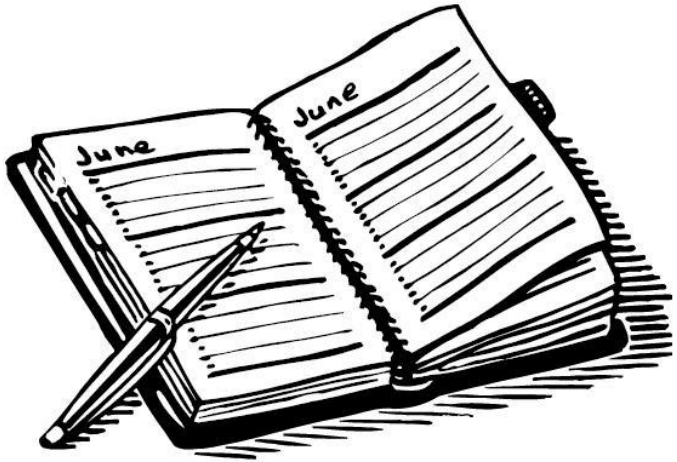
A Simple, Intuitive Mocking Framework

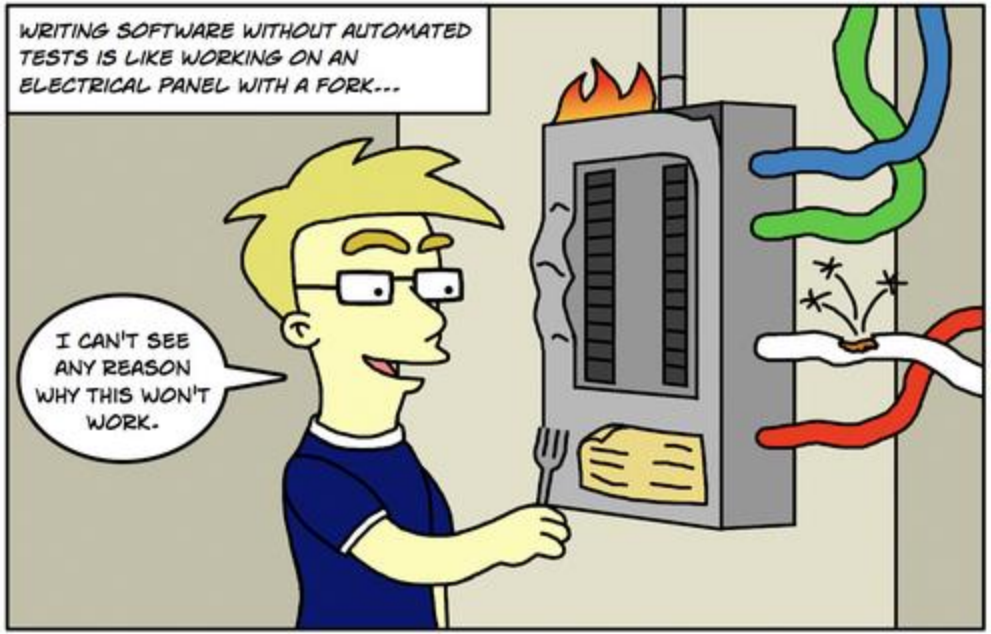
AXON ACTIVE
www.axon.vn

Agenda

- Refresh the importance of unit testing
- Mock framework
- **Mock with Mockito API**
  - Configuring Mockito in a Project
  - Creating Mock
  - Stubbing Method's Returned Value
  - Argument Matching
  - And more...
- Conclusion and Q&A



Refresh : the importance of Unit testing

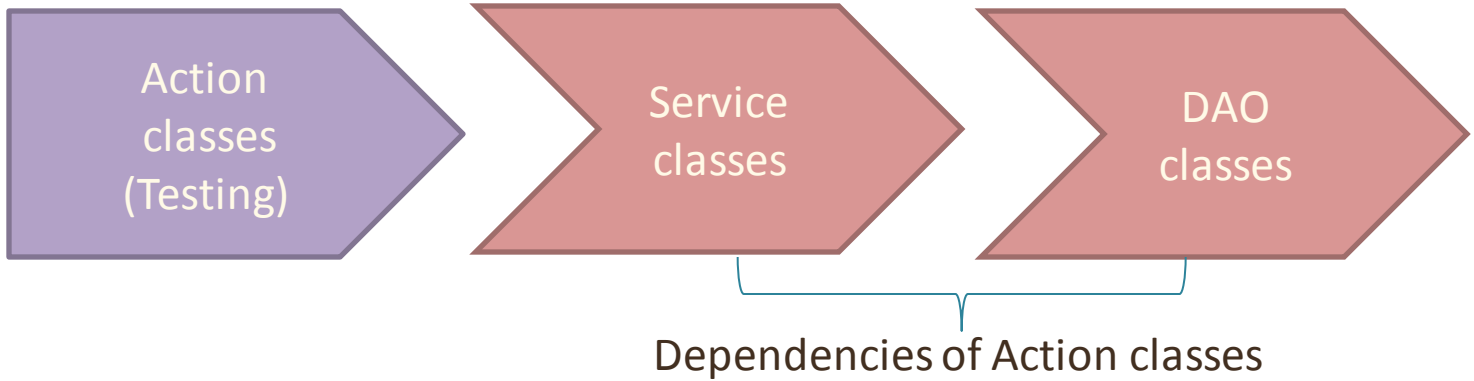


Refresh : the importance of Unit testing

- Help to ensure that code **meets requirements**
- Could be considered a form of **documentations** as to how the system currently operates.
- Help to ensure that **changes** made by the enhancement **do not break** the existing system.
- Encourage **refactoring**.
- Integrate with **Continuous Integration**
- Improve **design**

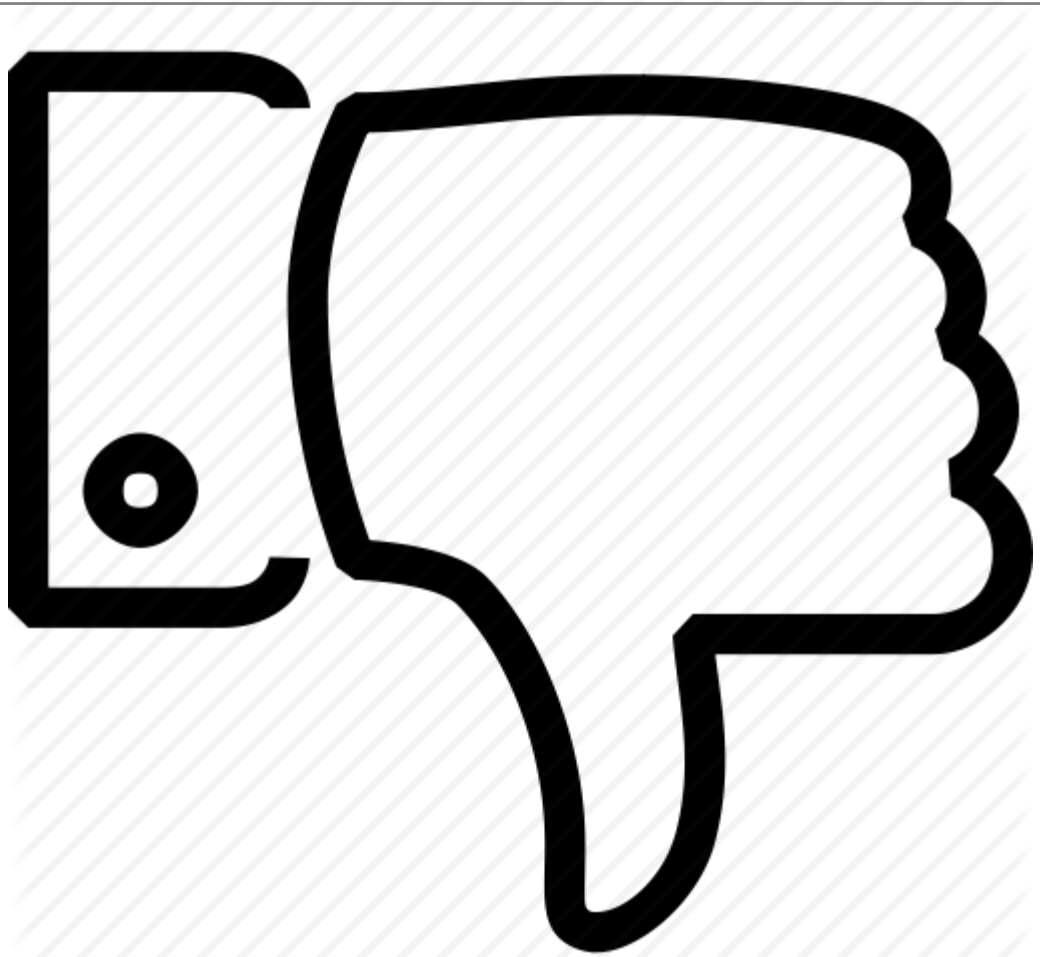
Problem…

- The '**hello world**' has **no dependencies** on outside classes.
- Software **has dependencies**.
- The dependencies have **not implemented** yet.
- The idea of UT : **test our code** without testing the dependencies



How I ‘solve’ the problem before ?

- **Didn’t** write unit testing for my code ☹
- Dependencies have been **implemented**
  - Invoke dependencies **directly** in unit test.
  - It’s not unit test anymore (**integration test** instead)
- Dependencies have **NOT** been implemented yet.
  - **Wait** for some guys implementing the dependencies first.
  - **Hard code** return value of the dependencies.



HOW

MOCK FRAMEWORK

“ Do One Thing and Do It Well “

Dan North

What mock is ?

In one sentence

"Mocks are object that simulates the behavior of real an object."

"break dependencies of your classes"

Why mock ?

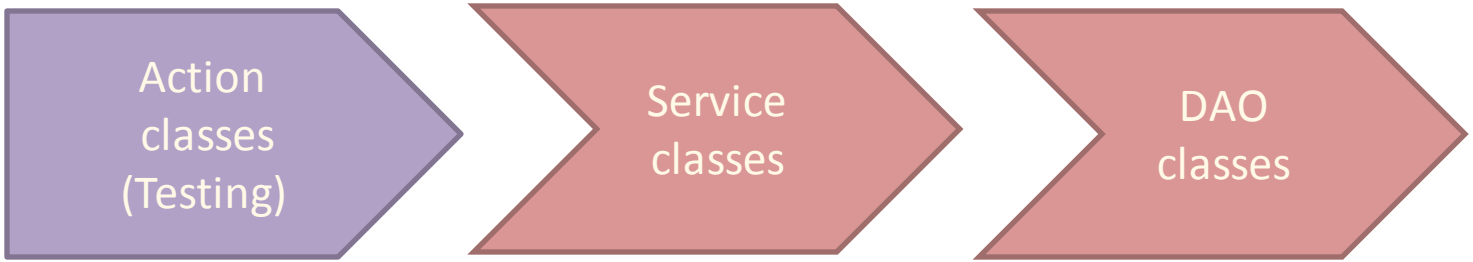
- **Break** dependencies.
- **Parallel** working.
- Do not need to modify code to "**hard code**" returning value of dependencies.
- Run unit testing **faster**.
- Improve **code design**.
- Have **fun** in unit testing. ☺

When I can use mock?

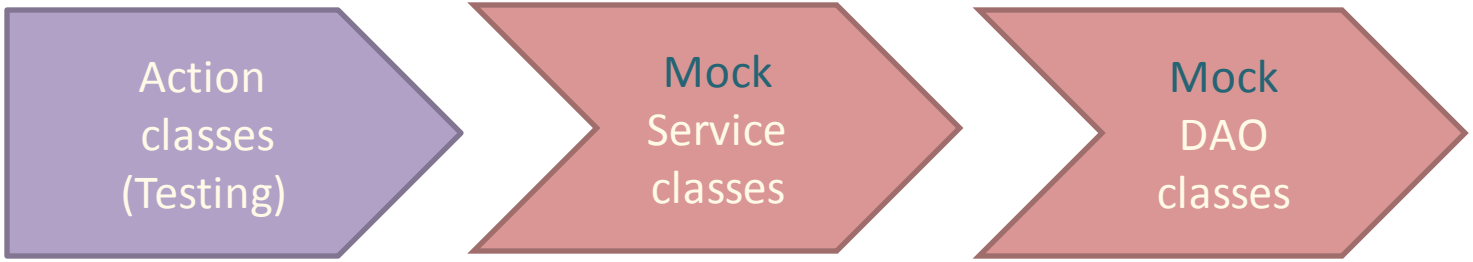
- The real object is
  - Difficult to setup
  - Has behavior that is hard to trigger
  - Slow
  - User interface
- Some cases
  - Dependencies are sensitive resources in real code.
  - Charge money on API calls / API limit.
  - Database accessing / Network interaction.

Mock model

Without mock



With mock



REAL SYSTEM

Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

CLASS IN UNIT TEST

Green = class in focus
Yellow = mocks for the unit test

Mocks API (Java)

EASYMOCK

jMock

What is the selected ?

In my opinion is

Why ?

- Easy to use
- Simple & clean syntax
- Excellent documentation

See more on SO

*"Mockito really **cleans up** the unit test by not requiring expectations. Personally, I much prefer the Mockito API to the EasyMock API for that reason." - Hamlet D'Arcy*

Install Mockito

Maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.17</version>
</dependency>
```

Gradle

```
'org.mockito:mockito-all:1.10.17'
```

In your code, import

```
import static org.mockito.Mockito.*;
```

Given-When-Then

Given-When-Then is a style of representing tests, part of BDD

- The **given** : describes the **state** of the world before you begin the behavior. You can think of it as the **pre-conditions** to the test.
- The **when** : is that **behavior** that you're specifying.
- The **then** : describes the changes you **expect** due to the specified behavior.

Method stub

- A stub may **simulate** the behavior of existing code (Wikipedia)
- What's the difference between faking, mocking, and stubbing? (view on SO)

Mockito - keywords

- **mock**(ClassToMock.class)
- **when**(methodCall)
- **thenReturn**(value)
- doReturn(value)
- thenThrow(Throwable)
- verify(mock).method(args)
- initMocks(this), @Mock, @Spy

Without mock


```
@Test
public void test() throws Exception {
  // Given
  TestingObject testingObj = new TestingObject();
  DependencyObject dependencyObj =
                            new DependencyObject();

  // When
  testingObject.doSomeThing(helper);

  // Then
  assertTrue(helper.somethingHappened());
}
```

Mockito "Template" Usage


```
@Test
public void test() throws Exception {
  // Given (prepare behavior)
  TestingObject testingObj = new TestingObject();
  DependencyObject mockObj =
                      mock(DependencyObject.class);
  when(mockObj.domesomething()).thenReturn("A");

  // When
  testingObject.doSomeThing(helper);

  // Then (verify)
  assertTrue(helper.somethingHappened());
}
```

- *You can use Mockito to create mocks of a regular (not final)* ***class*** *not only of an* ***interface****.*
- *Methods (include their arguments) that was* ***not stubbed****, then return* ***null*** *when invoked. (See here)*

That's quite enough !

Let's drink

Argument matchers


```
when(mockObject.dosomeThing(anyInt()).thenReturn(true);
```

- Mockito offer some built-in such as: *anyString(), anyInt(), any(Class<T> clazz), isNull*, …
- Custom argument matchers (extends *ArgumentMatcher*)

If you are using argument matchers, **all arguments** have to be provided by matchers.

ArgumentCaptor

In some situations, it is helpful to assert on certain arguments after the actual verification

```
ArgumentCaptor<Person> argument =
                    ArgumentCaptor.forClass(Person.class);
verify(mockObj).doSomething(argument.capture());
assertEquals("John", argument.getValue().getName());
```

In more details…

Stubbing multiple calls to the same method

Return different values for subsequent calls of the same method.

```
@Test
public void shouldReturnLastDefinedValueConsistently() {
    WaterSource waterSource = mock(WaterSource.class);

    when(waterSource.getWaterPressure()).thenReturn(3, 5);

    assertEquals(waterSource.getWaterPressure(), 3);
    assertEquals(waterSource.getWaterPressure(), 5);
    assertEquals(waterSource.getWaterPressure(), 5);
}
```

Stubbing void methods

- The stubbed method is passed as a **parameter** to a given / when method.
- So, cannot use this construct for **void** methods.
- Instead, you should use **willXXX..given** or **doXXX..when**

```
@Test(expectedExceptions = RuntimeException.class)
public void shouldStubVoidMethod() {
   doThrow(new RuntimeException()).when(mockedList).clear();
   //following throws RuntimeException:
   mockedList.clear();
}
```

Spying on real objects

- Some cases, **impossible** to test with **pure mocks**
  - Legacy code.
  - IoC containers
- "**partial mocking**" concept
  - Instead of mocking, calling **directly to real** object.
- Usually, there is **no reason** to spy on real objects
  - Code smell

Example of spying on real objects


```
List spy = spy(new LinkedList());

when(spy.size()).thenReturn(100); // size() is stubbed.

spy.add("one"); // Assume that this's legacy code.

assertEquals("one", spy.get(0)); // *real* method
assertEquals(100, spy.size());   // *stubbed* method

//optionally, you can verify
verify(spy).add("one");
```

Important gotcha

- Sometimes it's impossible to use **when..thenReturn** for stubbing spies.
- Consider **doReturn..when**

```
List spy = spy(new LinkedList());

// the list is yet empty -> IndexOutOfBoundsException
when(spy.get(0)).thenReturn("foo");

//You have to use doReturn() for stubbing
doReturn("foo").when(spy).get(0);
```

Annotations

- **@Mock**, **@Spy** : simplify the process of creating relevant objects using static methods.
- **@InjectMocks** : simplifies mock and spy **injection**
  - MockitoJUnit4Runner as a JUnit runner
  - Or, MockitoAnnotations.initMocks(testClass)
- Helpful in Spring DI

Annotations


```
public class OrderService {
  PriceService priceService;
  OrderDao orderDao;
}
```

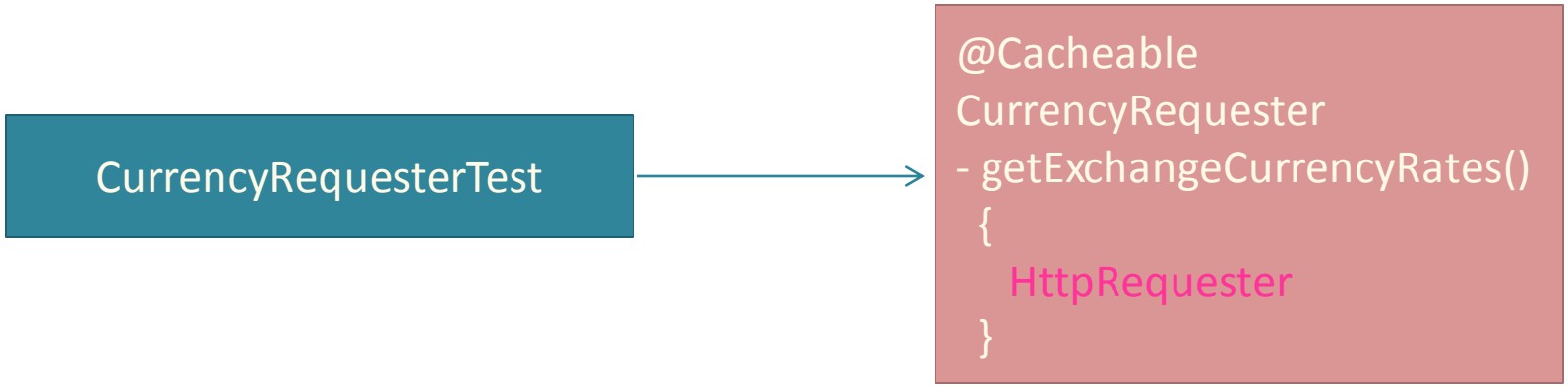
```
@RunWith(MockitoJUnitRunner.class)
public class MockInjectingTest {
 @Mock
 PriceService mockPriceService;
 @Spy
 OrderDao spyOrderDao;
 @InjectMocks
 OrderService testOrderService;
}
```

Some common verifying actions

- Verifying exact number of invocations / at least x / never

```
LinkedList mockedList = mock(LinkedList.class);
mockedList.add("once");

mockedList.add("twice");
mockedList.add("twice");

verify(mockedList, times(1)).add("once");

//exact number of invocations verification
verify(mockedList, times(2)).add("twice");

verify(mockedList, never()).add("never happened");
verify(mockedList, atLeastOnce()).add("twice");
```

Case : how to test 'cache' instance ?



Case : how to test 'cache' instance ?

```
HttpRequester mockRequester = mock(HttpRequester.class)

when(httpRequester.makeHttpGetRequest(anyString())).
thenReturn(result);

currencyRequester.getExchangeCurrencyRates();
currencyRequester.getExchangeCurrencyRates();

verify(httpRequester,times(0)).makeHttpGetRequest(anyString());
```

Limitations

Mockito can not :

- mock final classes
- mock enums
- mock final methods
- mock static methods
- mock private methods
- mock hashCode() and equals()

PowerMock or JMockit can be used to work with code that cannot be mocked with pure Mockito

Conclusion

Mock framework

Mockito
1.10.17

Dependencies

Method stub

mock
when..thenReturn

Unit testing

spy
'partial mocking'

thenThrow

Argument matchers

Annotations

Limitations

verify

Clean & clear

Write code testable

Unit testing is testing units.

Units bare without dependencies.

And this can be done with mocks

References

- https://code.google.com/p/mockito/
- http://refcardz.dzone.com/refcardz/mockito
- http://www.slideshare.net/fwendt/using-mockito

What will you do ?

- Try to google “mock test framwork”, “mockito”…
- Visit mockito home page
- Write yourself some code with dependencies, then try to test them with mock framework.
- Try to find another mock API with your programming language.
- Check out : https://bitbucket.org/phatvu/mockito-learning
  - Sample example about Mockito with Java.
  - Sample example about Mockito with Spring.

http://goo.gl/ePWvmA