Programa de Ingeniería de Sistemas y Computación
Universidad del Quindío

Curso: Programación Avanzada\
Guía: 05.1\
Título: Modelado del Dominio III – Pruebas Unitarias del Dominio\
Duración estimada: 120 minutos\
Docente: Christian Andrés Candela

OBJETIVO

En la guía anterior construimos el modelo de dominio implementando entidades, value objects, agregados y servicios de dominio. En esta guía daremos un paso fundamental para asegurar la calidad de dicho modelo:

Aprender a escribir pruebas unitarias efectivas para asegurar el correcto funcionamiento de los componentes centrales del dominio garantizando que las reglas de negocio e invariantes se cumplan a cabalidad sin depender de frameworks externos, infraestructura o bases de datos.

CONTEXTUALIZACIÓN TEÓRICA

En el enfoque de Diseño Guiado por el Dominio (DDD), el núcleo de nuestra aplicación (el Dominio) contiene la lógica de negocio más valiosa y crítica.

> El dominio debe ser la parte más rigurosamente probada de toda la aplicación.

¿Por qué probar el Dominio de forma aislada?

- **Rapidez:** Al no dependender de Spring Boot, bases de datos o redes, estas pruebas se ejecutan en milisegundos.
- **Confiabilidad:** Si una prueba falla, sabemos exactamente que una regla de negocio se rompió.
- **Diseño:** Escribir pruebas para el dominio nos obliga a mantenerlo puro, desacoplado y orientado al comportamiento.
- **Evolución segura:** Permite refactorizar con confianza.

Patrón AAA (Arrange, Act, Assert)

Todas nuestras pruebas seguirán este patrón:

1. **Arrange (Preparar):** Crear el contexto inicial configurando los objetos necesarios y el estado inicial.
2. **Act (Actuar):** Ejecutar el comportamiento que queremos probar (método).
3. **Assert (Comprobar):** Verificar que el resultado y el estado final son los esperados (incluyendo excepciones si se viola una regla).

PARTE 1: PRUEBAS DE VALUE OBJECTS

Los Value Objects se caracterizan por su inmutabilidad, su auto-validación, su falta de identidad y su comparación por valor. Las pruebas deben enfocarse en asegurar que no se pueden instanciar en un estado inválido y que la lógica contenida dentro de ellos se cumple estrictamente.

Ejemplo: Probando el Value Object Email

Utilizaremos `JUnit 5` para todas nuestras aserciones.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class EmailTest {

    @Test
    void dosEmailsConMismoValorDebenSerIguales() {
        Email e1 = new Email("usuario@uniquindio.edu.co");
        Email e2 = new Email("usuario@uniquindio.edu.co");

        assertEquals(e1, e2);
        assertEquals(e1.hashCode(), e2.hashCode());
    }

    @Test
    void noDebeCrearEmailInvalido() {
        // Arrange, Act & Assert
        Exception exception = assertThrows(ReglaDominioException.class, () -> {
            new Email("correo-invalido");
        });

        assertEquals("El formato del correo electrónico no es válido", exception.getMes
    }
}
```

Actividad 1:

1. Escriba pruebas unitarias para el Value Object `Email` validando diferentes formatos incorrectos.
2. Escriba pruebas unitarias para `CodigoSolicitud` validando longitud o caracteres permitidos de acuerdo a sus reglas de negocio.
3. Repita el proceso para otros Value Objects identificados.

PARTE 2: PRUEBAS DE ENTIDADES

Las Entidades tienen identidad propia, su estado puede cambiar a lo largo del tiempo mediante métodos que exponen su comportamiento y que a su vez protegen las reglas de negocio e invariantes del agregado. Las pruebas deben validar estos cambios de estado y las protecciones sobre los mismos.

Ejemplo: Probando la Entidad Usuario

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class UsuarioTest {

    @Test
    void debeAsignarNombreCorrectamente() {
        // Arrange
        Usuario usuario = new Usuario(new UsuarioId("123"), "Juan Perez");

        // Act
        usuario.actualizarNombre("Juan C. Perez");

        // Assert
        assertEquals("Juan C. Perez", usuario.getNombre());
    }

    @Test
    void noDebePermitirNombreVacio() {
        // Arrange
        Usuario usuario = new Usuario(new UsuarioId("123"), "Juan");

        // Act & Assert
        assertThrows(ReglaDominioException.class, () -> {
            usuario.actualizarNombre("");
        });
    }

    // . . .
}
```

Actividad 2:

1. Desarrolle las pruebas unitarias para las entidades que forman parte de su dominio (ej. `Usuario`).
2. Valide métodos que cambien el estado de la entidad asegurándose de que lancen excepciones si se rompen las reglas de negocio.

PARTE 3: PRUEBAS DE AGREGADOS (INVARIANTES)

El Agregado protege las **invariantes** (reglas de negocio fundamentales que siempre deben cumplirse). Probar un agregado significa intentar romper estas reglas y asegurar que el agregado se defiende lanzando excepciones.

Ejemplo: Probando el agregado Solicitud

Vamos a basarnos en la regla de negocio: "*No se puede cerrar una solicitud sin responsable y solo si está en estado ATENDIDA*".

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class SolicitudTest {

    @Test
    void debeCerrarSolicitudAtendida() {
        // Arrange
        Solicitud solicitud = new Solicitud(new SolicitudId("S-001"), TipoSolicitud.SOP
        Usuario responsable = new Usuario(new UsuarioId("U-001"), "Soporte TI");

        // Simulamos el paso a paso hasta que quede atendida
        solicitud.asignarResponsable(responsable);
        // (Supongamos que aquí se marca como atendida)
        // solicitud.setEstado(EstadoSolicitud.ATENDIDA); *Recordar hacerlo a través de

        // Act
        // solicitud.cerrar();

        // Assert
        // assertEquals(EstadoSolicitud.CERRADA, solicitud.getEstado());
    }

    @Test
    void noDebeCerrarSolicitudSinResponsable() {
        // Arrange
        Solicitud solicitud = new Solicitud(new SolicitudId("S-001"), TipoSolicitud.SOP

        EstadoSolicitud estadoInicial = solicitud.getEstado();

        // Observa que no asignamos responsable
        // Act & Assert
        ReglaDominioException excepcion = assertThrows(ReglaDominioException.class, ()
            solicitud.cerrar();
        });

        assertEquals("No se puede cerrar una solicitud sin responsable", excepcion.getM
        // El estado no debe haber cambiado
        assertEquals(estadoInicial, solicitud.getEstado());
    }
}
```

Cuando una invariante falla:

- Debe lanzarse excepción.
- El estado no debe cambiar.

Actividad 3:

1. Desarrolle las pruebas unitarias para el agregado `Solicitud`.
2. Incluya pruebas para todas las transiciones de estado permitidas y **no permitidas**.
3. Verifique que no es posible evadir las invariantes (ej. intente asignar un responsable a una solicitud que ya se encuentra cerrada y garantice que falle oportunamente).

PARTE 4: PRUEBAS DE SERVICIOS DE DOMINIO

Los servicios de dominio orquestan el comportamiento de los agregados y a menudo interactúan con componentes externos (como repositorios o servicios de terceros) a través de interfaces (Puertos), contienen la lógica de negocio que no pertenece a una única entidad o objeto de valor. Cuando un servicio de dominio requiere información o realizar validaciones que involucran componentes externos al dominio, pueden hacerlo a través de interfaces del dominio, pero no de infraestructura concreta.

Para probar la lógica de estos servicios sin la base de datos real, utilizamos **Mocks** (simulacros) usando librerías como `Mockito`.

Ejemplo: Probando NotificadorSolicitudes

Supongamos que nuestro servicio depende de una interfaz definida en el dominio para buscar usuarios a notificar.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class NotificadorSolicitudesTest {

    @Test
    void debeExtraerDestinatariosCorrectamente() {
        // Arrange
        UsuarioRepository usuarioRepoMock = mock(UsuarioRepository.class);
        NotificadorSolicitudes notificador = new NotificadorSolicitudes(usuarioRepoMock

        Solicitud solicitud = new Solicitud(new SolicitudId("S-001"), TipoSolicitud.SOP
        Usuario responsable = new Usuario(new UsuarioId("U-123"), "Ana");
        solicitud.asignarResponsable(responsable);

        // Si el notificador lo requiriese, se simula la base de datos
        // when(usuarioRepoMock.findById(cualquierId)).thenReturn(Optional.of(responsab

        // Act
        List<String> destinatarios = notificador.determinarDestinatarios(solicitud, Tip

        // Assert
        assertTrue(destinatarios.contains(responsable.getId().getValor())); // Asumiend
    }
}
```

Actividad 4:

1. Si durante el modelado diseñó **Servicios de Dominio**, implemente las correspondientes pruebas unitarias.
2. Utilice `Mockito` (`mock()`, `when()`, `verify()`) si el servicio requiere interactuar con elementos externos al dominio para simular respuestas estáticas.

EVALUACIÓN O RESULTADO

Al finalizar esta actividad usted debe tener en su proyecto:

1. Una carpeta `src/test/java` con el paquete `co.edu.uniquindio.solicitudes.domain`.

2. Clases de prueba para todas las Entidades, Value Objects, y el Agregado Raíz implementadas en la guía pasada.
3. El 100% de las reglas de negocio e invariantes descritas durante la fase de análisis deben estar validadas mediante sus correspondientes pruebas (las pruebas en verde).

PRÓXIMA ACTIVIDAD

La siguiente etapa de este proceso consiste en la **Guía 06: Casos de Uso** en la cual expondremos la capa de aplicación acoplándola con nuestra lógica de dominio ya comprobada.

REFERENCIAS BIBLIOGRÁFICAS

1. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
2. Osherove, R. (2013). *The Art of Unit Testing: with examples in C#*. Manning Publications.
3. JUnit 5 User Guide: https://junit.org/junit5/docs/current/user-guide/
4. Mockito Documentation: https://site.mockito.org/