♦ 5 Java unit testing interview Questions and Answers

Posted on October 8, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1 Why use mock objects in unit testing?

A1 Unit testing is widely accepted as a “best practice” for software development. When you write an object, you must also provide an automated test class containing methods by calling its various public methods with various parameters and making sure that the values returned are appropriate.

When you’re dealing with simple data or service objects, writing unit tests is straightforward. However, in reality the object under test rely on other objects or layers of infrastructure, and it is often expensive, impractical, or inefficient to instantiate these collaborators.

For example, to unit test an object that uses a database, it may be burdensome to install a local copy of the database, run your tests, then tear the local database down again. Mock objects provide a way out of this dilemma. A mock object conforms to the interface of the real object, but has just enough code to simulate the tested object and track its behavior. For example, a database connection for a particular unit test might record the query while always returning the same hard coded result. As long as the class being tested behaves as expected, it won't notice the difference, and the unit test can check that the proper query was emitted.

Here are some reasons why mock objects are handy:

- The unit tests as the name implies must test only a unit of the code and not all its collaborating dependencies. You only have to worry about the class under test. Mock objects allow you to achieve this by mocking external resource and coding dependencies. The example in the next question demonstrates how we can mock reading from a file, which is an external resource.
- The unit tests need to test for the proper boundary conditions. For example, positive values, negative values, zero value, etc. The mock object make your life easier for mimicking these boundary conditions.
- One of the biggest mistake one can make in writing quality unit tests is to have state dependencies between unit tests. The unit tests must be able to run in any order. The mock objects will help you isolate these state dependencies, and make your tests isolated and independent.For example
  - Test the DAO in isolation by mocking the calls to external resources like database, file, etc.
  - Test your service in isolation by mocking the calls to your DAO.

Having said this, too much mocking can make your code hard to read and understand. So, it is important to have the right balance without overdoing.

16 Technical Key Areas

open all | close all

Q2 How would you go about using mock objects in your unit tests?

A2

Step 1: Have the relevant dependencies required to write unit tests.

```
<properties>
  <junit.version>4.8.1</junit.version>
  <mockito.version>1.8.5</mockito.version>
  <powermock.version>1.4.8</powermock.version>
</properties>

<dependencyManagement>
  <dependencies>
        ...
      <dependency>
           <groupId>junit</groupId>
           <artifactId>junit</artifactId>
           <version>${junit.version}</version>
           <scope>test</scope>
      </dependency>
      <dependency>
           <groupId>org.mockito</groupId>
           <artifactId>mockito-all</artifactId>
           <version>${mockito.version}</version>
           <scope>test</scope>
      </dependency>
      <dependency>
           <groupId>org.powermock</groupId>
           <artifactId>powermock-module-junit4</
           <version>${powermock.version}</versio
           <scope>test</scope>
      </dependency>
      <dependency>
           <groupId>org.powermock</groupId>
           <artifactId>powermock-api-mockito</ar
           <version>${powermock.version}</versio
           <scope>test</scope>
      </dependency>
  ...
  </dependencies>
<dependencyManagement>
```

Step 2: The UserDaoImpl is an implementation of the interface UserDao. The implementation read the user names from a text file “users.txt”.

```
Peter Smith
Aaron Lachlan
Zara John
Felix Chan
```

The “UserDao.java” interface

```
1 import java.util.List;
2
3 public interface UserDao {
4       public List<string> readUsers() throws Users
5 }
```

The “UserDaoImpl.java” class that reads from the “user.txt” file implements the interface “UserDao.java”.

```
1  import java.io.InputStream;
2  import java.util.Arrays;
3  import java.util.Collections;
4  import java.util.List;
5  import java.util.Scanner;
6
7  public class UserDaoImpl implements UserDao {
8
9       private static final String DELIMITER = Syst
10
11      public UserDaoImpl(){}
12
13      @Override
14      public List<string> readUsers() throws Users
15
16          InputStream is = getResource();
17
18          if (is == null) {
19              throw new UsersException("users file
20          }
21
22          Scanner sc = new Scanner(is);
23          String value = sc.useDelimiter(DELIMITER
24
25          String[] users = value.split(DELIMITER);
26
27          return (users == null || users.length >
28                  .asList(users) : Collections.<st
29      }
30
31      private InputStream getResource()  {
32          ClassLoader cl = Thread.currentThread()
33          InputStream is = cl.getResourceAsStream(
34          return is;
35      }
36 }
```

STEP 3: Finally the unit test that uses the Mockito framework to mock the actual loading of the user names from text file. The user names will be supplied via the method getDummyIs(). The UserDaoImpl is partially mocked with the spy method. This means the getResource() method is mocked by supplying some dummy data within the test itself. The readUsers() method is executed from the class under test, which is UserDaoImpl. The getResource() method is

How good are your .....?

mocked to return a user name of “John Patrick” evey time it is invoked.

```
1  import java.io.ByteArrayInputStream;
2  import java.io.InputStream;
3  import java.util.List;
4  import junit.framework.Assert;
5  import org.junit.Test;
6  import org.junit.runner.RunWith;
7  import org.powermock.api.mockito.PowerMockito;
8  import org.powermock.core.classloader.annotation
9  import org.powermock.modules.junit4.PowerMockRun
10
11 @RunWith(PowerMockRunner.class)
12 @PrepareForTest(UserDaoImpl.class)
13 public class UserDaoWithMockTest {
14
15     @Test
16     public void testGetUsers() throws Exception
17         final UserDao partiallyMockedUserDao = P
18         PowerMockito.doReturn(getDummyIs()).when
19         List<string> users = partiallyMockedUser
20         Assert.assertEquals(1, users.size());
21     }
22
23     @Test(expected = unittest.UsersException.cla
24     public void testGetUsersNegative() throws Ex
25         final UserDao partiallyMockedUserDao = P
26         PowerMockito.doReturn(null).when(partial
27         partiallyMockedUserDao.readUsers();
28     }
29
30     @Test(expected = unittest.UsersException.cla
31     public void testGetUsers2() throws Exception
32         final UserDao partiallyMockedUserDao = P
33         PowerMockito.doReturn(new ByteArrayInput
34         List<string> users = partiallyMockedUser
35         Assert.assertEquals(0, users.size());
36     }
37
38     public InputStream getDummyIs() {
39         String str = "John Patrick";
40         return new ByteArrayInputStream(str.getB
41     }
42 }
```

Q3 What mocking frameworks have you used?

A3 Mockito, EasyMock, and PowerMock.

PowerMock is a framework that extends other mock libraries such as EasyMock and Mockito with more powerful capabilities like mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers and more.

Q4 What is the difference between a mock object and a stub?

A4 The key difference to note is the ability of the mock objects to verify if a particular method was invoked and if yes, how many times was invoked. This is demonstrated with the last two lines with the **verify** statement. This is a common Java interview question quizzing the candidate's understanding of the difference between a mock object and stub.

```
1  //The test case
2      @Test
3      public void testGetPositionFeedCSV() throws
4      {
5          String str = "dummyCSV";
6          //Set up behavior
7          when(mockMyAppService.getPositionFeedCSV
8          when(response.getWriter()).thenReturn(wr
9
10         //Invoke controller
11         controller.getPositionFeedCSV(PORTFOLIO_
12
13         //Verify behavior
14         verify(mockMyAppService, times(1)).getPo
15         verify(writer, times(1)).write(any(Strin
16     }
17
```

Q5 What is BDD?

A5 **BDD** is principally an idea about how software development should be managed by both business interests and technical insight. Test-driven development focuses on the developer's opinion on how parts of the software should work. Behavior-driven development focuses on the users' opinion on how they want your application to behave. So, when you start writing a test, you need to think about the stories, and each story should cover three things:

Given : an input value of 2
When : you multiply the input with 3
Then : result should be 6

Even you write unit tests as part of TDD (Test Driven Development) or without TDD , you need to think about Given … When … Then …

Here is a simple example using the **jBehave** framework in Java.

Step 1: Maven pom.xml file on jBehave dependency

```
<dependency>
      <groupId>org.jbehave</groupId>
      <artifactId>jbehave-core</artifactId>
      <version>3.8</version>
</dependency>
```

Step 2: Define the story in plain English that business users and testers can understand using **Given… When Then…** style. The “math.story” file under “src/main/resources/jbehave” folder

```
Scenario: 2 squared

Given a variable input with value 2
When I multiply input by 2
Then result should equal 4

Scenario: 3 squared

Given a variable input with value 3
When I multiply input by 3
Then result should equal 9
```

Step 3: Map the above scenarios based stories to Java equivalent.

```
import org.jbehave.core.annotations.Given;
import org.jbehave.core.annotations.Named;
import org.jbehave.core.annotations.Then;
import org.jbehave.core.annotations.When;
import org.jbehave.core.steps.Steps;

public class MathSteps extends Steps
{
    private int input;
    private int result;

    @Given("a variable input with value $value")
    public void givenInputValue(@Named("value")
    {
        input = value;
    }

    @When("I multiply input by $value")
    public void whenImultiplyInputBy(@Named("val
    {
        result = input * value;
```

```
22      }
23
24      @Then("result should equal $value")
25      public void thenInputshouldBe(@Named("value"
26      {
27          if (value != result) {
28              throw new RuntimeException("result i
29          }
30      }
31  }
```

Step 4: Write a main class to excute the scenarios.

```
1  import java.util.Arrays;
2  import java.util.List;
3  import org.jbehave.core.embedder.Embedder;
4
5  public class JBehaveTest
6  {
7      private static Embedder embedder = new Embed
8      private static List<String> storyPaths = Arr
9
10     public static void main(String[] args)
11     {
12         embedder.candidateSteps().add(new MathSt
13         try
14         {
15             embedder.runStoriesAsPaths(storyPath
16         }
17         catch (Exception e)
18         {
19             e.printStackTrace();
20         }
21
22     }
23 }
```

Behaviour-Driven Development (**BDD**) is an evolution in the thinking behind Test Driven Development (**TDD** — Writing tests before writing code) and Acceptance Test Driven Development (**ATDD** — write acceptnce tests, and for many agile teams, acceptance tests are the main form of functional specification and the formal expression of the business requirements). The BDD basically combines **TDD** and **Domain Driven Design**. It aims to provide **common vocabulary** that can be used between business and technology.

Popular Posts

♦ 11 Spring boot interview questions & answers

828 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

768 views

18 Java scenarios based interview Questions and Answers

400 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

389 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

296 views

♦ 7 Java debugging interview questions & answers

293 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

286 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

279 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

240 views

001B: ♦ Java architecture & design concepts interview questions & answers

202 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ Auditing, data retention and archiving interview Q&A

10 Core Java Best Practices with an industry strength code sample ›

Posted in Best Practice, Code Quality Job Interview Essentials, member-paid, Unit Testing Q&A

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer

 ↑