Com S 417
Software Testing

Fall 2017 – Week 7, Lecture 12

Announcements

- Chapter 4 O&A is available on digital reserve.
- We will probably only have 4 projects (not 5).

Topics

- In Class exercise
- Introduction to Mockito
- TDD (Test Driven Development)

The Mockito Concepts

How would you draw the class diagram?

- (At least) two interfaces: the mocked interface and the mockito mock control interface.
  - Mocked interface includes all methods known to the mocked object.
  - SUT will call these methods directly:
    - Particular particularObj = mock(Particular.class); //static
      - @Mock annotation does same.
    - particularObj.myParticularMethod(); //'normal' member
  - Test code will "message about" these methods by calling them as scoped members of the mockito control interface:
    - when(particularObj.myParticularMethod().thenReturn(true))

    Notice the dotted chain.

Mockito Concepts

Implicit argument verification

- ‘when’ is an expectation:

```
//add the behavior to add numbers
when(calcService.add(20.0,10.0)).thenReturn(30.0);

//subtract the behavior to subtract numbers
when(calcService.subtract(20.0,10.0)).thenReturn(10.0);
```

- Unexpected calls or calls with unexpected parameters cause an exception.
- The form above uses a ‘default’ argument checker. You can supply more flexible or more precise argument validation.

Explicit Verification

- Check (after the fact) that the mock object received a particular call (at any point in the test):

```
//verify call to calcService is made or not with same arguments.
verify(calcService).add(10.0, 20.0);
```

- Require an explicit order to the calls:
  - Create a verifier object. Think of it as a “template” or recording of a sequential expectation.

```
//create an inOrder verifier for a single mock
InOrder inOrder = inOrder(calcService);

//following will make sure that add is first called then subtract is called.
inOrder.verify(calcService).add(20.0,10.0);
inOrder.verify(calcService).subtract(20.0,10.0);
```

listings from:
https://www.tutorialspoint.com/mockito/mockito_ordered_verification.htm

Pre-defined verifiers

- Accessed through static methods on the mock control interface.

  - **atLeast (int min)** – expects min calls.
  - **atLeastOnce ()** – expects at least one call.
  - **atMost (int max)** – expects max calls.

Example Usage

```
//@Mock annotation is used to create the mock object to be injected
@Mock
CalculatorService calcService;
```

```
  //check a minimum 1 call count
  verify(calcService, atLeastOnce()).subtract(20.0, 10.0);

  //check if add function is called minimum 2 times
  verify(calcService, atLeast(2)).add(10.0, 20.0);

  //check if add function is called maximum 3 times
  verify(calcService, atMost(3)).add(10.0,20.0);
```

TutorialPoint example
What's Wrong?

- First example is not automated; doesn't use junit.
- Main() for test. Yech!
- No separation of test and production code.
- 'Sort of' uses dependency injection, but not consistently.
- What about additional test cases? what will they cost to setup and run?
- PortfolioManager isn't.
- Overkill for simple replacement.
- What is this?

```
//Create a portfolio object which is to be tested
StockService portfolio = new StockService();
```

Not the point!

```
public class PortfolioTester {
   public static void main(String[] args){

      //Create a portfolio object which is to be tested
      Portfolio portfolio = new Portfolio();

      //Creates a list of stocks to be added to the portfolio
      List<Stock> stocks = new ArrayList<Stock>();
      Stock googleStock = new Stock("1","Google", 10);
      Stock microsoftStock = new Stock("2","Microsoft",100);

      stocks.add(googleStock);
      stocks.add(microsoftStock);

      //Create the mock object of stock service
      StockService stockServiceMock = mock(StockService.class);

      // mock the behavior of stock service to return the value of various stocks
      when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);
      when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);
```

```
        //add stocks to the portfolio
        portfolio.setStocks(stocks);

        //set the stockService to the portfolio
        portfolio.setStockService(stockServiceMock);

        double marketValue = portfolio.getMarketValue();

        //verify the market value to be
        //10*50.00 + 100* 1000.00 = 500.00 + 100000.00 = 100500
        System.out.println("Market value of the portfolio: "+ marketValue);
    }
}
```

A more realistic example

Production Quality Test Code

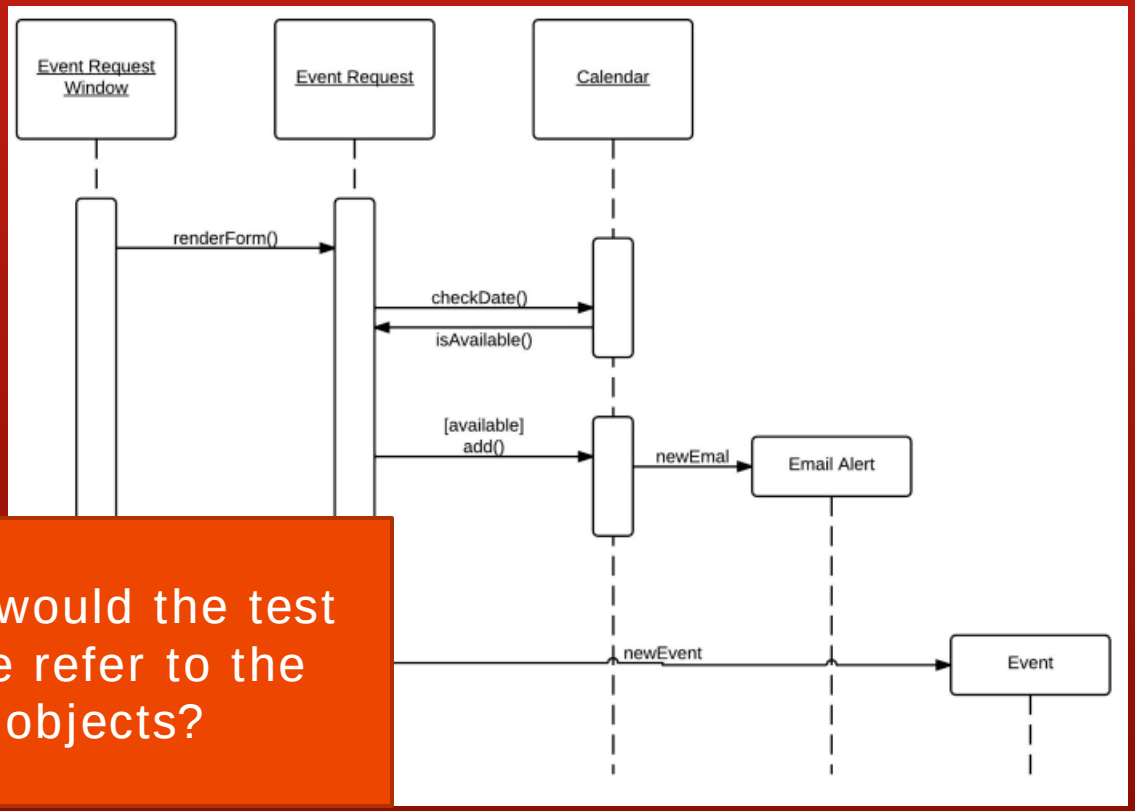
- Use production worthy classes.
- Separate construction from operation.
- Use dependency injection to configure any dependency that might change – especially during test.
- When there is lots of setup, try to make the setup reusable.
- There are two items to mock for full testing.

Advantages

- Easy to extend for additional tests.
  - With one more layer of files, additional tests could be added without any programming skills.
- Can easily modify to test with real portfolio and mock stock service.

So why mess with Mockito?

- Without equal when you need to test that the messages in a protocol occurred in the correct order. Think about validating a sequence diagram.



How would the test code refer to the objects?

For testing convenience, you can put all five interfaces in the same mock!

Not a code smell when done for the right reason.

```
Foo mock = Mockito.mock(Foo.class, withSettings().extraInterfaces(Bar.class));
```

The wrong RunWith?

```
// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(MockitoJUnitRunner.class)
public class MathApplicationTester {

   //@InjectMocks annotation is used to create and inject the mock object
   @InjectMocks
   MathApplication mathApplication = new MathApplication();

   //@Mock annotation is used to create the mock object to be injected
   @Mock
   CalculatorService calcService;

   @Test
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      when(calcService.add(10.0,20.0)).thenReturn(30.00);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
   }
}
```

- This shortens the code, but we are still hard coding each test case!

From the javadoc

- Mockito JUnit Runner keeps tests clean and improves debugging experience. Make sure to try out MockitoJUnitRunner.StrictStubs which automatically detects **stubbing argument mismatches** and is planned to be the default in Mockito v3. JUnit Runner is compatible with JUnit 4.4 and higher and adds following behavior: (new since Mockito 2.1.0) Detects unused stubs in the test code. See UnnecessaryStubbingException. Similar to JUnit rules, the runner also reports stubbing argument mismatches as console warnings (see MockitoHint). To opt-out from this feature, use @RunWith(MockitoJUnitRunner.Silent.class)
- Initializes mocks annotated with Mock, so that explicit usage of MockitoAnnotations.initMocks(Object) is not necessary. Mocks are initialized before each test method.
- Validates framework usage after each test method. See javadoc for Mockito.validateMockitoUsage().

TDD: The Discipline

The Three Rules of TDD:

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test

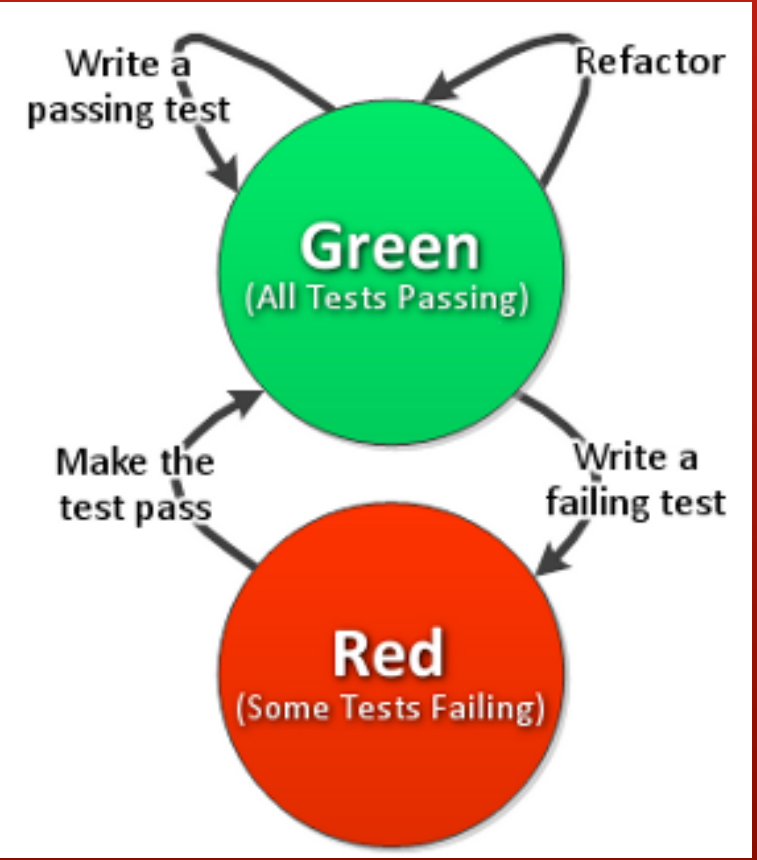


Figure from: http://www.peterprovost.org/blog/2012/05/02/kata-the-only-way-to-learn-tdd/
Rules from http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

The FizzBuzz Code Kata

https://www.youtube.com/watch?v=JyRouDwzCoo

- Code Kata
  - Code Katas are to programming as Compulsory Figures are to figure skating.
  - https://www.youtube.com/watch?v=n2LwMId43uU
  - A Code Kata is an exercise designed to build and demonstrate technical precision, accuracy, and skill. For more, see CodeKata.com

Note: watch the lower left corner of the screen for the test result when the camera zooms out.

Writing code == writing tests

No handoffs. No deferred activities. Seamless integration.