Unit Testing

Unit testing is a crucial practice in software development aimed at verifying the correctness of individual units or components (such as functions, methods, or classes) of a software application. It involves writing and executing automated tests to ensure that each unit behaves as expected under various conditions. Unit testing is typically performed during the development phase and helps identify and eliminate defects early in the development process, reducing the risk of issues in later stages of integration and deployment.

Example1: an unit test implementation in Java Spring Boot application

```
@ExtendWith(MockitoExtension.class)
public class OrderServiceTest {

  @Mock
  private OrderRepository mockOrderRepository;

  @Autowired
  private OrderService orderService;

  @Test
  public void testPlaceOrder_success() {
    Order order = new Order(1L, "ITEM_123", 2);
    when(mockOrderRepository.save(order)).thenReturn(order);
    orderService.placeOrder(order);
    verify(mockOrderRepository).save(order);
  }
}
```

let's consider testing the placeOrder method in the OrderService class, which interacts with the OrderRepository to save an order.

In the OrderServiceTest class, we use Mockito for mocking dependencies. We create a mock object (mockOrderRepository) of the OrderRepository interface and inject it into the OrderService instance (orderService) using @Mock.

The testPlaceOrder method is written to test the placeOrder functionality. Here, we create a sample Order object, call the placeOrder method with this object, and then verify that the save method of the mock repository was called with the sample order using verify from Mockito.

This testing approach ensures that we isolate the placeOrder method for testing and verify its correct interaction with the OrderRepository, simulating database operations without actually hitting the database during testing. Adjust the test logic and assertions

as needed based on your specific application requirements and actual implementations.

Example2: an unit test implementation in Python

```python
import unittest

3 usages
def calculate_discount(price, discount_percent):
    discount = price * (discount_percent / 100)
    return price - discount

class TestDiscountCalculation(unittest.TestCase):

    def test_no_discount(self):
        price = 100
        discount_percent = 0
        expected_price = price

        discounted_price = calculate_discount(price, discount_percent)
        self.assertEqual(discounted_price, expected_price)

    def test_with_discount(self):
        price = 200
        discount_percent = 25
        expected_price = price * (1 - discount_percent / 100)

        discounted_price = calculate_discount(price, discount_percent)
        self.assertEqual(discounted_price, expected_price)

    def test_invalid_discount(self):
        price = 50
        discount_percent = -10

        with self.assertRaises(ValueError):
            calculate_discount(price, discount_percent)

if __name__ == '__main__':
    unittest.main()
```

Example2 is also an unit test written in Python. We need to import unittest module similar we have been doing above. In this scenerio, We define the calculate_discount function that calculates the discounted price based on the provided price and discount percentage. Then we define multiple test methods like test_with_discount, test_invalid_discount, test_no_discount to test different aspects of our method.

Integration Testing

Integration testing is a software testing method that validates the functionality of the combined units or components of an application. It specifically targets the interactions and interfaces between different units to ensure they work together as expected. Integration testing becomes crucial when multiple modules or units are integrated into a larger application, ensuring that the integrated system functions correctly as a whole.

One effective approach to integration testing is black box testing, also known as behavioral testing. In black box testing, the tester evaluates the software's functionality based solely on the provided specifications or requirements. The tester has no knowledge of the internal workings or implementation details of the software being tested.

Example3: an integration testing implementation in Java Spring Boot application

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

@SpringBootTest
@AutoConfigureMockMvc
public class CustomerControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetCustomerById() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/customers/1")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.id").value(1));
    }
}
```

In this example, Lets assume we have a simple application like in our homework which is named Online Shopping System. We have a Customer entity, a CustomerRepository interface extending JpaRepository, a CustomerService class, and a RESTful controller

(CustomerController) that uses the CustomerService and we want to test the integration of the controller layer. After including the necessary dependencies, we create an integration test class for our controller so we use the @SpringBootTest annotation to use the beans for testing. In our method testGetCustomerById() , we are simply sending a GET resquest to (/customers/{id}) endpoint which exists in our CustomerController and we want to verify that the response status is OK (200) and the response body contains a JSON field "id".

As the application evolves or new features are added, additional integration tests can be added to the CustomerControllerIntegrationTest class to cover more endpoints and scenarios, ensuring comprehensive testing of the application's integration points. Adjust the test cases according to your application's specific requirements and endpoints.

To conclude, integration testing ensures that the controller layer, along with its interactions with the service layer and data access layer, works correctly within the context of the running a Spring application. This approach allows us to validate the behavior of our RESTful endpoints and their integration with the underlying business logic.