Clean Code II Manuel Sánchez Romero DAM1

Índice:

Introducción:

En este Trabajo seguiremos limpiando los más posible los archivos .java del proyecto anterior para que queden con el código más limpio y claro posible ya una vez aplicados los primeros terceros Bloques de Clean Code aquí proseguiremos con los cuatro Bloques Siguientes: Objetos y estructuras de datos, Control de Errores, Test Unitarios y Clases.

Bloque IV Objetos y Estructuras de Datos:

La primera cosa que debemos saber al inspeccionar el código en este bloque es ajustarnos a lo que nos dice la ley de Deméter así también como algunas reglas Básicas de la POO(Programación Orientada a Objetos) por lo cual debemos tener en cuenta que un objeto solo **debería** poder acceder a sus más cercanos en otras palabras a otros objetos de su clase y no a los de otra perteneciente esto lo hacemos para que sea más simple la Estructura de los Datos que tengamos además de asegurar la seguridad a nuestro Código cerrando el acceso ajeno lo más posible para eso se usa la Encapsulación un método proveniente de la POO para realizar lo comentado realmente ahora echando vista al código del Proyecto:

```
public class Estadistica {
    //Atributos de los Personajes
    int puntosDeVida;
    int puntosDeMagia;
    int medidorHabilidadDefinitiva;
    //Constantes de Daño General
    final static int DANOFISICO = 100;
    final static int DANOMAGICO = 400;
    //Constante de Desgaste
    final static int DESGASTEMAGICO = 40;
    //Constantes de la Habilidad Definitiva
    final static int INCREMENTOMEDIDORHABILIDAD_DEFINITIVA = 10;
    final static int DANOHABILIDAD_DEFINITIVA = 1000;
    final static int CAPACIDADSUFICIENTE_MEDIDORHABILIDAD_DEFINITIVA = 100;
    //Constante para estadisticas Vacias
    final static int VACIO = 0;
```

Por lo que vemos no cumplimos con lo que hemos explicado por lo cual lo que haremos es lo siguiente para encapsular atributos de una clase es asignarles el modificador prívate para que los atributos de los objetos de Estadística son ahora privados de la Clase Estadística y ninguna otra clase puede acceder a ellos de forma “abierta” por decirlo de algún modo:

```
public class Estadistica {
    //Atributos de los Personajes
    private int puntosDeVida;
    private int puntosDeMagia;
    private int medidorHabilidadDefinitiva;
    //Constantes de Daño General
    private final static int DANOFISICO = 100;
    private final static int DANOMAGICO = 400;
    //Constante de Desgaste
    private final static int DESGASTEMAGICO = 40;
    //Constantes de la Habilidad Definitiva
    private final static int INCREMENTOMEDIDORHABILIDAD_DEFINITIVA = 10;
    private final static int DANOHABILIDAD_DEFINITIVA = 1000;
    private final static int CAPACIDADSUFICIENTE_MEDIDORHABILIDAD_DEFINITIVA = 100;
    //Constante para estadisticas Vacias
    private final static int VACIO = 0;
```

Por lo cual para que siga el Programa Funcionando(ya que como ahora los atributos son privados algunos atributos no están siendo ni leídos o el programa main nos los alcanza) lo que se usan son getters y setters funciones las cuales nos permiten poder recoger estos datos pero no de la misma forma que antes ya que apuntaremos al getter o al setter en vez de directamente al objeto por lo cual así hacemos un buen uso de nuestros atributos para que nuestros Objetos estén mejor hechos y así también limpios por lo cual para eso haremos todas las modificaciones necesarias que hemos mencionado:

```java
public int getPuntosDeVida() {
    return puntosDeVida;
}

public void setPuntosDeVida(int puntosDeVida) {
    this.puntosDeVida = puntosDeVida;
}

public int getPuntosDeMagia() {
    return puntosDeMagia;
}

public void setPuntosDeMagia(int puntosDeMagia) {
    this.puntosDeMagia = puntosDeMagia;
}

public int getMedidorHabilidadDefinitiva() {
    return medidorHabilidadDefinitiva;
}

public void setMedidorHabilidadDefinitiva(int medidorHabilidadDefinitiva) {
    this.medidorHabilidadDefinitiva = medidorHabilidadDefinitiva;
}

public static int getCapacidad() {
    return CAPACIDADSUFICIENTE_MEDIDORHABILIDAD_DEFINITIVA;
}

public static int getVacio(){
    return VACIO;
}

public static int getDesgaste(){
    return DESGASTEMAGICO;
}
```

Ahora ya hecho esto solo nos faltaría por ultimo cambiar el acceso en el Main para poder terminar toda esta tarea tanto de Encapsulación como de limpieza:

```java
if( ataque.equals("Golpes Normales") ){
    int ataqueJugador = Jugador.usaGolpesNormales(Enemigo);
    incrementoDefinitiva = Enemigo.aumentaMedidorHabilidadDefinitiva(Enemigo);
    Enemigo.setPuntosDeVida(ataqueJugador);
    Enemigo.setMedidorHabilidadDefinitiva(incrementoDefinitiva);
    System.out.println("Estadisticas Actuales del Enemigo: "+Enemigo);
    System.out.println("El enemigo Contrataca");
    int ataqueEnemigo = Enemigo.usaGolpesNormales(Jugador);
    incrementoDefinitiva = Jugador.aumentaMedidorHabilidadDefinitiva(Jugador);
    Jugador.setPuntosDeVida(ataqueEnemigo);
    Jugador.setMedidorHabilidadDefinitiva(incrementoDefinitiva);
    System.out.println("Tus estadisticas actuales:" + Jugador);
```

Y así con todo el Main restante al ser bastante extenso no mostrare Todo el Main modificado sin embargo puedes mirarlo entero si te apetece en el repositorio que he creado en GitHub además con esto aprovechamos enseñanzas de los bloques Anteriores y los setters de las Constantes CAPACIDADSUFICIENTE_MEDIDORHABILIDAD_DEFINITIVA y VACIO por unos

nombres más cortos en sus getters:

```
            if(Enemigo.getPuntosDeVida() == Estadistica.getVacio()){
                System.out.println("Has ganado la batalla");
                break;
            }else{
                System.out.println("El enemigo usa su Habilidad Definitiva");
                int ataqueEnemigoDefinitivo = Enemigo.usaHabilidadDefinitiva(Jugador);
                Jugador.setPuntosDeVida(ataqueEnemigoDefinitivo);
                System.out.println("Tus estadisticas actuales:" + Jugador);
                reinicioHabilidadDefinitiva = Enemigo.desgasteUlti(Enemigo);
                Enemigo.setMedidorHabilidadDefinitiva(reinicioHabilidadDefinitiva);
            }
        }
    //Comprobacion de Contorno
    }else{
        System.out.println("Incorrecto introduce uno de los 3 ataques mencionados");
    }
    if(Enemigo.getPuntosDeVida() == Estadistica.getVacio()){
        System.out.println("Has ganado la batalla");
        break;
    }
```

Ya habiendo hecho esto habríamos arreglado el Problema de Suciedad en nuestro Código respecto a Objetos y Estructuras de Datos.

Bloque V Manejo de Errores:

Ahora entramos al Bloque de Manejo de Errores aquí nos basaremos en cómo se manejan los errores en nuestro código el fin de esto es hacerlo de la forma más limpia posible que sepamos para así también dejar a nuestro programa en un estado consistente lo cual de hecho es uno de los principios de este bloque para esto en vez de usar código de retorno en constructores o en el propio main lo que haremos será utilizar Excepciones que son lo que mayoritariamente utilizaremos aquí aunque los ejemplos serán breves debido a que nuestro proyecto en el Main ya tiene la mayoría de sus problemas resueltos por lo cual lo que haremos será añadir dos formas de Manejo de errores (O sea dos Excepciones) para poder tratar esta tema de fondo y también ampliar un poco más la limpieza de este mismo lo primero sería cubrir un caso en el que al hacerse el constructor de los objetos en el Main se dé por poner datos negativos ahí en vez de utilizar una comprobación de contorno con return la cual muestre un mensaje de error utilizaremos una excepción no verificada(unchecked Exception en Ingles) ya que en Java se nos permite hacerlo para no tener que obligatoriamente atrapar la excepción(Ya que si esta se propaga más de lo debido puede cambiar bastantes cosas en la Capa del Programa y eso no sería limpio ni ventajoso al menos en este caso):

```
//Constructor Completo
Estadistica(int Vida,int Magia,int Definitiva){
    if (Vida < 0 || Magia < 0 || Definitiva < 0){
        throw new IllegalArgumentException(s: "Los valores de vida, magia y definitiva no pueden ser negativos");
    }
    puntosDeVida = Vida;
    puntosDeMagia = Magia;
    medidorHabilidadDefinitiva = Definitiva;
}
```

Ahora para el segundo caso lo que haremos será escribir un bloque try-catch-finally para manejar los errores que pueda haber al crear objetos en un Main obviamente lo pondremos al principio ya que como se dice en este bloque se debe escribir esto lo primero para así asegurarnos de dejar el programa en un estado consistente como justo hemos mencionado anteriormente simplemente lo que haremos es crear uno o dos objetos con datos erróneos que normalmente darían error aquí lo que haremos es darles sus debidos mensajes de Error para que sean más Comprendidos fácilmente en caso de que ocurran:

```
//Excepcion try-catch-finally para Pruebas
try {
    // Intenta realizar algunas operaciones con Estadistica
    Estadistica jugador = new Estadistica(Vida: -50, Magia: 30, Definitiva: 10);
    System.out.println("Estadísticas del jugador: " + jugador);
    // Intenta usar Vacío Purpura
    Estadistica enemigo = new Estadistica(Vida: 200, Magia: 0, Definitiva: 10);
    jugador.usaVacioPurpura(HeridaM:enemigo);
} catch (IllegalArgumentException e) {
    //Captura la excepción del constructor
    System.err.println("Error en la creación de estadísticas: " + e.getMessage());
} catch (IllegalStateException e) {
    //Captura la excepción del ataque Vacío Purpura
    System.err.println("Error al usar Vacío Purpura: " + e.getMessage());
} finally {
    // Bloque finally
    System.out.println(x: "Fin del programa");
}
```

Ya con estos dos excepciones es suficiente para completar este bloque aunque un consejo que final que se pueda dar es el de nunca devolver Null ya que puede ser peligroso al nivel de que puede cargarse todo el programa a base de Chequeos nulos además quien llame a la función no **NECESITA** saber que lo que llega es Nulo o no aunque para este caso lo aplica ya que estamos hablando de limpiar lo dejamos aquí como Consejo por lo cual con todo esto explicado y dos ejemplos puestos para mejorar tanto la funcionalidad del código como su limpieza damos por terminado este Bloque.

Bloque VI Pruebas Unitarias:

Ahora Empezamos con el Bloque VI de Clean Code en este mismo nos vamos a tratar de usar test pero claro tendremos que usarlos de una manera limpia y concreta ya que estos también tienen sus propias reglas que cumplir a la hora de utilizarlos una de las razones por las que son parte de Clean Code es porque las pruebas Unitarias son **VITALES** para cualquier programa que se cree ya que facilitan la vida del Desarrollador y de la siguiente persona que tenga que tocar el programa debido a que gracias a estas pruebas se pueden detectar de manera temprana los errores porque si una de sus reglas es que ya teniendo el programa Planteado creemos sus pruebas correspondientes por todas las ventajas mencionadas anteriormente por si acaso en el desarrollo del propio Programa se nos dificulta por algún error o imprevisto en algún momento de este para eso primero deberemos poner las dependencias y plugins necesarios en nuestro pom.xml el cual también puedes revisar en el repositorio:

```xml
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.10.1</version>
        <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.10.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
 <build>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.12.1</version>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
             <artifactId>maven-surefire-plugin</artifactId>
             <version>3.2.5</version>
        </plugin>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-failsafe-plugin</artifactId>
           <version>3.2.5</version>
        </plugin>
    </plugins>
</build>
```

Ya puestos los plugins como las dependencias ahora podremos crear los test en su archivo java correspondiente el cual puedes revisar en el repositorio como todos los ejemplos que se exponen aquí:

```java
@Test
public void testConstructorDefault() {
    Estadistica estadistica = new Estadistica();
    assertEquals(expected: 0, actual: estadistica.getPuntosDeVida());
    assertEquals(expected: 0, actual: estadistica.getPuntosDeMagia());
    assertEquals(expected: 0, actual: estadistica.getMedidorHabilidadDefinitiva());
}

@Test
public void testConstructorConValores() {
    Estadistica estadistica = new Estadistica(Vida: 100, Magia: 50, Definitiva: 10);
    assertEquals(expected: 100, actual: estadistica.getPuntosDeVida());
    assertEquals(expected: 50, actual: estadistica.getPuntosDeMagia());
    assertEquals(expected: 10, actual: estadistica.getMedidorHabilidadDefinitiva());
}

@Test
public void testUsaGolpesNormales() {
    Estadistica jugador = new Estadistica(Vida: 200, Magia: 0, Definitiva: 50);
    Estadistica enemigo = new Estadistica(Vida: 500, Magia: 0, Definitiva: 20);
    int resultadoAtaque = jugador.usaGolpesNormales(Herida: enemigo);
    assertEquals(expected: 400, actual: resultadoAtaque); //Vida Esperada del Enemigo despues del Ataque
}

@Test
public void testUsaVacioPurpura() {
    Estadistica jugador = new Estadistica(Vida: 300, Magia: 50, Definitiva: 30);
    Estadistica enemigo = new Estadistica(Vida: 800, Magia: 200, Definitiva: 10);
    int resultadoAtaque = jugador.usaVacioPurpura(HeridaM:enemigo);
    assertEquals(expected: 400, actual: resultadoAtaque); // Vida Esperada del Enemigo despues del Ataque
}

@BeforeAll
public static void setUpClass() {
    Jugador = new Estadistica(Vida: 100, Magia: 50, Definitiva: 10);
    Enemigo = new Estadistica(Vida: 200, Magia: 100, Definitiva: 20);
    System.out.println(x: "Inicializacion antes de todas las pruebas");
}

@AfterAll
public static void tearDownClass() {
    System.out.println(x: "Inicializacion despues de todas las pruebas");
}

@BeforeEach
public void setUp() {
    System.out.println(x: "Tareas de Limpieza antes de cada Prueba");
}
```

Ahora ya hechos los test y que funcionan perfectamente tenemos algunas cosas más que inspeccionar lo que viene siendo los test son buenos ya que cumplen la Regla F.I.R.S.T la cual básicamente consiste en que un test sea Rapido,Independiente para que de igual el momento de la ejecución el Test se pueda usar sin problema alguno, Repetible que un test sea capaz de poder repetirse todas las veces que se pueda en cualquier tipo de Entorno en el que se vaya a usar,Autovalidable(que el test ya de por si se indique que falla o se pasa sin que tenga comprobar esto mismo dentro del propio Test) y por último que sea Oportuno como hemos dicho antes de escribir estos test antes de empezar con el programa de verdad porque si no los test serán más difíciles de hacer y testear además debemos verificar más cosas ya que los test también necesitan la misma limpieza que el Código Normal por lo cual necesitamos también los principios de los Tres primeros Bloques en los Primeros test lo mínimo que podemos hacer será aportar más comentarios ya que como vemos los dos primeros test carecen de Comentarios además de que no estaría mal agrupar el contenido por Comentarios como

Hacíamos anteriormente con la Clase Estadística y el Main:

```
    //Test Normales
    @Test
    public void testConstructorDefault() {
        Estadistica estadistica = new Estadistica();
        //Aseguramos que los Valores esten vacios como por Defecto
        assertEquals(expected: 0, actual: estadistica.getPuntosDeVida());
        assertEquals(expected: 0, actual: estadistica.getPuntosDeMagia());
        assertEquals(expected: 0, actual: estadistica.getMedidorHabilidadDefinitiva());
    }

    @Test
    public void testConstructorConValores() {
        Estadistica estadistica = new Estadistica(Vida: 100, Magia: 50, Definitiva: 10);
        //Lo mismo que en el anterior Test pero con Valores Determinados
        assertEquals(expected: 100, actual: estadistica.getPuntosDeVida());
        assertEquals(expected: 50, actual: estadistica.getPuntosDeMagia());
        assertEquals(expected: 10, actual: estadistica.getMedidorHabilidadDefinitiva());
    }

    @Test
    public void testUsaGolpesNormales() {
        Estadistica jugador = new Estadistica(Vida: 200, Magia: 0, Definitiva: 50);
        Estadistica enemigo = new Estadistica(Vida: 500, Magia: 0, Definitiva: 20);
        int resultadoAtaque = jugador.usaGolpesNormales(Herida: enemigo);
        assertEquals(expected: 400, actual: resultadoAtaque); //Vida Esperada del Enemigo despues del Ataque
    }

    @Test
    public void testUsaVacioPurpura() {
        Estadistica jugador = new Estadistica(Vida: 300, Magia: 50, Definitiva: 30);
        Estadistica enemigo = new Estadistica(Vida: 800, Magia: 200, Definitiva: 10);
        int resultadoAtaque = jugador.usaVacioPurpura(HeridaM:enemigo);
        assertEquals(expected: 400, actual: resultadoAtaque); // Vida Esperada del Enemigo despues del Ataque
    }
//Tareas Antes,Despues,etc...
@BeforeAll
public static void setUpClass() {
    Estadistica Jugador = new Estadistica(Vida: 100, Magia: 50, Definitiva: 10);
    Estadistica Enemigo = new Estadistica(Vida: 200, Magia: 100, Definitiva: 20);
    System.out.println(x: "Inicializacion antes de todas las pruebas");
}

@AfterAll
public static void tearDownClass() {
    System.out.println(x: "Inicializacion despues de todas las pruebas");
}

@BeforeEach
public void setUp() {
    System.out.println(x: "Tareas de Limpieza antes de cada Prueba");
}

@AfterEach
public void tearDown() {
    System.out.println(x: "Tareas de Limpieza despues de cada Prueba");
}
```

Ahora esto ya solucionado si inspeccionamos un poquito vemos que en los test relacionados con los Construcciones estamos utilizando varias aserciones lo cual rompe una de las normas

para que un test sea limpio la cual es que por cada test deberíamos usar solo una aserción pero en caso de esto ser imposible la solución más limpia que tenemos a este dilema es agrupar todas las aserciones para que sean una sola para esto usaremos asertAll:

```
@Test
public void testConstructorDefault() {
    Estadistica estadistica = new Estadistica();
    //Aseguramos que los Valores esten vacios como por Defecto
    assertAll(
    () -> assertEquals(expected: 0, actual: estadistica.getPuntosDeVida()),
    () -> assertEquals(expected: 0, actual: estadistica.getPuntosDeMagia()),
    () -> assertEquals(expected: 0, actual: estadistica.getMedidorHabilidadDefinitiva())
    );
}

@Test
public void testConstructorConValores() {
    Estadistica estadistica = new Estadistica(Vida: 100, Magia: 50, Definitiva: 10);
    //Lo mismo que en el anterior Test pero con Valores Determinados
    assertAll(
    () -> assertEquals(expected: 100, actual: estadistica.getPuntosDeVida()),
    () -> assertEquals(expected: 50, actual: estadistica.getPuntosDeMagia()),
    () -> assertEquals(expected: 10, actual: estadistica.getMedidorHabilidadDefinitiva())
    );
}
```

Ahora ya hecho esto hemos todos los problemas de suciedad que tenían nuestros test por lo cual con todo esto terminado podemos dar fin a este bloque.

Bloque VII Clases:

Por ultimo tenemos el Bloque más Corto al menos para este Caso el de Clases ya de que por si gracias a los Tres primeros Bloques de Clean Code este Proyecto ya de por si cumple el Principio de Responsabilidad Única, otras cosas también es que el Encapsulamiento o forma de usar los atributos de la Clase si hacerlas privados, públicos o constantes en vez de variables ya lo hemos tratado en el Bloque IV por lo cual lo damos por hecho ahora siguiendo revisando los principios de este bloque también hemos revisado que cumple al igual también el Principio Open/Closed además de que si revisamos bien nuestro Código está bien Cohesionado ya que los atributos principales que representan la salud,los puntos de Magia y un medidor para la habilidad definitiva están cohesionados para mostrar las Estadísticas de un Personaje en Juego como lo es este Programa además los métodos están relacionados con la forma de utilizar las estadísticas del personaje, como usar ataques normales, lanzar habilidades especiales, y gestionar el medidor de habilidad definitiva para Derrotar al Enemigo de Forma Aplastante por último punto a defender tenemos los Constructores ya que tanto el Vacío como el Completo hacen su función en cohesión a la inicialización de las Estadísticas de un Personaje ya sea el de Un Jugador o el de un Enemigo por lo cual como mucho El Mejor Ejemplo que podríamos dar a este Bloque seria el hacer un Constructor Copia ya que una de las Reglas para que una Clase se considere Limpia es utilizar Copias de los Objetos de esta misma para que se pueda trabajar con esta misma de forma Concurrente por lo cual procederemos hacer lo mencionado para así al menos poder proporcionar un ejemplo lo suficientemente bueno para no dejar a este bloque de lado:

```
Estadistica CopiarObjeto(){
    Estadistica copia = new Estadistica();
    copia.puntosDeVida = this.puntosDeVida;
    copia.puntosDeMagia = this.puntosDeMagia;
    copia.medidorHabilidadDefinitiva = this.medidorHabilidadDefinitiva;
    return copia;
}
```

Ya con esto nos Aseguramos de tener una Copia de todos los objetos de la Clase en caso de necesitarla para poder Trabajar con el Programa de forma Concurrente como hemos explicado anteriormente siendo así que no tengamos más ejemplos de los que rascar en este Bloque ya que es bastante limitado al usar conceptos de Bloques Anteriores hace que este tenga menos Contenido pero al menos tenemos esto para poder profundizar más sobre esto.

Conclusión:

En Fin con otros Tres Bloques analizados y corregidos cuando ha sido necesario hemos conseguido que los códigos de este Proyecto sean aún más limpios que la vez anterior sin hacer falta de buscar la Perfección ya que de eso se trata el Clean Code como explique anteriormente en el Trabajo anterior esto es simplemente escribir el código de una Manera Inteligente para que cualquiera que lo lea pueda entenderlo a simple vista ya que seguramente el Código que hagamos lo puedan tocar otras personas en algún futuro y así les ahorramos tanto tiempo como problemas. Ahora si te apetece ver todos los Ejemplos más en detalle he hecho lo mismo que en el anterior Trabajo por lo cual puedes revisar los ejemplos Java antes y después de su renovada limpieza (La Clase Estadística y El Main explícitamente) además de que ahora todo el Proyecto estará en un ZIP el cual puedes exportar a tu IDE y probarlo tú de primera mano todo esto de lo que te hablo está en el repositorio de siempre.