TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

PRUEBA Y CALIDAD DE SOFTWARE

EJEMPLOS

EJEMPLOS

Ejemplo práctico 1: Método simple de suma

Se va a realizar una clase en java para la suma de dos números la cual será la clase que se debe probar.

Figura 1. Código clase CalculadoraService

```java
@Service
public class CalculadoraService {
    public int sumar(int a, int b) {
        return a + b;
    }
}
```

La imagen anterior muestra la clase suma, un código sencillo pero servirá de ejemplo para guiar sobre lo que es una prueba unitaria; se necesita probar si la suma funciona de forma correcta.

A continuación, se hará el caso de prueba para la clase con JUnit. Más adelante se dará más detalle sobre qué es JUnit.

Figura 2. Caso de prueba Clase Suma

```java
@SpringBootTest
public class CalculadoraServiceTest {

    @Autowired
    private CalculadoraService calculadoraService;

    @Test
    public void testSumar() {
        int resultado = calculadoraService.sumar(10, 5);
        assertEquals(15, resultado);
    }
}
```

En esta prueba se verifica que la suma de dos enteros (10 y 5) dé como resultado 15. La prueba se ejecuta automáticamente y validará si el valor retornado coincide con el esperado.

Ejemplo práctico 2: Prueba con lógica condicional

Ahora se hará una prueba con un grado más de complejidad, cuando el código tiene condicionales.

Figura 3. Clase validar Edad

```java
public class ValidadorEdad {
    public String clasificarEdad(int edad) {
        if (edad < 18) return "Menor de edad";
        else if (edad <= 60) return "Adulto";
        else return "Adulto mayor";
    }
}
```

Con el código que muestra la imagen, se debe hacer una prueba unitaria que permita saber si ese código está dando siempre el resultado correcto.

Figura 4. Casos de prueba para clase validar edad

```
public class ValidadorEdadTest {

    private ValidadorEdad validador;

    @BeforeEach
    public void setUp() {
        validador = new ValidadorEdad();
    }

    @Test
    public void testClasificarEdadMenor() {
        assertEquals("Menor de edad", validador.clasificarEdad(17));
    }

    @Test
    public void testClasificarEdadAdulto() {
        assertEquals("Adulto", validador.clasificarEdad(45));
    }

    @Test
    public void testClasificarEdadMayor() {
        assertEquals("Adulto mayor", validador.clasificarEdad(75));
    }
}
```

Se prueban las tres posibles rutas lógicas de la función. Esto permite asegurar la cobertura de decisiones, una métrica clave en pruebas unitarias.

Para finalizar este tema, se va a revisar cómo se realiza una prueba unitaria con una clase más avanzada, donde se utilizará Mockito para simular dependencias.

Figura 5. Clase obtener Cliente por Id

```
@Service
public class ClienteService {
    @Autowired
    private ClienteRepository clienteRepository;

    public Cliente obtenerCliente(Long id) {
        return clienteRepository.findById(id).orElse(null);
    }
}
```

La imagen anterior muestra como se tiene un código más avanzado, ya usa arquitecturas de desarrollo y se tiene un repositorio; en este caso, nos enfocamos en las pruebas que son las objeto de estudio en este curso.

Figura 6. Caso de prueba simulando el repositorio

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

    @Mock
    private ClienteRepository clienteRepository;

    @InjectMocks
    private ClienteService clienteService;

    @Test
    public void testObtenerClienteExistente() {
        Cliente cliente = new Cliente(1L, "Ana", "ana@correo.com");
        when(clienteRepository.findById(1L)).thenReturn(Optional.of(cliente));

        Cliente resultado = clienteService.obtenerCliente(1L);
        assertNotNull(resultado);
        assertEquals("Ana", resultado.getNombre());
    }

    @Test
    public void testObtenerClienteInexistente() {
        when(clienteRepository.findById(2L)).thenReturn(Optional.empty());
        Cliente resultado = clienteService.obtenerCliente(2L);
        assertNull(resultado);
    }
}
```

Con Mockito se simula el comportamiento del repositorio para que retorne un cliente o un valor vacío, según el caso. Esto evita tener que acceder a una base de datos real, manteniendo la prueba aislada.