TESTING

Unit

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing involves only those characteristics that are vital to the performance of the unit under test.

Models / components

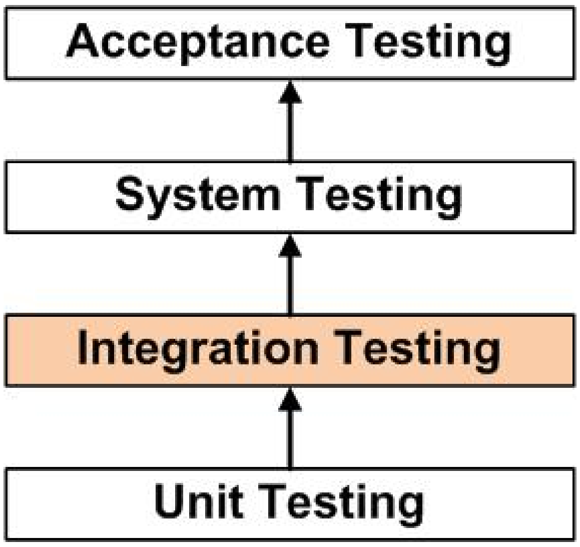
Difference between unit and component test: The basic difference between the two is that in unit testing, all the methods of other classes and modules are mocked. On the other hand, for component testing, all stubs and simulators are replaced with the real objects for all the classes (units) of that component, and mocking is used for classes of other components.

Models of testing:

1. Waterfall model
2. V Model
3. Agile Model
4. Spiral Model
5. RAD Model

Integration / endpoint

Integration testing is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration Testing.



End to end

End-to-end testing is a methodology used to test whether the flow of an application is performing as designed from start to finish. The purpose of carrying out end-to-end tests is to identify system dependencies and to ensure that the right information is passed between various system components.

For example, a simplified end-to-end testing of an email application might involve:

- Logging in to the application
- Accessing the inbox
- Opening and closing the mailbox
- Composing, forwarding or replying to email
- Checking the sent items
- Logging out of the application

Assertions

An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program. A test assertion is defined as an expression, which encapsulates some testable logic specified about a target under test.

Benefits of Assertions:

- It is used to detect subtle errors which might go unnoticed.
- It is used to detect errors sooner after they occur.
- Make a statement about the effects of the code that is guaranteed to be true.

```
@Test
  void fibonacciTesterforIndexSix() {
    assertEquals(8, Fibonacci.fibonacci(6) );
  }
```

Types of assertions

Checking the value of something
assertTrue
assertFalse
assertNull
assertNotNull
assertEqual
assertNotEqual
assertIdentical
assertNotIdentical

Mocking

Mocking is creating an object that mimics the behavior of another object. It's a strategy for isolating an object to test it and verify its behavior.

```
public void checkIfTokenIsProvided() throws Exception {
    mockMvc.perform(get("/pitches")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isUnauthorized())
        .andExpect(jsonPath("$.error").value("unauthorized"));
  }
```

Mocking (mock, stub, spy)

Spock is a Java testing framework capable of handling the full life cycle of a computer program.

Stub - is concerned with simulating specific behaviour. In Spock this is all a stub can do, so it is kind of the simplest thing.

Mock - is concerned with standing in for a (possibly expensive) real object, providing no-op answers for all method calls. In this regard, a mock is simpler than a stub. But in Spock, a mock can also stub method results, i.e. be both a mock and a stub. Furthermore, in Spock we can count how often specific mock methods with certain parameters have been called during a test.

Spy - always wraps a real object and by default routes all method calls to the original object, also passing through the original results. Method call counting also works for spies. In Spock, a spy can also modify the behaviour of the original object, manipulating method call parameters and/or results or blocking the original methods from being called at all.

Expected skills (4/5)

- **Able to explain the reason for testing**

Reasons to test software:

1. Software testing saves money
2. Security
3. Product quality
4. Customer satisfaction

- **Able to explain the difference between unit and integration tests**

(See above)

- **Able to create different test scenarios / test cases**

```
@Test
  public void pitchBadgeMissingContentTypeCheckStatus() throws Exception {
    this.mockMvc.perform(post("/pitch")
        .header("userTokenAuth", elements.getValidToken())
        .content(stringify(elements.getValidPitchDto()))
        .andExpect(status().isUnsupportedMediaType());
  }

  @Test
  public void getPitchesWithCorrectHeader() throws Exception {
    mockMvc.perform(get("/pitches")
        .header("userTokenAuth", token)
```

```
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().string(stringify(pitchSetDTO)))
                .andReturn();
        }
```

- **Able to explain the need for mocking**

Mocking objects provide you with the ability to test what you write without having to address dependency concerns.

- **Able to show different types of assertions**

(See above)