Unit Testing - A comprehensive guide

by Vlad Ungureanu

LEARN stuff IO

Agenda

Unit Testing
A comprehensive guide

- What is unit testing ?
- Why use unit testing ?
- Simple JUnits
- Basic Annotations
- Assertions
- Running tests with Configuration
- Mocks
- Mock Tests
- MVC Tests
- In memory DB tests
- Component tests
- TDD
- TDD and Acceptance TDD

The Evolution of Debugging

Dec. 1950
Before Debugger

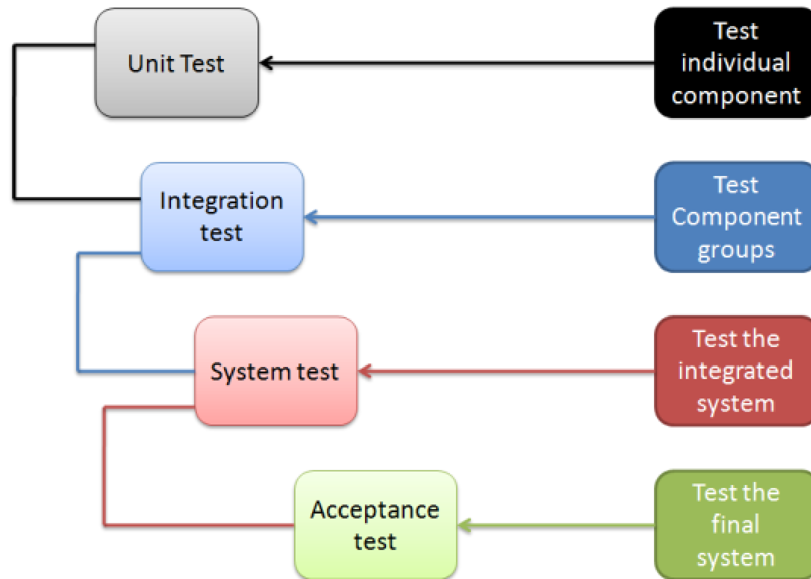
Dec. 1980
After Debugger

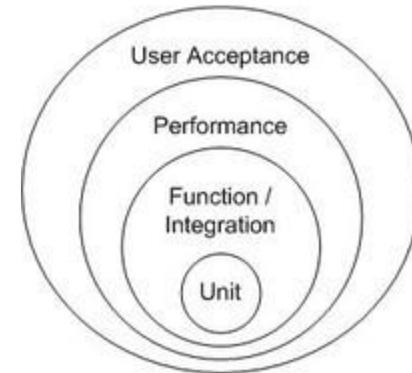
Dec. 2000
Before Easy Unit Testing

Dec. 2011
After Easy Unit Testing

LEARN stuff IO

Testing





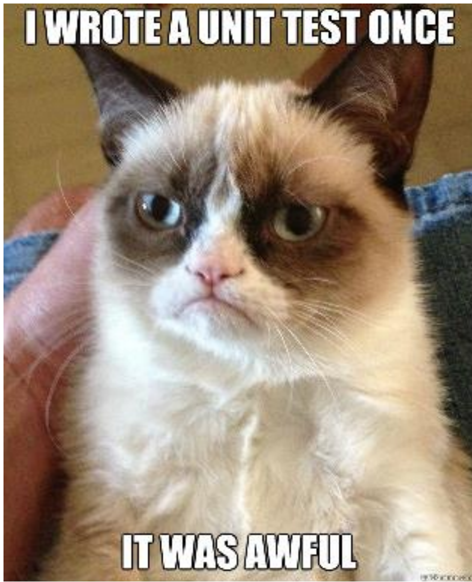
What is Unit Testing ?

- 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 and behavior.

- Unit tests are low-level, focusing on a small part of the software system

- Object-oriented design tends to treat a class as the unit

- Testing a unit means making sure that the unit performs the expected behavior in a *predictable*, *repeatable* and *isolated* way


I WROTE A UNIT TEST ONCE
IT WAS AWFUL

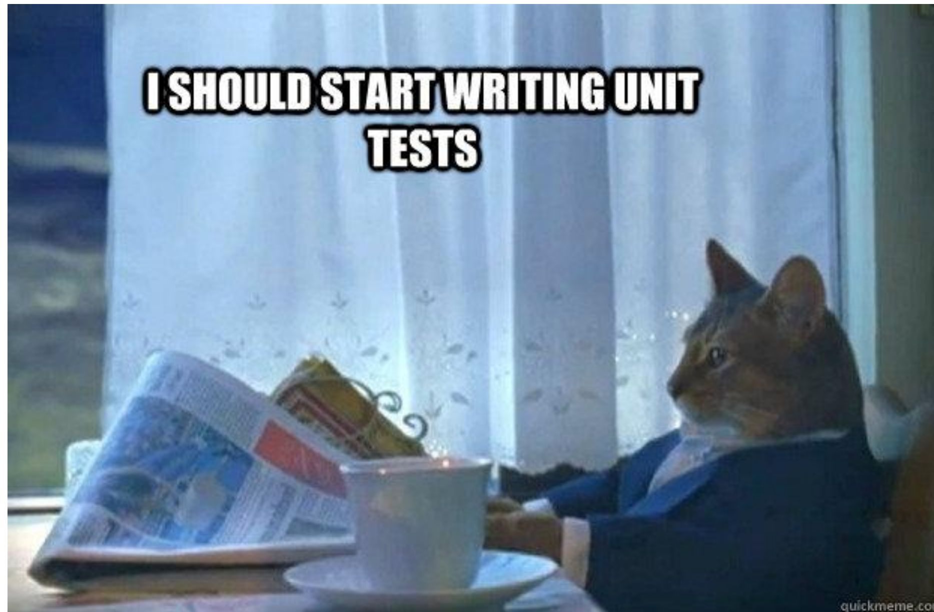
LEARN stuff IO

Why use unit testing?

12 Reasons to help make up your mind! *

- Reduce bugs in old and new features
- Can be used as documentation
- Reduce cost of change
- Improve Design
- Allow refactoring
- Constrain features
- Protect against other programmers
- Faster Development
- Forces you to think
- Reduce stress
- Increase client confidence in your code
- Mandatory when bringing changes to delivered code

- *Some say it is very fun…


I SHOULD START WRITING UNIT TESTS
quickmeme.com

Simple JUnits

```
@Test
public void multiplicationOfIntegersShouldReturnIntegerValue() {
        // MyClass.multiply is tested
        MyClass tester = new MyClass();

        // Test
        assertEquals("10 x 2 must be 20", 20, tester.multiply(10, 2));
      }
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
        // MyClass.multiply is tested
        MyClass tester = new MyClass();

        // Test
        assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

Basic Annotations

@Test public void method()	The @Test annotation identifies a method as a test method.
@Test (expected = **Exception.class**) public void method()	Fails if the method does not throw the named exception.
@Test(timeout=100) public void method()	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment.
@After public void method()	This method is executed after each test. It is used to cleanup the test environment. It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Needs to be static.
@AfterClass public static void method()	This method is executed once, after all tests have finished. It is used to perform clean-up activities, for example, to disconnect from a database. Needs to be static.
@Ignore	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.

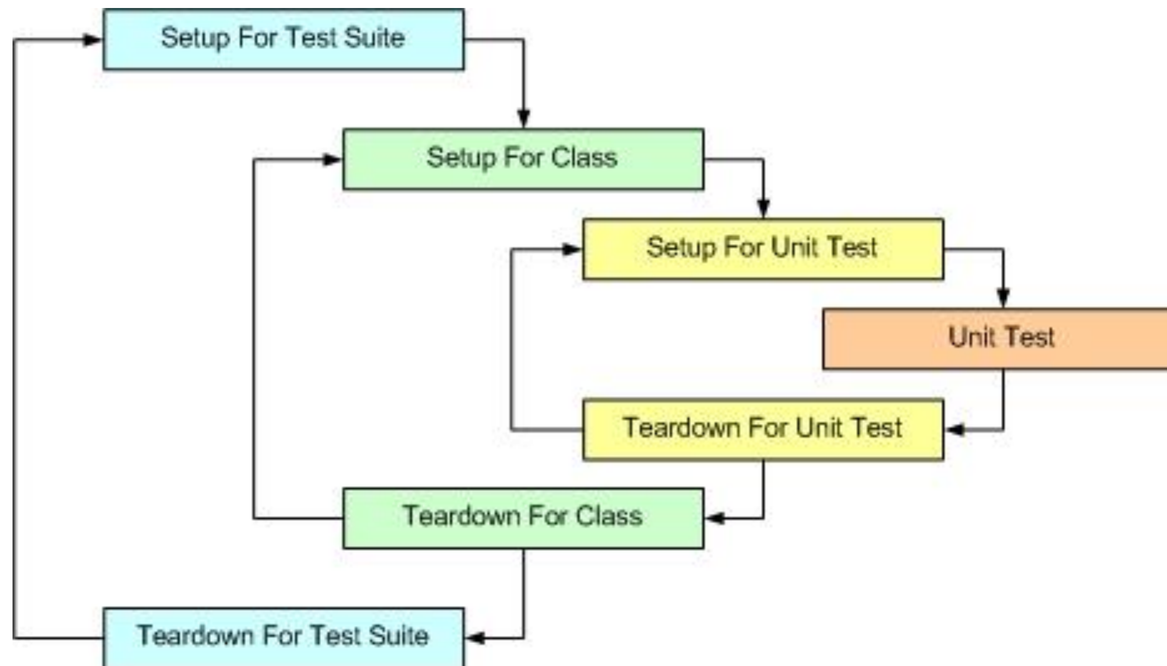
Assertions

fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The String parameter is optional.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertFalse([message], boolean condition)	Checks that the boolean condition is false.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays and other objects the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables do not refer to the same object.

Running tests with Configuration

```
// Standard JUnit runner
@RunWith(SpringJUnit4ClassRunner.class)

// JUnit runner with access to Spring Context
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"classpath:hs.xml","classpath:root-context.xml"})

// JUnit runner with access to we configuration and Spring context
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(locations = { "classpath:application-context.xml" })

// JUnit runner with access to PowerMock features like mocking static methods
@RunWith(PowerMockRunner.class)
```

Unit Testing – Behind the Scenes



LEARN stuff IO

Mocks

- Mock objects are simulated objects that mimic the behavior of real objects in controlled ways
- A programmer typically creates a mock object to test the behavior of some other object that depends on the first object
- Mocks are injected in the system under test (the code unit we want to test)
- Mocks have their behavior defined based on specific testing scenarios or cases

```
// mock declaration
@Mock
AuthenticationService authenticationService;

// within a test
when(authenticationService .authenticate(any(AuthenticationRequest.class)))
        .thenReturn(new AuthenticationResponse());
```

Mock Tests

```
@Mock
private QuizDAO quizDAO;

@InjectMocks
private static MetaAssembler metaAssembler;

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

LEARN stuff IO

Mock Tests

```
@Test
public void shouldTestAssembleQuiz(){
    // given
    Mockito.when(quizDAO.quizFound(any(String.class), any(String.class),
    any(Boolean.class))).thenReturn(false);

    // when
    Quiz quiz = metaAssembler.assembleQuiz(ArtefactMetaBuilder.getQuiz());

    // then
    assertTrue(quiz.getQuiz_name().equals("test"));
}
```

MVC Tests

```
MockMvc mockMvc;
ObjectWriter jsonWriter;
@InjectMocks
HomeController homeController;
@Before
public void setup() throws IOException {
	jsonWriter = new ObjectMapper().writer();
	MockitoAnnotations.initMocks(this);
	InternalResourceViewResolver viewResolver = new

				InternalResourceViewResolver();
	viewResolver.setPrefix("/WEB-INF/jsp/view/");
	viewResolver.setSuffix(".jsp");
	mockMvc = MockMvcBuilders.standaloneSetup(homeController)
			.setViewResolvers(viewResolver).build();
}
```

MVC Tests

```java
@Test
public void checkHomeMappingShowGetOkStatus() throws Exception {
	// then
	mockMvc.perform(get("/home")).andExpect(
		MockMvcResultMatchers.status().isOk());
}
```

In Memory DB Tests - hs.xml

```
<jdbc:embedded-database id="dataSource" type="HSQL">
</jdbc:embedded-database>
<!-- Defines the entity manager factory -->
<bean id='entityManagerFactory'
        class='org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean'>
        <property name="persistenceUnitName" value="endava-quiz" />
        <property name='dataSource' ref='dataSource' />
        <property name="jpaVendorAdapter">
                <bean id="jpaAdapter"
                        class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                        <property name="databasePlatform" value="org.hibernate.dialect.HSQLDialect" />
                        <property name="generateDdl" value="true" />
                        <property name="showSql" value="false" />
                </bean>
        </property>
</bean>
<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<tx:annotation-driven />
```

In Memory DB Tests

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"classpath:hs.xml","classpath:root-context.xml"})
@Transactional
public class QuizDAOTest {
	@Autowired
	private QuizDAO quizDAO;
	private Quiz quiz;

	@Before
	public void setUp() {
		quiz = new Quiz();
		quiz.setQuiz_name("test");
		quizDAO.save(quiz);
	}
```

In Memory DB Tests

```
@Test
public void shouldTestSaveQuiz() {
        Quiz quiz = new Quiz();
        quiz.setQuiz_name("test");
        quizDAO.save(quiz);
        assertTrue(quiz.getId() != null);
}
```

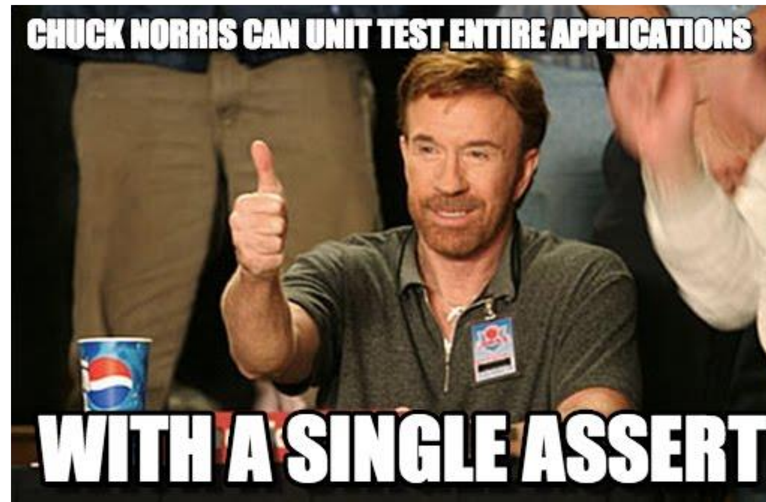
Component Tests

```
@Autowired
private WebApplicationContext wac;

@Before
public void setUp() {
    mockServerClient = new MockServerClient(MOCK_SERVER_HOST,
        MOCK_SERVER_PORT);
    mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();

    mockServerClient.reset();
  }
```

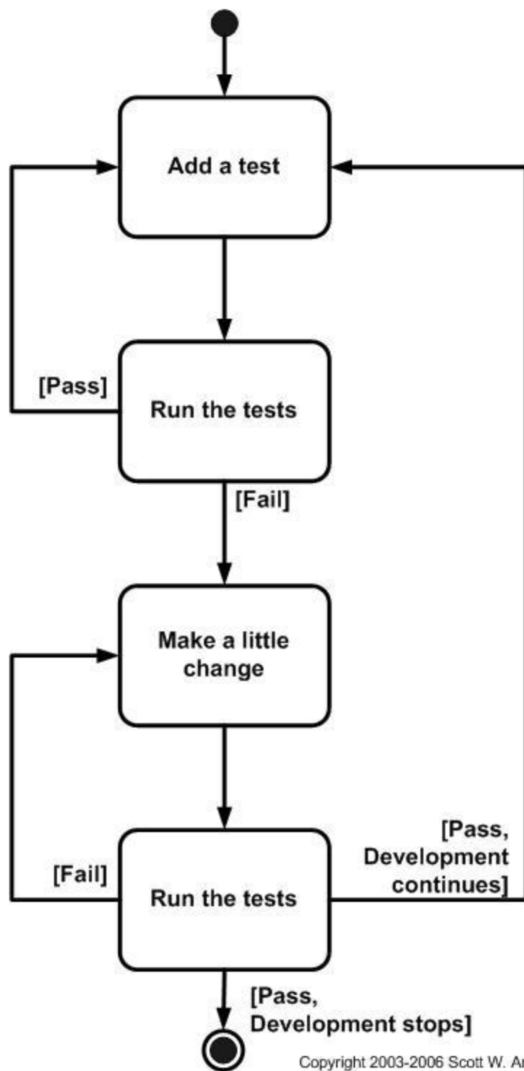
LEARN stuff IO

Component Tests

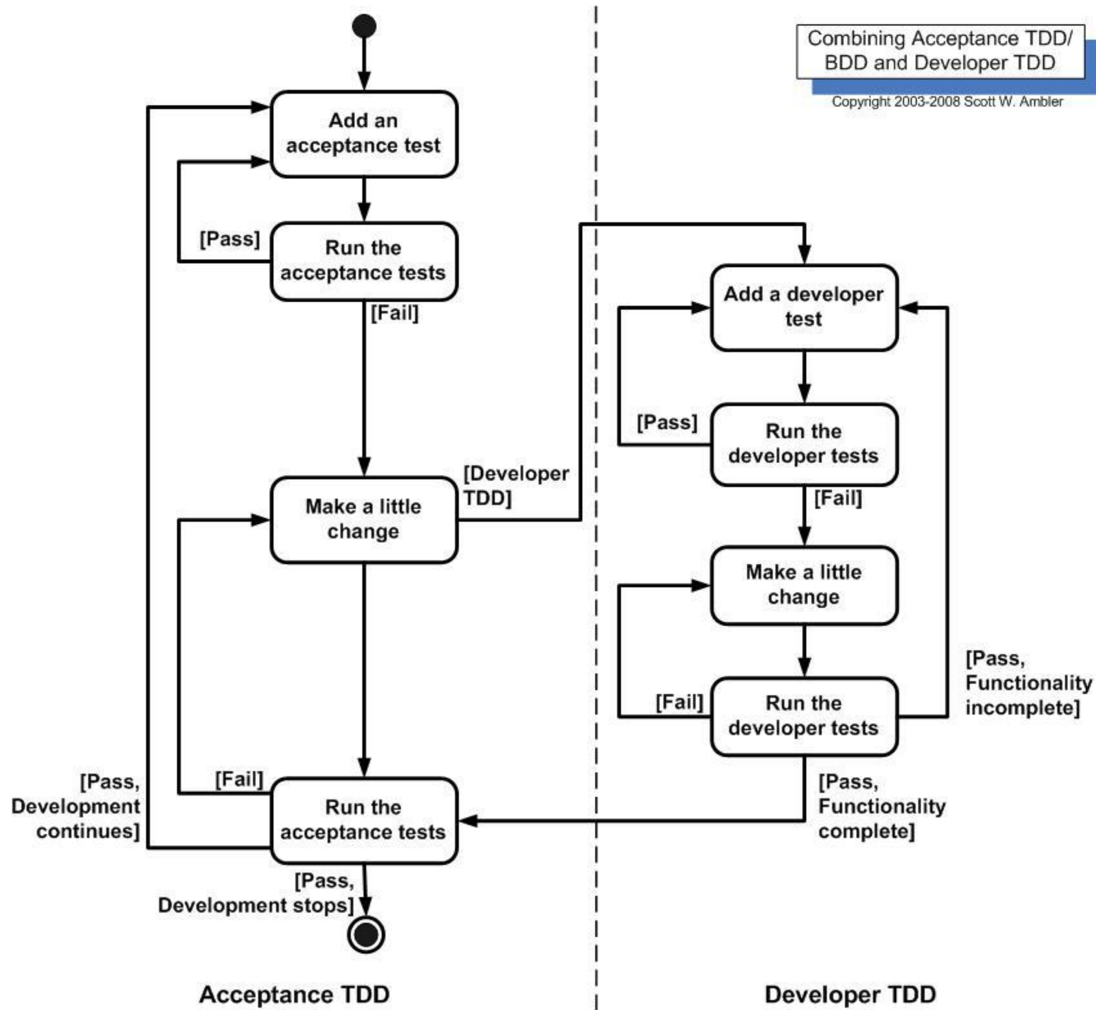
```
@Test
public void logoutValidTokenShouldReturnSuccess() throws Exception {
   final int responseStatus = 200;
   final String response = "";
   final LogoutRequest logoutRequest = aLogoutRequest();
   setupExpectationsFor(logoutRequest, responseStatus, response);

   mockMvc.perform(
         post(LOGOUT_PATH).contentType(APPLICATION_JSON_VALUE)
         .content(jsonWriter.writeValueAsString(logoutRequest)))
         .andExpect(status().is(responseStatus));
}
```


CHUCK NORRIS CAN UNIT TEST ENTIRE APPLICATIONS
WITH A SINGLE ASSERT

TDD



TDD and Acceptance TDD



THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –