Mockito

CS585 Software Verification and Validation
http://cs585.yusun.io
January 14, 2015

Yu Sun, Ph.D.
http://yusun.io
yusun@cpp.edu

CAL POLY POMONA

Tricky Unit Testing Scenarios

- A test acquires a connection from the database that fetches/ updates data
- It connects to the Internet and downloads files
- It interacts with an SMTP server to send e-mails
- It performs I/O operations
- …

Qualities of Unit Testing

- Order independent and isolated
  - *ATest.java* should not be dependent on the output of the *BTest.java* test class
  - *when_an_user_is_deleted_the_associated_id_gets_deleted()* should not be dependent on the order of

    when_a_new_user_is_created_an_id_is_returned()

Qualities of Unit Testing

- Trouble-free setup and run
  - Unit tests should not require a DB connection or an Internet connection or a clean-up temp directory

Qualities of Unit Testing

- Effortless execution
  - Unit tests should run fine on all computers, not just on a specific computer

Qualities of Unit Testing

- Formula 1 Execution
  - A test should not take more than a second to finish the execution

Mock an Object

- mock()

```
MarketWatcher marketWatcher = Mockito.mock(MarketWatcher.class);
Portfolio portfolio = Mockito.mock(Portfolio.class);
```

- Mock annotation

```
@Mock
MarketWatcher marketWatcher;

@Mock
Portfolio portfolio;

@Before
public void setUp() {
  MockitoAnnotations.initMocks(this);
}
```

Stub a Method

- The stubbing process defines the behavior of a mock method such as the value to be returned or the exception to be thrown when the method is invoked

```
@Test
public void marketWatcher_Returns_current_stock_status() {
    Stock uvsityCorp = new Stock("UV", "Uvsity Corporation", new
BigDecimal("100.00"));

    when(marketWatcher.getQuote(anyString())).
      thenReturn(uvsityCorp);

    assertNotNull(marketWatcher.getQuote("UV"));
}
```

Stub a Method

- Stub Options:
  - thenReturn(x): This returns the x value
  - thenThrow(x): This throws an x exception
  - thenAnswer(Answer answer): Unlike returning a hardcoded value, a dynamic user-defined logic is executed. It's more like for fake test doubles, Answer is an interface
  - thenCallRealMethod(): This method calls the real method on the mock object

Stub a Method

```
@RunWith(MockitoJUnitRunner.class)
public class StockBrokerTest {

  @Test
  public void when_ten_percent_gain_then_the_stock_is_sold() {
    //Portfolio's getAvgPrice is stubbed to return $10.00
    when(portfolio.getAvgPrice(isA(Stock.class))).
      thenReturn(new BigDecimal("10.00"));
    //A stock object is created with current price $11.20
    Stock aCorp = new Stock("A", "A Corp", new BigDecimal("11.20"));
    //getQuote method is stubbed to return the stock
    when(marketWatcher.getQuote(anyString())).thenReturn(aCorp);
    //perform method is called, as the stock price increases
    // by 12% the broker should sell the stocks
    broker.perform(portfolio, aCorp);

    //verifying that the broker sold the stocks
    verify(portfolio).sell(aCorp,10);
 }
}
```

Verify

- The *verify* method verifies the invocation of mock objects

Verify

- times(int wantedNumberOfInvocations)
  - This method is invoked exactly n times; if the method is not invoked wantedNumberOfInvocations times, then the test fails.
- never()
  - This method signifies that the stubbed method is never called or you can use times(0) to represent the same scenario. If the stubbed method is invoked at least once, then the test fails.
- atLeastOnce()
  - This method is invoked at least once, and it works fine if it is invoked multiple times. However, the operation fails if the method is not invoked.

Verify

- atLeast(int minNumberOfInvocations)
  - This method is called at least n times, and it works fine if the method is invoked more than the minNumberOfInvocations times. However, the operation fails if the method is not called minNumberOfInvocations times.
- atMost(int maxNumberOfInvocations)
  - This method is called at the most n times. However, the operation fails if the method is called more than minNumberOfInvocations times.

Verify

- only()
  - The only method called on a mock fails if any other method is called on the mock object.
- timeout(int millis)
  - This method is interacted in a specified time range.

Verify

- verifyZeroInteractions(Object... mocks)
  - verifies whether no interactions happened on the given mocks
- verifyNoMoreInteractions(Object... mocks)
  - checks whether any of the given mocks has any unverified interaction

```
@Test
public void verify_no_more_interaction() {
    Stock noStock = null;
    portfolio.getAvgPrice(noStock);
    portfolio.sell(null, 0);
    verify(portfolio).getAvgPrice(eq(noStock));
    //this will fail as the sell method was invoked
    verifyNoMoreInteractions(portfolio);
}
```

Verify with Arguments Matcher

```
when(mockObject.getValue(1)).thenReturn(expected value);
```

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
```

```
verify(mock).someMethod(1, anyString(), "third argument");
```

```
class BlueChipStockMatcher extends ArgumentMatcher<String>{
    @Override
    public boolean matches(Object symbol) {
       return "FB".equals(symbol) ||  "AAPL".equals(symbol);
    }
}
```

Throw Exceptions

```
@Test(expected = IllegalStateException.class)
public void throwsException() throws Exception {
    when(portfolio.getAvgPrice(isA(Stock.class))).thenThrow(new
IllegalStateException("Database down"));
    portfolio.getAvgPrice(new Stock(null, null, null));
}
```

```
doThrow(exception).when(mock).voidmethod(arguments);
```

Stubbing Consecutive Calls

```
  @Test
  public void consecutive_calls() throws Exception {
    Stock stock = new Stock(null, null, null);

    when(portfolio.getAvgPrice(stock)).thenReturn(BigDecimal.TEN,
BigDecimal.ZERO);

    assertEquals(BigDecimal.TEN, portfolio.getAvgPrice(stock));
    assertEquals(BigDecimal.ZERO, portfolio.getAvgPrice(stock));
    assertEquals(BigDecimal.ZERO, portfolio.getAvgPrice(stock));
  }
```

```
when(portfolio.getAvgPrice(stock))
.thenReturn(BigDecimal.TEN)
.thenReturn(BigDecimal.TEN)
.thenThrow(new IllegalStateException())
```

Stubbing with an Answer

- Stubbed methods return a hardcoded value but cannot return an on the fly result. *Answer* provides the callbacks to compute the on the fly results.

```
class TotalPriceAnswer implements Answer<BigDecimal>{
  @Override
  public BigDecimal answer(InvocationOnMock invocation) throws Throwable  {
    BigDecimal totalPrice = BigDecimal.ZERO;

    for(String stockId: stockMap.keySet()) {
      for(Stock stock:stockMap.get(stockId)) {
        totalPrice = totalPrice.add(stock.getPrice());
      }
    }
    return totalPrice;
  }
}
```

Stubbing void Method

- The doReturn() method is is used only when

  when(mock).thenReturn(return) cannot be used

- The when-thenReturn method is more readable than doReturn(); also, doReturn() is not a safe type. The thenReturn method checks the method return types and raises a compilation error if an unsafe type is passed

```
doThrow(new RuntimeException()).
doNothing().
doAnswer(someAnswer).
when(mock).someVoidMethod();
```

Verifying the Invocation Order

```
@Test public void inorder() throws Exception {
  Stock aCorp = new Stock("A", "A Corp", new BigDecimal(11.20));
  portfolio.getAvgPrice(aCorp);
  portfolio.getCurrentValue();
  marketWatcher.getQuote("X");
  portfolio.buy(aCorp);
  InOrder inOrder = inOrder(portfolio,marketWatcher);
  inOrder.verify(portfolio).buy(isA(Stock.class));
  inOrder.verify(portfolio).getAvgPrice(isA(Stock.class));
}
```

Capturing Arguments with ArgumentCaptor

- ArgumentCaptor provides an API to access objects that are instantiated within the method under the test

```
public void buildPerson(String firstName,
          String lastName, String middleName, int age){
  Person person = new Person();
  person.setFirstName(firstName);
  person.setMiddleName(middleName);
  person.setLastName(lastName);
  person.setAge(age);
  this.personService.save(person);
}
```

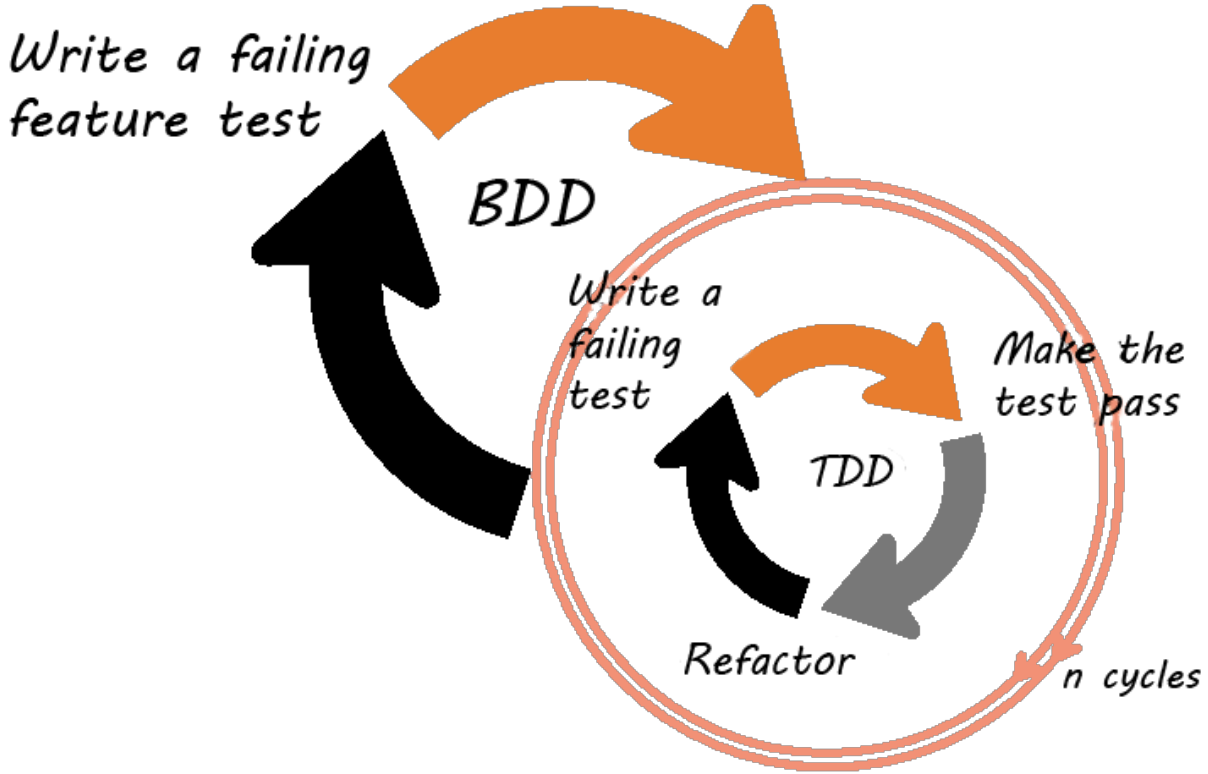
Capturing Arguments with ArgumentCaptor

```
@Test
public void argument_captor() throws Exception {
  when(portfolio.getAvgPrice(isA(Stock.class))).thenReturn(new BigDecimal("10.00"));
  Stock aCorp = new Stock("A", "A Corp", new BigDecimal(11.20));
  when(marketWatcher.getQuote(anyString())).thenReturn(aCorp);
  broker.perform(portfolio, aCorp);

  ArgumentCaptor<String> stockIdCaptor = ArgumentCaptor.forClass(String.class);

  verify(marketWatcher).getQuote(stockIdCaptor.capture());
  assertEquals("A", stockIdCaptor.getValue());

  //Two arguments captured
  ArgumentCaptor<Stock>  stockCaptor = ArgumentCaptor.forClass(Stock.class);
  ArgumentCaptor<Integer> stockSellCountCaptor = ArgumentCaptor.forClass(Integer.class);

  verify(portfolio).sell(stockCaptor.capture(), stockSellCountCaptor.capture());
  assertEquals("A", stockCaptor.getValue().getSymbol());
  assertEquals(10, stockSellCountCaptor.getValue().intValue());
}
```

Behavior-Driven Development (BDD)

- Unit test names should start with the word *should* and should be written in the order of the business value
- Acceptance tests (AT) should be written in a user story manner, such as "As a (role) I want (feature) so that (benefit)"
- Acceptance criteria should be written in terms of scenarios and implemented as "Given (initial context), when (event occurs), then (ensure some outcomes)"

Behavior-Driven Development (BDD)



Spying Objects (Partial Mock)

```
Stock realStock = new Stock("A", "Company A", BigDecimal.ONE);
Stock spyStock = spy(realStock);
//call real method from  spy
assertEquals("A", spyStock.getSymbol());

//Changing value using spy
spyStock.updatePrice(BigDecimal.ZERO);

//verify spy has the changed value
assertEquals(BigDecimal.ZERO, spyStock.getPrice());

//Stubbing method
when(spyStock.getPrice()).thenReturn(BigDecimal.TEN);

//Changing value using spy
spyStock.updatePrice(new BigDecimal("7"));

//Stubbed method value 10.00  is returned NOT 7
assertNotEquals(new BigDecimal("7"), spyStock.getPrice());
//Stubbed method value 10.00
assertEquals(BigDecimal.TEN,  spyStock.getPrice());
```