Unit Testing & TDD Training

Marcelo Nicolás Busico

Agenda

- Part 1: Unit Testing
- Part 2: Object Oriented Design for Testing
- Part 3: Test-driven development

Unit Testing

Basic Concepts

Definición:

- Código hecho para probar el código de una parte de nuestra aplicación de forma automatizada, verificando el funcionamiento (resultado / comportamiento) de los métodos de una clase particular.

Basic Concepts

Características principales:

- **Independientes**: Un test no debe afectar a otro. Tampoco importa el orden de ejecución de los tests.
- **Rápidos**: Un test que demora varios segundos no es un unit test.
- **Entorno controlado**: No depender de sistemas externos. Conocer la respuesta de los colaboradores.

Test Classes Structure

- **Setup de la clase:** Se ejecuta sólo una vez al inicializar la clase de tests.
- **Setup del test:** Se ejecuta cada vez antes de iniciar la ejecución de un test unitario.
- **Métodos de test**: Implementación del test.
- **TearDown del test**: Se ejecuta al finalizar cada unit test.
- **TearDown de la clase**: Se ejecuta al finalizar todos los tests de la clase.

Example

```
public class PaymentServiceTest {

    private PaymentService paymentService;

    @BeforeClass
    public static void setUpClass() {
        StorageServiceFactory.setStorageService(new InMemoryStorageService());
    }

    @AfterClass
    public static void tearDownClass() {
        StorageServiceFactory.useDefaultStorageService();
    }

    @Before
    public void setUp() {
        paymentService = new PaymentService();
    }

    @After
    public void tearDown() {
        paymentService.dispose();
    }

    @Test
    public void comingSoonTest() {
    }

}
```

Test Naming

- Utilizar **nombres descriptivos** de tests. Provee feedback rápido cuando ese test falla.

- No prefijar con el **sufijo test** a no ser que sea requerido por el framework.

- Debe poder decir en pocas palabras, nombre del **método** bajo testing, **condición** probada, y **resultado** esperado.

Test Naming

- Ejemplo nombre incorrecto:
  - testCalculateArea4
- Ejemplo de un nombre de test descriptivo:
  - calculateArea_sidesWithNegativeValues_shouldFailWithException

Test Organization

- **Arrange / Setup**
  - Se preparan los objetos necesarios para la ejecución del test.
- **Act / Test**
  - Se realiza la llamada a la clase bajo testing (SUT), a un método específico.
- **Assert / Verify**
  - Se verifica el resultado / comportamiento esperado.

Verification Types

- **State Verification**
  - Se verifica el resultado devuelto por el test, sin importar las interacciones que realice para obtener este resultado. Útil en la mayoría de los casos.

- **Behavior Verification**
  - Se verifica que las interacciones del objeto bajo testing con sus colaboradores sean las correctas (se utilizan mocks para esto). Útil para algunos casos específicos (Ej: caché).

Example - State Verification

```
@Test
public void getPayments_oneRegisteredPayment_shouldReturnListWithRegisteredPayment() {
    //Setup
    Payment payment = new Payment(50);
    StorageService storageService = StorageServiceFactory.getStorageService();
    storageService.saveObject(payment);

    //Test
    List<Payment> payments = paymentService.getPayments();

    //Verify
    assertNotNull("getPayments result should not be null.", payments);
    assertEquals("getPayments with invalid amount of payments.", 1, payments.size());
    assertEquals("getPayments did not retrieve previously stored payment.", payment, payments.get(0));
}
```

Test Doubles

- **Dummy**: Objetos que se pasan a la clase bajo test solo para satisfacer sus dependencias pero nunca se utilizan para la ejecución del test. (Ej. Dependencias en Constructor).

- **Fake**: Objetos con implementación funcional, pero con atajos útiles para testing (Ej. Base de Datos en Memoria).

Test Doubles

- **Stubs**: Objetos con respuestas pre-programadas, pero no responden a nada extra a lo que fue programado para el test. Útiles para verificaciones de estado.

- **Mocks**: Objetos pre-programados con “expectations”, que tienen especificaciones en cuanto a su uso: forma de ser llamados, cantidad de invocaciones, parámetros de invocación. Útiles para verificaciones de comportamiento.

Example - Behavior Verification (Easy Mock)

```
@Test
public void registerPayment_validPaymentData_shouldCallPaymentGatewayWithPaymentData_usingEasyMock() {
    //Setup - Data
    Payment payment = new Payment(50);
    StorageService mockStorageService = EasyMock.createMock(StorageService.class);
    PaymentService testPaymentService = new PaymentService(mockStorageService);

    //Setup - Expectations
    mockStorageService.saveObject(payment);
    EasyMock.expectLastCall();
    EasyMock.replay(mockStorageService);

    //Test
    testPaymentService.registerPayment(payment);

    //Verify
    EasyMock.verify(mockStorageService);
}
```

Example - Behavior Verification (Mockito)

```java
@Test
public void registerPayment_validPaymentData_shouldCallPaymentGatewayWithPaymentData_usingMockito() {
    //Setup - Data
    Payment payment = new Payment(50);
    StorageService mockStorageService = Mockito.mock(StorageService.class);
    PaymentService testPaymentService = new PaymentService(mockStorageService);

    //Test
    testPaymentService.registerPayment(payment);

    //Verify Behavior
    Mockito.verify(mockStorageService, Mockito.times(1)).saveObject(payment);
}
```

Object Oriented Design for Testing

Dependency Injection

- Permite que las clases de las que depende la implementación de una clase A se puedan especificar desde el exterior de la clase A, ya sea al momento de su construcción o a través de propiedades de la misma.

Dependency Injection

(Using constructor)

```java
public class MyClassWithConstructorDI {

    private final PaymentService paymentService;

    public MyClassWithConstructorDI(PaymentService paymentService) {
        if (paymentService == null) {
            throw new IllegalArgumentException("Argument paymentService null.");
        }

        this.paymentService = paymentService;
    }

    public void verifyAndRegisterPayment(Payment payment) {
        if (payment != null) {
            paymentService.registerPayment(payment);
        }
    }
}
```

Dependency Injection

(Using setter)

```
public class MyClassWithSetterDI {

    private PaymentService paymentService;

    public void setPaymentService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void verifyAndRegisterPayment(Payment payment) {
        verifyPaymentServiceIsInitialized();

        if (payment != null) {
            paymentService.registerPayment(payment);
        }
    }

    private void verifyPaymentServiceIsInitialized() {
        if (paymentService == null) {
            throw new IllegalStateException("paymentService not yet initialized.");
        }
    }
}
```

Single Responsibility

- Las clases deberían tener una única responsabilidad. Esto evita el acoplamiento de funcionalidades innecesarias y facilita la posibilidad de modificar y reutilizar las clases.
- Es opuesto a tener clases Utils/Helper, ya que tarde o temprano dejarán de tener una única responsabilidad.

Don’t talk to strangers

- Guía para el diseño de clases, en donde un objeto solo debería invocar métodos de los objetos directamente conocidos por él, y no de sus dependencias indirectas (extraños).

- Mantiene un bajo acoplamiento del código, facilita la mantenibilidad del código.

Tell don’t ask

- Principio que define que las interacciones no deberían preguntar por datos para luego decidir que acción del objeto invocar, sino que el objeto debería decidir por sí mismo y uno debería solicitarle al objeto directamente la acción sin preguntar primero.
- Mejora la cohesión y testeabilidad de las clases.

Avoid Singletons

- El Singleton es un patrón que comparte un estado global en la aplicación.
- No permite el control total de las clases bajo testing cuando dependen de singletons.
- La ejecución (y orden de ejecución) de un test puede afectar el resultado de la ejecución de otros tests.

Test-driven development

What TDD is?

- En un proceso de desarrollo de software, guiado por tests.
- Consiste en escribir primero uno o más unit tests de una clase a implementar, luego implementar el código que hace pasar ese unit test.

What TDD is?

- Cada Unit Test representa un requerimiento de uso para esa clase.
- Cada Unit Test conserva y evalúa el conocimiento del desarrollador sobre la clase a lo largo del tiempo.
- Se evita complejidad de código innecesaria que no es específica a un requerimiento.

What TDD is not?

- No es una técnica de testing.
- No es implementar una clase y luego agregar unit tests para tener buena cobertura de código.
- No son pruebas funcionales, deben ejecutar rápido y no depender de sistemas externos.

The TDD Process - Step 1

Agregar un nuevo unit test, que represente un requerimiento para un método de la clase a ser implementado.

The TDD Process - Step 2

Ejecutar el nuevo test y verificar que esté fallando. En caso de que el test no falle, significa que el requerimiento ya ha sido implementado. Continuar nuevamente con el paso 1, agregando un nuevo unit test para un nuevo requerimiento.

The TDD Process - Step 3

Implementar el método de la clase, para lograr que el nuevo unit test agregado pase exitosamente.

The TDD Process - Step 4

Ejecutar toda la suite de unit test, verificando que todos los tests de la clase pasen exitosamente. En caso de encontrar tests fallidos, corregir la implementación que hace que esos tests fallen. Siempre volver a ejecutar la suite cada vez que se cambia el código.

The TDD Process - Step 5

Mejoras y limpieza del código de la clase, esto se puede realizar sin miedo de romper la lógica de algún requerimiento, ya que al hacerlo tendríamos el feedback inmediato de los tests fallidos.

TDD Process in a nutshell

1. Agregar un nuevo test.
2. Ejecutar el test.
3. Implementar el código.
4. Ejecutar la test suite completa.
5. Refactor y limpieza de código.

Tips

- Se puede encarar el desarrollo usando TDD a partir de una interfaz que defina los métodos a ser implementados y testeados.
- Una persona podría ser la encargada de implementar los tests y otra hacer la implementación de la clase.

Tips

- Como alternativa para evitar la interfaz es definir la clase con sus métodos, simplemente lanzando una excepción indicando que el método no ha sido aún implementado.

Quick Practice

Questions?

Thank you so much!