Práctica Clean Code

(https://github.com/FranciscoHernandezPuertas/CleanCode2Ev)

Ejemplo IV: Objetos y Estructuras de Datos

Antes (DAM1HernandezPuertas_Francisco_ObjetosEstructurasPrevio.java):

```java
public class DAM1HernandezPuertas_Francisco_ObjetosEstructurasViejo {
    Run | Debug
    public static void main(String[] args) {
        // Crear un objeto de tipo Persona con datos directamente expuestos
        Persona persona = new Persona(nombre:"John", edad:30);

        // Imprimir los datos directamente desde el objeto
        System.out.println("Nombre: " + persona.nombre);
        System.out.println("Edad: " + persona.edad);

        // Operación directa en los datos expuestos
        if (persona.edad > 18) {
            System.out.println(x:"La persona es mayor de edad");
        } else {
            System.out.println(x:"La persona es menor de edad");
        }
    }
}

class Persona {
    public String nombre;
    public int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

Después (DAM1HernandezPuertas_Francisco_ObjetosEstructurasNuevo.java):

```java
public class DAM1HernandezPuertas_Francisco_ObjetosEstructurasNuevo {
    Run | Debug
    public static void main(String[] args) {
        // Crear un objeto de tipo Persona utilizando abstracciones
        Persona persona = new Persona(nombre:"John", edad:30);

        // Utilizar métodos para obtener y mostrar datos
        System.out.println("Nombre: " + persona.getNombre());
        System.out.println("Edad: " + persona.getEdad());

        // Lógica basada en comportamientos del objeto, no en sus datos internos
        if (persona.esMayorDeEdad()) {
            System.out.println(x:"La persona es mayor de edad");
        } else {
            System.out.println(x:"La persona es menor de edad");
        }
    }
}

class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Métodos para acceder a los datos de manera controlada
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    // Método para lógica relacionada con la edad
    public boolean esMayorDeEdad() {
        return edad > 18;
    }
}
```

1. **Explicación:**
   Encapsulación de Datos:
   - Se cambiaron los atributos nombre y edad en la clase Persona a privados.
   - Se crearon métodos de acceso (getNombre() y getEdad()) para obtener los datos de manera controlada.
2. **Aplicación de la Ley de Demeter:**
   - Se eliminó la manipulación directa de los datos en la clase Main (DAM1HernandezPuertas_Francisco_ObjetosEstructurasPrevio).
   - Se introdujo lógica en la clase Persona mediante el método esMayorDeEdad(), evitando la exposición de detalles internos**.**

Ejemplo V: Manejo de Errores

Antes (DAM1HernandezPuertas_Francisco_ManejoErroresPrevio.java):

```java
public class DAM1HernandezPuertas_Francisco_ManejoErroresPrevio {
    Run | Debug
    public static void main(String[] args) {
        // Llamada a una función que devuelve un código de error
        int result = divide(a:10, b:0);

        // Verificación del código de error y manejo del resultado
        if (result == -1) {
            System.out.println(x:"Error: División por cero");
        } else {
            System.out.println("Resultado: " + result);
        }
    }

    // Función que devuelve un código de error en caso de división por cero
    public static int divide(int a, int b) {
        if (b == 0) {
            return -1; // Código de error
        }
        return a / b;
    }
}
```

Después (DAM1HernandezPuertas_Francisco_ManejoErroresNuevo.java):

```java
public class DAM1HernandezPuertas_Francisco_ManejoErroresNuevo {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Llamada a una función que puede lanzar una excepción
            int result = divide(a:10, b:0);

            // Operaciones con el resultado si no hay excepción
            System.out.println("Resultado: " + result);
        } catch (ArithmeticException e) {
            // Manejo de la excepción de división por cero
            System.out.println(x:"Error: División por cero");
        } finally {
            // Bloque finally para dejar el programa en un estado consistente
        }
    }

    // Función que lanza una excepción en caso de división por cero
    public static int divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException(s:"División por cero"); // Excepción unchecked
        }
        return a / b;
    }
}
```

Explicación:

1. **Uso de Excepciones:**
    - Se reemplazó el código de retorno por el lanzamiento de una excepción (ArithmeticException) en la función divide.
2. **Manejo de Excepciones:**
    - Se implementó un bloque try-catch en el método main para manejar la excepción lanzada por la función divide.
      Se agregó un bloque finally para garantizar que el programa esté en un estado consistente.
3. **Excepciones Unchecked:**
    - Se utilizó una excepción no comprobada (ArithmeticException), ya que es preferible en este caso según la teoría proporcionada.
4. **No Devolver Null:**
    - Se evitó devolver un valor nulo en la función divide, optando por lanzar una excepción en caso de error.

Ejemplo VI: Test Unitarios

Antes (DAM1HernandezPuertas_Francisco_TestUnitariosPrevio.java):

```java
// Clase sin nombre duplicado
public class TemperaturaTestIncorrecto {

    @BeforeAll
    void setup() {
        // Configuración común para todos los tests
       }

    @Test
    void testCelsiusAFahrenheit() {
        Temperatura temperaturaObj = new Temperatura();
        double resultado = temperaturaObj.celsiusAFahrenheit(25.0);
        assertEquals(77.0, resultado, 0.01);
       }

    @Test
    void testCelsiusAReamur() {
        Temperatura temperaturaObj = new Temperatura();
        double resultado = temperaturaObj.celsiusAReamur(25.0);
        assertEquals(20.0, resultado, 0.01);
       }

    @Test
    void ejemploAsuncionCondicional() {
        double temperaturaAmbiente = obtenerTemperaturaAmbiente();
        Assumptions.assumeTrue(temperaturaAmbiente > 0, "La prueba requiere temperatura ambiente superior a cero grados Celsius");
       }

    @Test
    void asercionTimeout() {
        assertTimeout(Duration.ofSeconds(2), () -> {
            Thread.sleep(5000);
        });
       }

    @AfterEach
    void tearDown() {
        // Limpieza después de cada test
       }

    private double obtenerTemperaturaAmbiente() {
        return 10.0;
       }
}
```

Después (DAM1HernandezPuertas_Francisco_TestUnitariosNuevo.java):

```java
@DisplayName("Temperatura Test")
public class TemperaturaTestCorrecto {

    private Temperatura temperaturaObj;

    @BeforeAll
    static void setup() {
        // Configuración común para todos los tests
       }

    @BeforeEach
    void beforeEach() {
        temperaturaObj = new Temperatura();
       }

    @Nested
    @DisplayName("Celsius a Fahrenheit")
    class CelsiusAFahrenheit {

        @Test
        @DisplayName("Convierte correctamente a Fahrenheit")
        void testCelsiusAFahrenheit() {
            double resultado = temperaturaObj.celsiusAFahrenheit(25.0);
            Assertions.assertEquals(77.0, resultado, 0.01);
             }

        @Test
        @DisplayName("Maneja correctamente temperaturas por debajo del cero absoluto")
        void testCelsiusAFahrenheitConError() {
            double resultado = temperaturaObj.celsiusAFahrenheit(-300.0);
            Assertions.assertEquals(999999, resultado, 0.01);
             }
       }

    @Nested
    @DisplayName("Celsius a Reamur")
    class CelsiusAReamur {

        @Test
        @DisplayName("Convierte correctamente a Reamur")
        void testCelsiusAReamur() {
            double resultado = temperaturaObj.celsiusAReamur(25.0);
            Assertions.assertEquals(20.0, resultado, 0.01);
             }

        @Test
        @DisplayName("Maneja correctamente temperaturas por debajo del cero absoluto")
        void testCelsiusAReamurConError() {
            double resultado = temperaturaObj.celsiusAReamur(-300.0);
            Assertions.assertEquals(999999, resultado, 0.01);
             }
       }

    @Test
    @DisplayName("Ejemplo de asunción condicional basada en temperatura ambiente")
    @Tag("condicional")
    void ejemploAsuncionCondicional() {
        double temperaturaAmbiente = obtenerTemperaturaAmbiente();
        Assertions.assertTrue(temperaturaAmbiente > 0, "La prueba requiere temperatura ambiente superior a cero grados Celsius");
       }

    @Test
    @DisplayName("AssertTimeout")
    void asercionTimeout() {
        assertTimeout(Duration.ofSeconds(2), () -> {
            Thread.sleep(1000);
        });
       }

    @AfterEach
    void afterEach() {
        // Limpieza después de cada test
       }

    private double obtenerTemperaturaAmbiente() {
        // Método ficticio para obtener la temperatura ambiente
        return 10.0; // Ejemplo de temperatura ambiente
       }
}
```

Explicación:

1. **Organización de tests (Regla 29):**
    - Se sigue la estructura comúnmente recomendada para organizar tests en JUnit con anotaciones como @BeforeAll, @BeforeEach, @Nested, y @AfterEach, garantizando una estructura clara y ordenada.
2. **Cada test tiene un solo Assert (Regla 26):**
    - Cada método de prueba contiene un solo Assert, evitando la sobrecarga de verificación en un solo test.
3. **Principio de Responsabilidad Única (Regla 31):**
    - Cada método de prueba se centra en una funcionalidad específica de la clase Temperatura, como la conversión entre unidades o la gestión de condiciones ambientales.
4. **Asserts con mensajes claros (Reglas 24, 25):**
    - Los mensajes de los Asserts se han mejorado para proporcionar información más útil en caso de fallo, facilitando la identificación del problema.
5. **Cada test prueba un único concepto (Regla 27):**
    - Cada método de prueba aborda una única funcionalidad, asegurando que se pueda entender fácilmente lo que se está probando.
6. **Clases pequeñas (Regla 30):**
    - La clase de prueba (TemperaturaTestCorrecto.java) está enfocada en la funcionalidad de la clase Temperatura y evita la adición de funcionalidades no relacionadas.
7. **Uso de BeforeAll y AfterEach (Reglas 29, 33):**
    - Se utiliza @BeforeAll para la configuración común antes de ejecutar los tests y @AfterEach para la limpieza después de cada test, garantizando un entorno consistente y preparado para las pruebas.
8. **Uso de JUnit (Requisito implícito):**
    - Se siguió utilizando JUnit para la realización de las pruebas, cumpliendo con la práctica común en el desarrollo de pruebas unitarias en Java.
9. **Regla FIRST:**
    - Fast (Rápido): Los tests se diseñaron para ser rápidos, sin realizar operaciones innecesarias que podrían ralentizar la ejecución de las pruebas.
    - Independent (Independiente): Cada método de prueba es independiente entre sí, sin depender del resultado de otros tests, y pueden ejecutarse en cualquier orden.

- Repeatable (Repetible): Los tests son repetibles en cualquier entorno, ya que no dependen de estados externos y deberían dar los mismos resultados independientemente del entorno de ejecución.
- Self-Validating (Auto-validable): Los tests son auto-validables; si un test falla, la salida del framework de pruebas proporciona información suficiente para determinar el motivo del fallo.
- Timely (Oportuno): Los tests se escribieron justo antes del código de producción correspondiente, siguiendo la práctica del desarrollo basado en pruebas (TDD) para asegurar la alineación con la implementación.

Ejemplo VII: Clases

Antes (DAM1HernandezPuertas_Francisco_ClasesPrevio.java):

```java
public class DAM1HernandezPuertas_Francisco_ClasesPrevio {
}
class Empleado {
    // Variables públicas (violando la buena práctica)
    public String nombre;
    public int edad;
    public double salarioActual;

    // Constantes privadas
    private static final int SALARIO_BASE = 50000;
    private static final double BONO = 0.1;

    // Funciones públicas
    public Empleado(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        this.salarioActual = SALARIO_BASE;
    }

    public void aumentarSalario() {
        // Algoritmo para aumentar el salario
        this.salarioActual *= (1 + BONO);
        notificarAumento();
    }

    // Funciones privadas
    private void notificarAumento() {
        // Lógica de notificación
        System.out.println("El salario ha sido aumentado para " + this.nombre);
    }

    // Violando el principio de responsabilidad única
    public void cambiarDatos(String nuevoNombre, int nuevaEdad) {
        this.nombre = nuevoNombre;
        this.edad = nuevaEdad;
    }
}
```

Después (DAM1HernandezPuertas_Francisco_ClasesNuevo.java):

```java
public class DAM1HernandezPuertas_Francisco_ClasesNuevo {
}
class Empleado {
    // Constantes privadas
    private static final int SALARIO_BASE = 50000;
    private static final double BONO = 0.1;

    // Variables privadas
    private String nombre;
    private int edad;
    private double salarioActual;

    // Constructor
    public Empleado(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        this.salarioActual = SALARIO_BASE;
    }

    // Funciones públicas
    public void aumentarSalario() {
        // Algoritmo para aumentar el salario
        this.salarioActual *= (1 + BONO);
        notificarAumento();
    }

    // Funciones privadas
    private void notificarAumento() {
        // Lógica de notificación
        System.out.println("El salario ha sido aumentado para " + this.nombre);
    }

    // Constructor de copia
    public Empleado(Empleado original) {
        this.nombre = original.nombre;
        this.edad = original.edad;
        this.salarioActual = original.salarioActual;
    }
}
```

Explicación:

1. **Variables públicas y Principio de Responsabilidad Única:**
    - *Problema:* Se han declarado variables como públicas, violando la recomendación de evitar variables públicas. Además, se ha violado el principio de responsabilidad única al agregar una función (**cambiarDatos**) que no tiene relación directa con la responsabilidad principal de la clase.
    - *Solución:* Se han cambiado las variables y constantes a privadas para mejorar el encapsulamiento. Se eliminó la función **cambiarDatos** para cumplir con el principio de responsabilidad única.
2. **Cohesión y Dependencia de detalles de implementación:**
    - *Problema:* La clase tiene variables y métodos que no están fuertemente relacionados, disminuyendo la cohesión. Además, la clase depende de detalles de implementación al exponer directamente sus variables internas.
    - *Solución:* La clase ahora tiene variables y métodos más relacionados, mejorando la cohesión. Se han encapsulado las variables y métodos internos para reducir la dependencia de detalles de implementación.

3. **Constructor de copia y Separación de construcción de un sistema de su uso:**

    - *Añadido:* Se ha agregado un constructor de copia para permitir la creación de nuevos objetos duplicando un objeto existente, siguiendo el principio de separar la construcción de un sistema de su uso.