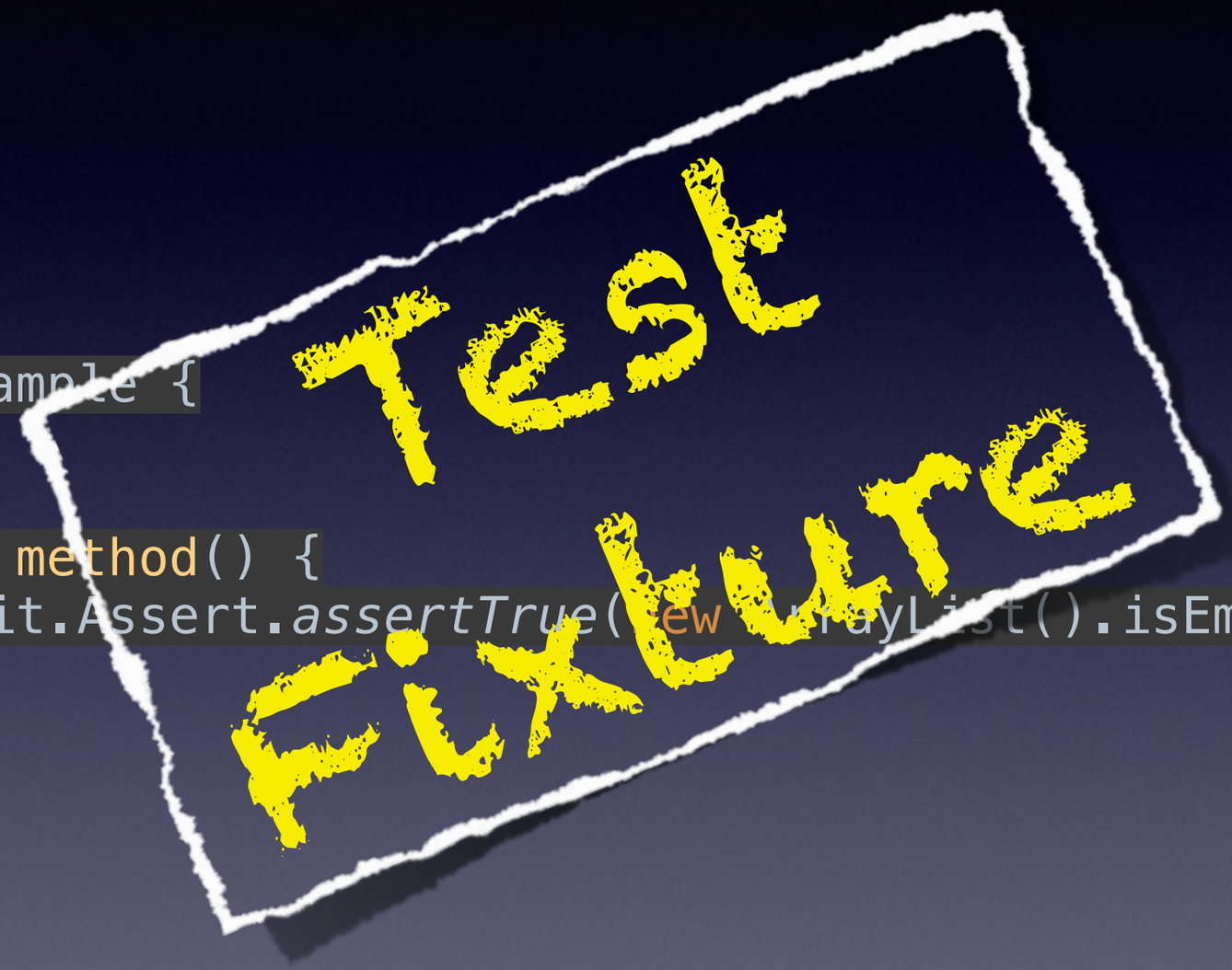
Common Rules for Testing Frameworks

- Each unit test must run independently of all other unit tests.
- Errors must be detected and reported test by test.
- It must be easy to define which unit tests will run.

JUnit

- Founders, Kent Beck and Erich Gamma,
- like ~14 years ago.
- Kent Beck the creator of sUnit and tossed the coin on the term, XP.
- Major differences between 3.x and 4.x
- new packaging, annotations and etc...
- The Latest and Greatest of jUnit is: 4.12 (we will use this version)

The Simplest Test




```
public class Example {

    @Test
    public void method() {
        org.junit.Assert.assertTrue(new ArrayList().isEmpty());

    }
}
```

The Simplest Test

- What happens with the given simplest test?
  - jUnit creates a new instance of Example class.
  - It invokes the method() which is annotated with @Test
    - The method should be declared as public
- @Test annotation bundles 2 optional parameters.
  - expected: the exception that the method should throw.
  - timeout: the amount of time in ms for the test to fail.

Preparing and Tearing Down for Tests

- @Before
- @BeforeClass

- @After
- @AfterClass

@Before and @After

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class SampleTest {

    @Before
    public void setup() {
        // prepare some stuff..
    }

    @Test
    public void checkIfTheCodeWorksOkOrNot() {
        // Some assertions and bla bla...
    }

    @After
    public void tearDown() {
        // tear down all...
    }
}
```

A Sample Test Case

- Let's think of a CalculatorService where it offers 4 methods like,
    - public CalculationResult add(Integer a, Integer b);
    - public CalculationResult subtract(Integer a, Integer b);
    - public CalculationResult multiply(Integer a, Integer b);
    - public CalculationResult divide(Integer a, Integer b);
- How would you test it?

Make sense?

```
public static void main(String[] args) {
    CalculatorService service = new CalculatorService();
    CalculationResult result;

    result = service.add(4, 5);
    if (result.getResult() == 9) System.out.println("Ekleme testi basarili");
    else System.out.println("Ekleme testi basarisiz!");

    result = service.subtract(6, 4);
    if (result.getResult() == 2) System.out.println("Cikarma testi basarili");
    else System.out.println("Cikarma testi basarisiz!");

    result = service.multiply(6, 4);
    if (result.getResult() == 24) System.out.println("Carpma testi basarili");
    else System.out.println("Carpma testi basarisiz!");

    result = service.divide(6, 6);
    if (result.getResult() == 1) System.out.println("Bolme testi basarili");
    else System.out.println("Bolme testi basarisiz!");
}
```

The power of unit testing

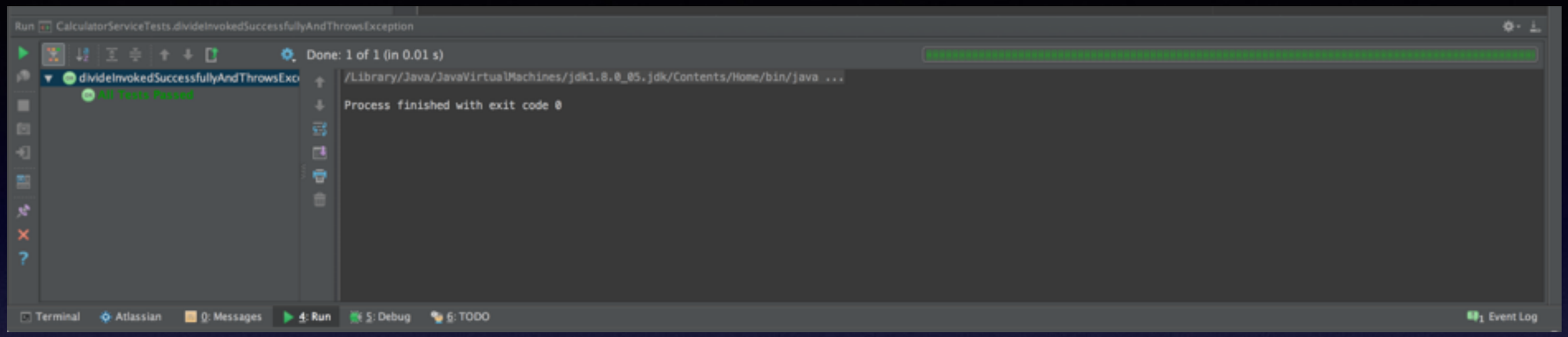
```java
public class CalculatorServiceTests {

    CalculatorService service = new CalculatorService();

    @Test
    public void addInvokedSuccessfullyAndReturnedProperCalculationResult() {
        CalculationResult result = service.add(4, 5);

        assertNotNull(result);
        assertEquals(result.getAction(), Action.ADD);
        assertEquals(result.getParam1(), Long.valueOf(4));
        assertEquals(result.getParam2(), Long.valueOf(5));
        assertEquals(result.getResult(), new Double(9));
    }

    @Test
    public void subtractInvokedSuccessfullyAndReturnedProperCalculationResult() {
        CalculationResult result = service.subtract(6, 4);

        assertNotNull(result);
        assertEquals(result.getAction(), Action.SUBTRACT);
        assertEquals(result.getParam1(), Long.valueOf(6));
        assertEquals(result.getParam2(), Long.valueOf(4));
        assertEquals(result.getResult(), new Double(2));
    }
```

The power of unit testing

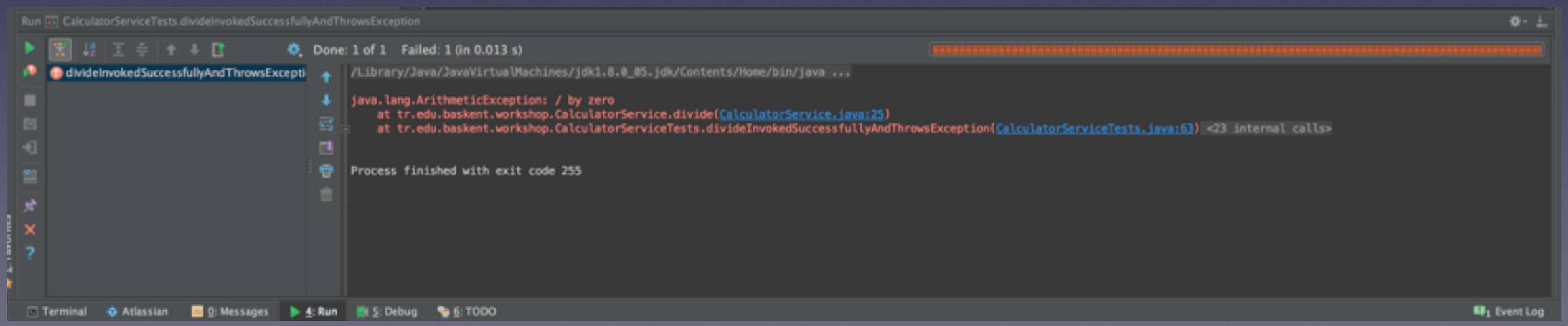
```
    @Test
    public void multiplyInvokedSuccessfullyAndReturnedProperCalculationResult() {
        CalculationResult result = service.multiply(6, 4);

        assertNotNull(result);
        assertEquals(result.getAction(), Action.MULTIPLY);
        assertEquals(result.getParam1(), Long.valueOf(6));
        assertEquals(result.getParam2(), Long.valueOf(4));
        assertEquals(result.getResult(), new Double(24));
    }
```

```
    @Test
    public void divideInvokedSuccessfullyAndReturnedProperCalculationResult() {
        CalculationResult result = service.divide(6, 6);

        assertNotNull(result);
        assertEquals(result.getAction(), Action.DIVIDE);
        assertEquals(result.getParam1(), Long.valueOf(6));
        assertEquals(result.getParam2(), Long.valueOf(6));
        assertEquals(result.getResult(), new Double(1));
    }

    @Test(expected = ArithmeticException.class)
    public void divideInvokedSuccessfullyAndThrowsException() {
        CalculationResult result = service.divide(6, 0);
    }
}
```

When tests passed



When tests failed



src:junit-basic

Hamcrest Matchers

- Hamcrest is a transitive dependency coming out with JUnit, which offers matchers for assertions and behaviour verifications.
- Matcher objects are also known as constraints or predicates.
- Hamcrest is not a testing library!
- A simple example would be:

```
assertThat(workshop, is(goingGood()));
```

Anatomy of an expression

actual value

- assertThat(lecture, is(goingGood()));

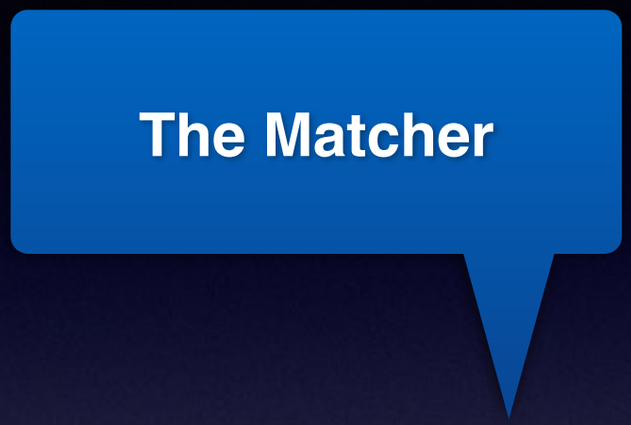
Matcher, which represents the expectation on the value.

Alternatives

- assertEquals can also be used right? like, assertEquals(theNameOfTheGreatestTeamInTheWorld, “Galatasaray”);
- But what about collections? assertEquals(myTeam, europe.getSoccerTeams().iterator().next());
- This will work if my team is the first element in the list. But what if, it resides in any place of that list?
- We can do it like:

```
boolean found = false;
for (Team team : europe.getSoccerTeams())  {
    if (team.equals(myTeam)) found = true;
}
assertTrue(found);
```

The Matcher Way




```
assertThat(europe.getSoccerTeams(), hasItem(myTeam));
```

instead of,

```
boolean found = false;
for (Team team : europe.getSoccerTeams())  {
    if (team.equals(myTeam)) found = true;
}
assertTrue(found);
```

These matcher methods are coming from org.hamcrest.Matchers class.

equalTo Matcher

```
public static <T> Matcher<T> equalTo(T operand) {
    return IsEqual.equalTo(operand);
}
```

```
public class IsEqual<T> extends BaseMatcher<T> {
  @Factory
  public static <T> Matcher<T> equalTo(T operand) {
      return new IsEqual<T>(operand);
  }
}
```

assertion syntax

- People are usually declare the assertion as,

```
assertEquals(result.getAction(), Action.MULTIPLY);
```

- But it's also possible to re-write this line as,

```
assertThat(result.getAction(), is(Action.MULTIPLY));
```

- which makes it more readable in english.
- Have you noticed anything different in the second line?

static imports

- We, as test coders, tend to use static imports wherever and whenever possible.
- The is() method used in the previous slide, is imported as a “static import”.
- It could be written as CoreMatchers.is() but this is not that readable, and will lead to code pollution.

Test Suite

- You can create a suite to group test classes. When you run the test suite class, it will run all the tests.

```
package tr.edu.hacettepe.bbm490;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        JunitTest1.class,
        JunitTest2.class
})

public class JunitTestSuite {
}
```

- Using *Suite* as a runner allows you to manually build a suite containing tests from many classes.

To ignore a Test

- In practice you shouldn't be doing this or at least you shouldn't be committing&pushing code with ignored test methods.
- But if you must somehow, you can do it with @Ignore annotation.
- Methods annotated with @Test that are also annotated with @Ignore will not be executed as tests.
- Also, you can annotate a class containing test methods with @Ignore and none of the containing tests will be executed.

Testing with Spring

The power of Spring Testing

- IoC makes application code feasible for unit testing.
- Just create mock objects and set them into the objects that you want to test.
- The rule of thumb is:
  no environmental dependency!
- no DB, appServer, network, or even the IoC container itself.

The power of Spring Testing

- Doing only the unit testing won't make any sense, since it'll be like,
- test tires, engine, and doors of a car separately, and you assume that everything will work as expected when you assemble all those parts together and form the car out of them.
- We need to bring some parts of the whole system together and try to see if they will work.
- Those parts in a software system usually include objects from several layers, existing transactional context and database, network interaction or the security context.

Our aim is to do the testing without deploying and running the whole application on an application server.

Introducing new Dependencies

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-test</artifactId>
   <version>4.1.6.RELEASE</version>
   <scope>test</scope>
</dependency>

<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
   <scope>test</scope>
</dependency>
```

Sometimes we might need Junit like…

```
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
      <exclusions>
         <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-all</artifactId>
      <version>1.3</version>
   </dependency>
</dependencies>
```

Spring TestContext

- Spring offers the TestContext framework which works either with Junit or TestNG (we'll stick with JUnit).
- It helps,
  - with the creation of Spring Container in test classes
  - helps testing ORM related codes in an existing transactional context
  - helps to test web functionality without deploying the application code into a web container

src:junit-spring

SpringJUnit4ClassRunner

- The *Spring TestContext Framework* offers full integration with JUnit 4.5+ through a custom runner. This helps us on handling of the,
  - loading of application contexts
  - dependency injection with beans
  - transactional test method execution

```
@RunWith(SpringJUnit4ClassRunner.class)
```

@ContextConfiguration

- defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests.
- locations attribute defines the locations of the configuration files. multiple files can be provided with the attribute.

```
@ContextConfiguration(locations = {"classpath*:/applicationContext.xml"})
```

@TransactionConfiguration

- defines class-level metadata for configuring transactional tests.
- annotation provides 2 attributes:
  - transactionManager: the name of the manager bean that'll handle transactions.
  - defaultRollback: states that if transactions should be rollbacked by default. It's default value is true. We also need to annotate the class with @Transactional to handle the rollback.

```
@TransactionConfiguration
```

Base Integration Test Case

- All these annotations need to be declared for each test class. (Against the DRY principle, right)
- We can create a base abstract class, that would contain all these annotations like,

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:/applicationContext.xml"})
@TransactionConfiguration
@Transactional
public abstract class BaseIntegrationTestCase {
}
```

Integrating Tests w/ Maven

- To run the tests within Maven context, we need to define maven-surefire-plugin. During the test phase, this plugin executes the unit tests of the application.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${maven.surefire.plugin.version}</version>
    <configuration>
        <runOrder>alphabetical</runOrder>
        <includes>
           <include>**/*Test.java</include>
        </includes>
    </configuration>
</plugin>
```

- Definition given above runs all the @Test methods of all the classes that are suffixed with Test.