Unit Testing

Devdatta Kulkarni

Unit Testing

- How to ensure that a ``unit of code'' is working as expected?
- In your assignment1, how did you know that you had accounted for all the specified and non-specified combinations of the query parameters?

Unit Testing

- Concepts
  - Code under test
  - Dependencies
- Question that we ask?
  - When the dependencies behave as specified, does the code under test behave as we expect it to behave?

Unit Testing: Example 1

- What does testing the ‘doGet’ method mean?
  - It means verifying that:
    a) When the method is called with instances of HttpServletRequest and HttpServletResponse objects as input parameters, the HttpServletResponse’s Printwriter’s println() method is passed the “Hello world.” string
    b) HttpServletResponse’s getWriter() method is called

Unit Testing: Example 2

- What does testing the ‘doGet2’ method mean?
  - It means verifying that:

  a) When “username” query parameter is passed in, the response is of the form “Hello <name>”

  b) When “username” is null or empty

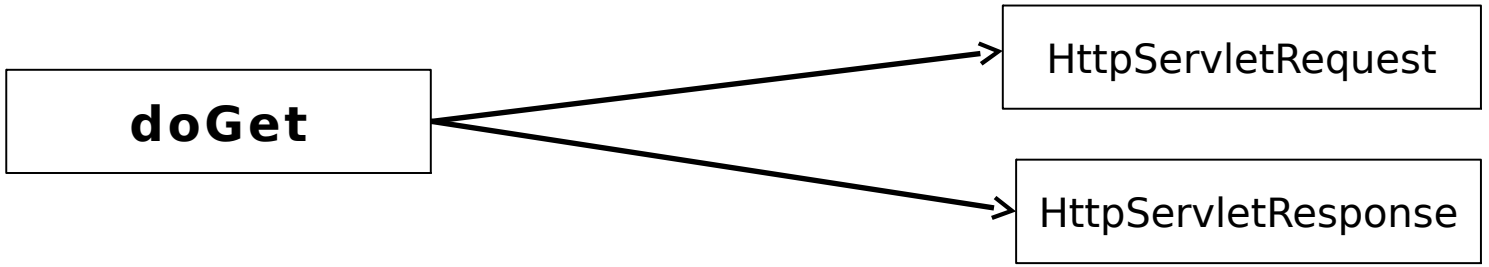
  These are two separate test cases

Unit Testing: Stages

1) Decompose your application into small units
2) Control the values that are generated by any dependencies
   - This is called setting up ``expectations'' for the code under test
3) Invoke the code under test
4) Verify that the result is what you are expecting

Unit Testing: Expectation setting and verification

- So, we need to set expectations on the dependencies
  - How to do it?



How do we make the HttpServletRequest object return the values that we control?

Enter
Mocks

Mocks

- Mock
  - A ``mock'' is an object that can be used for testing purposes in place of the real dependency object
  - A mock object provides mechanisms to:
    - set the values that would be returned when a method is called on the object
    - verify that a method was called on an object
    - verify that a method was called given number of times on an object

Unit Testing Library

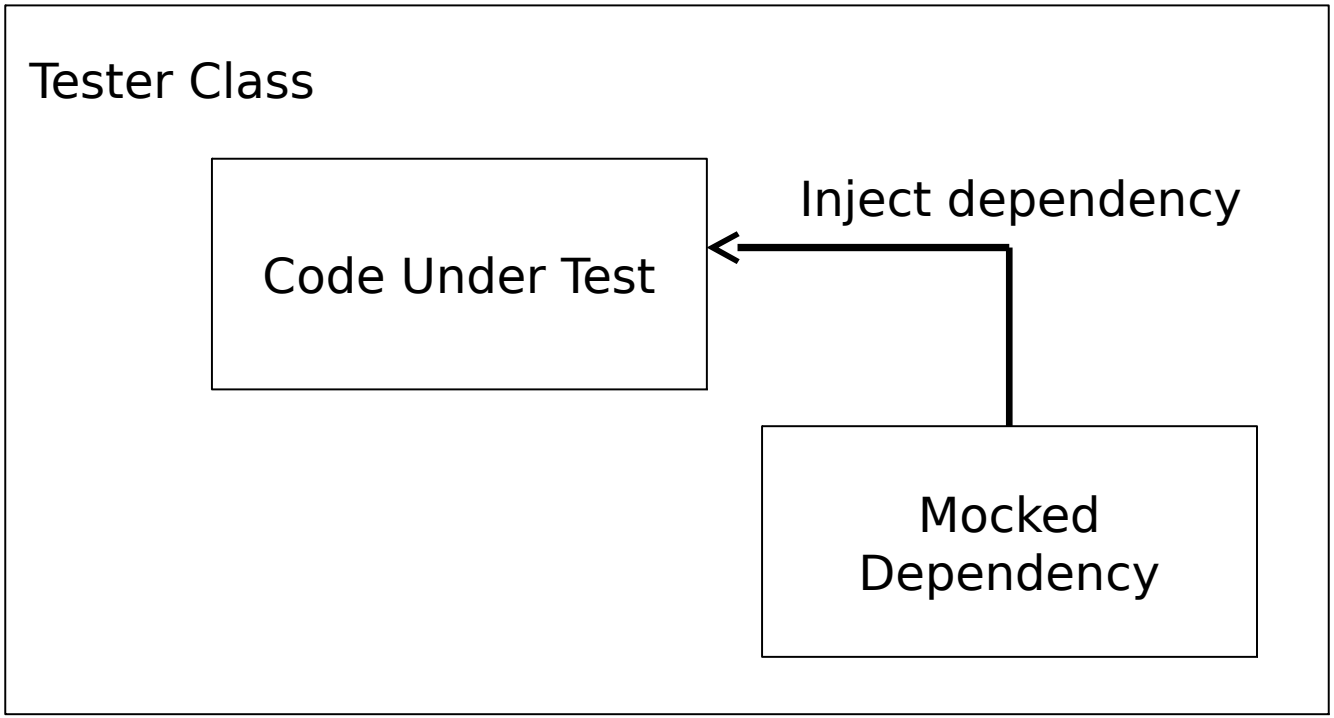
- Junit (http://junit.org/)

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
</dependency>
```

- Mockito (https://code.google.com/p/mockito/)

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-all</artifactId>
<version>1.9.5</version>
</dependency>
```

Unit Testing



Unit testing examples

- https://github.com/devdattakulkarni/ModernWebApps.git
  - Servlets/servlet-unit-test
  - Servlets/mockito-example
- Running tests:
  - Right click on src/test/java in the "Project Explorer" view
  - Select "Run As" -> Junit test

Code organization for Unit testing

- Typical code structure:

  src/main/java/*.java

  src/test/java/*.java

- Example:

  src/main/java/HelloServlet.java

  src/test/java/TestHelloServlet.java

JUnit Annotations

- @Before
  - This annotation identifies the method that runs *before* running any test method
    - Conventionally this method is named as *setUp*
- @Test
  - This annotation identifies a test method.

Unit testing steps

- Step 1: Identify *code under test*
- Step 2: Identify dependencies in code under test
- Step 3: Create mock dependencies
- Step 4: Set up expectations
- Step 5: Invoke *code under test*
- Step 6: Assertions and verifications
  - Assert state
  - Verify behavior

Step 1: Identify code under test

Let us test the following method in doGet method in HelloServlet:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.getWriter().println("Hello world.");
}
```

Step 2: Identify dependencies in code under test

- What are the dependencies of *doGet()* method?

- HttpServletRequest and HttpServletResponse objects

Step 3: Create mock dependencies

HttpServletRequest request = *mock(HttpServletRequest.class);*

Structure:

<Type> mockedObject = *mock(*Type.class)

Step 4: Set up expectations

PrintWriter writer = *mock(PrintWriter.class);*

when(response.getWriter()).thenReturn(writer);

Structure:

*when(*mockedObject.method()).*thenReturn(*value under our control*)*

Step 5: Call code under test

```
helloServlet.doGet(request, response);
```

Step 6: Verify interaction

verify(writer).println("Hello world.");
verify(response).getWriter();

Structure:
*verify(*mockedObject).methodName();

Step 6: Verify state

Pattern:

assertEquals(expected, actual)

Code under test and dependent objects

How to add dependency objects in the code under test?

- If we create dependencies on-the-fly, we won't be able to mock them and control their behavior
- But we need temporary objects all the time, which we will end up creating at runtime (on-the-fly)

So then what to do?

Code under test and dependent objects

Thumb rule:

- Don’t create application-level domain objects such as controller classes, service classes, and so on as part of a method’s execution
- Create them outside of any method
  - Create them inside constructor, or init method (for Servlets), or let the *container create them and inject into your class*

Points to remember

- Mockito does not allow mocking of *Final* classes
  - Cannot Mock URL class
- Cannot call private methods from the test class
  - Need to make methods either public or protected

References

- Static imports:
  - https://docs.oracle.com/javase/1.5.0/docs/guide/language/static-import.html
  - http://monkeyisland.pl/2008/04/26/asking-and-telling/
- Types of test objects:
  - http://www.javaworld.com/article/2074508/core-java/mocks-and-stubs---understanding-test-doubles-with-mockito.html