Análisis de Caso

Mocking

Ashley Rodriguez.

Situación Inicial.

Una startup llamada FoodDeliveryX desarrolla un sistema de entregas a domicilio que se integra con múltiples proveedores externos, desde pasarelas de pago hasta servicios de geolocalización. Actualmente, cuando el equipo desea probar la funcionalidad de “solicitar repartidor” o “confirmar pago”, debe conectarse a ambientes externos o entornos de staging costosos y poco confiables. Para acelerar las pruebas y no depender de la disponibilidad de dichos servicios, se ha decidido incorporar técnicas de Mocking. El objetivo es simular el comportamiento de las dependencias (por ejemplo, un “ProveedorDeRepartidorService” o un “PagoOnlineService”) y así garantizar pruebas unitarias más rápidas, focalizadas y estables en cada funcionalidad interna. El equipo pretende usar Mockito para crear mocks y verificar interacciones, permitiendo aislar las capas internas del sistema durante el desarrollo en Visual Studio Code.

1. Revisión de Dependencias y Servicios

*Identifica y describe al menos dos servicios externos críticos en la aplicación (por ejemplo, “ServicioPago” y “ServicioRepartidor”).

Servicios externos críticos:

- ServicioPago: Encargado de procesar pagos a través de un proveedor externo. Sin mocking, cada prueba requeriría conexión a internet y acceso al entorno del proveedor, lo que introduce lentitud y posibles fallos por indisponibilidad.
- ServicioRepartidor: Gestiona la asignación de un repartidor a un pedido, Usarlo real en pruebas podría generar solicitudes reales, consumir recursos y depender de que el servicio esté disponible.

Por qué complican las pruebas unitarias:

- Dependencia de factores externos (internet, disponibilidad de APIs,etc).
- Costos si los proveedores cobran por uso.

Cómo mocking soluciona el problema:

- Permite simular la respuesta de los servicios sin hacer llamadas reales.
- Garantiza pruebas rápidas, estables y repetibles.

2. Creación de Mocks con Mockito

pom.xml | PedidoService.java | ServicioPago.java | ServicioRepartidor.java | PedidoServiceTest.java 1

src > main > java > com > ejemplo > PedidoService.java > Java > PedidoService

```java
package com.ejemplo;

public class PedidoService {
    private final ServicioPago servicioPago;
    private final ServicioRepartidor servicioRepartidor;

    public PedidoService(ServicioPago servicioPago, ServicioRepartidor servicioRepartidor) {
        this.servicioPago = servicioPago;
        this.servicioRepartidor = servicioRepartidor;
    }

    public boolean procesarPedido(String pedidoId, double monto, String direccion) {
        if (servicioPago.procesarPago(pedidoId, monto)) {
            servicioRepartidor.asignarRepartidor(pedidoId, direccion);
            return true;
        }
        return false;
    }
}
```

```java
@ExtendWith(MockitoExtension.class)
public class PedidoServiceTest {

    @Mock
    ServicioPago servicioPago;

    @Mock
    ServicioRepartidor servicioRepartidor;

    @InjectMocks
    PedidoService pedidoService;

    @Test
    void procesarPedido_exitoso() {
        when(servicioPago.procesarPago(pedidoId:"123", monto:100.0)).thenReturn(true);

        boolean resultado = pedidoService.procesarPedido(pedidoId:"123", monto:100.0, direccion:"Calle Falsa 123");

        assertTrue(resultado);
        verify(servicioPago).procesarPago(pedidoId:"123", monto:100.0);
        verify(servicioRepartidor).asignarRepartidor(pedidoId:"123", direccionEntrega:"Calle Falsa 123");
    }
```

3. Simulación de Errores y Excepciones

```java
@Test
void procesarPedido_pagoRechazado() {
    when(servicioPago.procesarPago(anyString(), anyDouble()))
        .thenThrow(new RuntimeException(message:"Pago rechazado"));

    assertThrows(expectedType:RuntimeException.class, () -> {
        pedidoService.procesarPedido(pedidoId:"456", monto:200.0, direccion:"Av. ramon 72");
    });

    verify(servicioPago).procesarPago(pedidoId:"456", monto:200.0);
    verifyNoInteractions(servicioRepartidor);
}
```

4. Captura de Argumentos y Uso de Spies

```java
@Test
void capturarArgumentosRepartidor() {
    when(servicioPago.procesarPago(anyString(), anyDouble())).thenReturn(true);

    pedidoService.procesarPedido(pedidoId:"789", monto:150.0, direccion:"Calle Luna 45");

    ArgumentCaptor<String> captorPedido = ArgumentCaptor.forClass(String.class);
    ArgumentCaptor<String> captorDireccion = ArgumentCaptor.forClass(String.class);

    verify(servicioRepartidor).asignarRepartidor(captorPedido.capture(), captorDireccion.capture());

    assertEquals(expected:"789", captorPedido.getValue());
    assertEquals(expected:"Calle Luna 45", captorDireccion.getValue());
}
}
```

5. Configuración y Ejecución en Visual Studio Code

Se utilizó Visual Studio Code como entorno de desarrollo, integrando las herramientas de Maven y JUnit para ejecutar pruebas de forma automatizada.

Para configurar el entorno:

1. Se añadió la dependencia de **JUnit 5** al archivo pom.xml, para escribir y ejecutar las pruebas unitarias.

```
>> C:\Users\ashle\OneDrive\Desktop\mocking>
java version "24.0.2" 2025-07-15
Java(TM) SE Runtime Environment (build 24.0.2+12-54)
Java HotSpot(TM) 64-Bit Server VM (build 24.0.2+12-54, mixed mode, sharing)
PS C:\Users\ashle\OneDrive\Desktop\mocking>
```

```xml
<!-- JUnit 5 -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
</dependency>
</dependencies>
```

2. En pom.xml, se agregan las dependencias de Mockito.

```xml
<dependencies>
  <!-- Mockito -->
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
  </dependency>
```

3. Ejecución en terminal:

   “mvn test”

```
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.161 s
[INFO] Finished at: 2025-08-11T21:46:54-04:00
```

4. Extensiones utilizadas en VSC:

   ***Extension Pack for Java** (Microsoft): Incluye Language Support for Java, Maven for Java, Java Test Runner, entre otros.

Reflexión Final

Ventajas de aislar dependencias en pruebas unitarias

- Velocidad: Al no depender de conexiones a internet ni de servicios externos, las pruebas se ejecutan en milisegundos, lo que permite iterar más rápido en el desarrollo.
- Fiabilidad: Los resultados de las pruebas son consistentes y no están sujetos a la disponibilidad o latencia de terceros.
- Detección temprana de errores: Los fallos en la lógica interna se identifican antes de integrar con servicios reales, evitando problemas costosos en producción.

Dificultades al introducir mocking a pruebas integradas

- Confianza inicial: Algunos desarrolladores pueden desconfiar de que un test con mocks cubra todos los escenarios que se dan en producción.
- Mantenimiento de pruebas: Si las interfaces cambian, los mocks deben actualizarse para que reflejen el contrato real.

Cómo la adopción de mocks en Visual Studio Code reduce la complejidad y promueve un desarrollo seguro

- Entornos más simples.
- Pruebas siempre disponibles.
- Iteración continua.
- Desarrollo seguro.