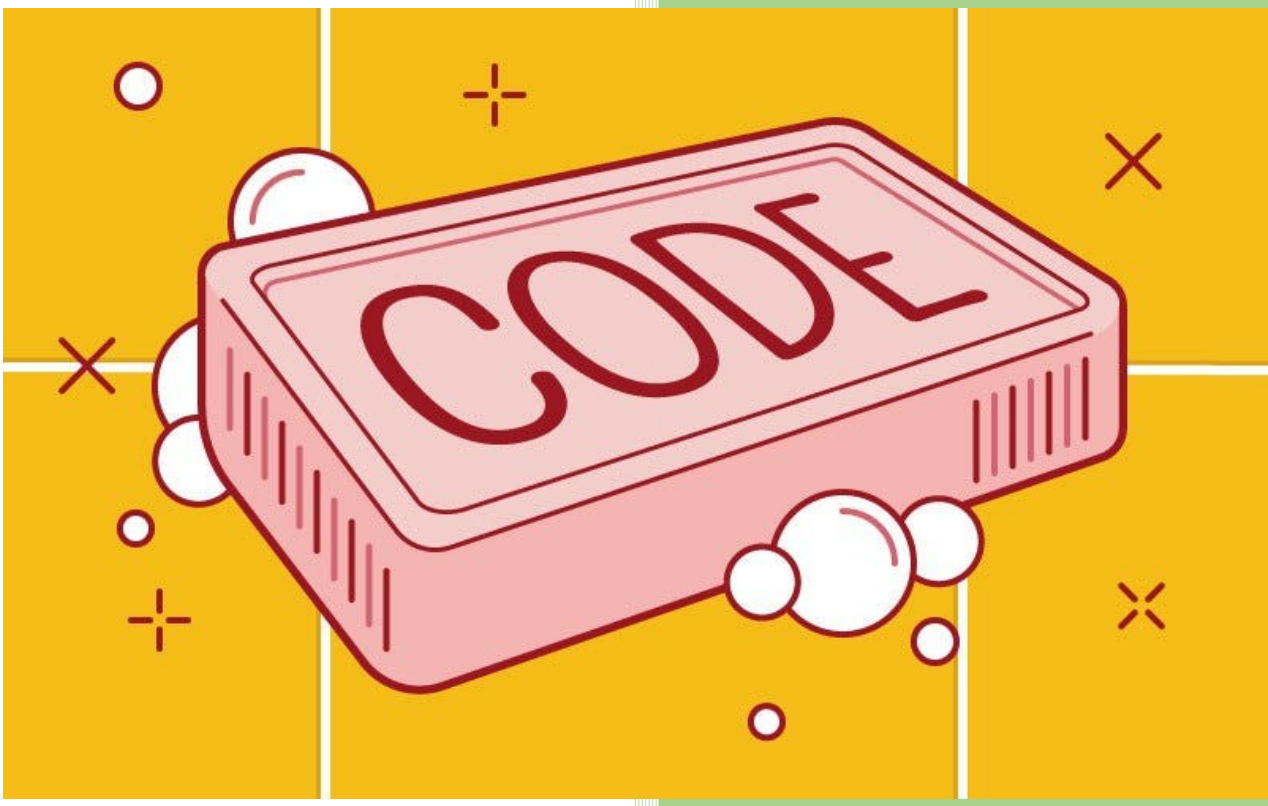
CLEAN CODE



Daniel Silva Moratilla

3/2/2024

Contenido Clean Code

BLOQUE 4. NOMBRES

1.1. Diferencias entre objetos y estructuras de datos

Mientras los objetos ocultan sus datos detrás de abstracciones y exponen funciones para operar con ellos, las estructuras de datos exponen directamente sus datos sin funciones con lógica asociada, enfatizando la importancia de no mezclar la creación de estructuras de datos con conocimiento de lógica operativa.

```
public class Estudiante {
    String NIA;
    String nombre;
    int edad;
```

Estructura de datos que expone directamente

```
public String getNIA(){
    return NIA;
}

public String getNombre() {
    return nombre;
}

public int getEdad() {
    return edad;
}
```

Objeto con abstracciones

1.2. La ley de Demeter

La Ley de Demeter sugiere que un objeto no debería conocer demasiados detalles internos de otros objetos. Debería interactuar solo con objetos cercanos inmediatos y no debería "interrumpir" la cadena de conocimiento para obtener información.

```
public class Estudiante {
    private String NIA;
    private Nombre nombre;
    private int edad;


    public String getNombre() {
        return nombre.getNombre();
    }

public class Nombre {
    private String nombre;

    public String getNombre() {
        return nombre;
    }
```

Sin seguir la ley de Demeter (Para obtener el nombre necesita la ayuda de otra clase)

```
public class Estudiante {
    private String NIA;
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }
```

Siguiendo la ley de Demeter (Obtiene directamente el nombre)

BLOQUE 5. MANEJO DE ERRORES

2.1. Usa excepciones en lugar de código de retorno

Las excepciones pueden hacer que el código sea menos transparente al permitir lanzar errores en cualquier momento. Existen riesgos de que el código que llama a una función pueda olvidar gestionar el error, lo que podría resultar en la falla de la aplicación si no se maneja adecuadamente.

```
public String getNombre() {
    return nombre;
}
```

MAL (No se comprueba el error)

```
public String getNombre() {
    if (nombre == null || nombre.isEmpty()) {
        throw new IllegalStateException(s: "El nombre del estudiante no es válido.");
    }
    return nombre;
}
```

BIEN

2.2. Escribe primero el try-catch-finally

Cuando se implementa manejo de errores con excepciones en un código, se asume que en cualquier momento el flujo del programa puede interrumpirse y saltar al bloque “catch”.

```
    public Estudiante(String NIA, String nombre, int edad) {
        if (NIA == null || NIA.isEmpty() || nombre == null || nombre.isEmpty() || edad < 0) {
            throw new IllegalArgumentException(s: "Los parámetros de creación del estudiante no son válidos.");
        }
        this.NIA = NIA;
        this.nombre = nombre;
        this.edad = edad;
    }

}
```

MAL (No se emplea ningun try-catch-finally)

```
public Estudiante(String NIA, String nombre, int edad) {
    try {
        if (NIA == null || NIA.isEmpty() || nombre == null || nombre.isEmpty() || edad < 0) {
            throw new IllegalArgumentException(s: "Los parámetros de creación del estudiante no son válidos.");
        }
        this.NIA = NIA;
        this.nombre = nombre;
        this.edad = edad;
    } catch (IllegalArgumentException e) {
        System.err.println("Error al crear el estudiante: " + e.getMessage());
        throw new MiExcepcion(mensaje:"Error al crear el estudiante.", causa: e);
    } finally {
        System.out.println(x: "Bloque finally: Operaciones finales o de limpieza.");
    }

}

private static class MiExcepcion extends RuntimeException {
    public MiExcepcion(String mensaje, Throwable causa) {
        super(message:mensaje, cause: causa);
    }
}
```

BIEN

2.3. Usa excepciones unchecked

Utilizar excepciones "unchecked" puede ofrecer mayor flexibilidad y reducir la dependencia del código en cambios específicos de excepciones

```
public int getEdad() {
    try {
        if (edad < 0) {
            throw new NullPointerException(s: "La edad del estudiante no es válida.");
        }
        return edad;
    } catch (NullPointerException e) {
        System.err.println("Error al obtener la edad del estudiante: " + e.getMessage());
        throw new MiExcepcion(mensaje:"Error al obtener la edad del estudiante.", causa: e);
    }

}
```

MAL (getEdad() no refleja el cambio porque NullPointerException no es una excepción verificada)

```
public int getEdad() {
    try {
        if (edad < 0) {
            throw new IllegalStateException(s: "La edad del estudiante no es válida.");
        }
        return edad;
    } catch (IllegalStateException e) {
        System.err.println("Error al obtener la edad del estudiante: " + e.getMessage());
        throw new MiExcepcion(mensaje:"Error al obtener la edad del estudiante.", causa: e);
    }
}
```

BIEN (Ahora sí es una es una excepción unchecked, por lo que no requieren declaración en la firma del método o que se manejen específicamente)

2.4. No devuelvas Null

Devolver nulos puede ser peligroso, ya que el código que llama a una función no tiene garantía de si recibirá un valor no nulo. Es preferible adoptar estrategias alternativas, como el uso de valores opcionales o el manejo explícito de casos especiales, para evitar la propagación de nulos.

```
public int getEdad() {
    try {
        if (edad < 0) {
            throw new NullPointerException(s: "La edad del estudiante no es válida.");
        }
        return edad;
    } catch (NullPointerException e) {
        System.err.println("Error al obtener la edad del estudiante: " + e.getMessage());
        throw new MiExcepcion(mensaje:"Error al obtener la edad del estudiante.", causa: e);
    }

}
```

MAL (Si la edad es 0 se está lanzando un NullPointerException, lo cual queremos evitar)

```
public int getEdad() {
    try {
        if (edad < 0) {
            throw new IllegalStateException(s: "La edad del estudiante no es válida.");
        }
        return edad;
    } catch (IllegalStateException e) {
        System.err.println("Error al obtener la edad del estudiante: " + e.getMessage());
        throw new MiExcepcion(mensaje:"Error al obtener la edad del estudiante.", causa: e);
    }
}
```

BIEN (Se lanza una excepción si la edad es menor a cero, en lugar de devolver un valor nulo)

BLOQUE 6: TEST UNITARIOS

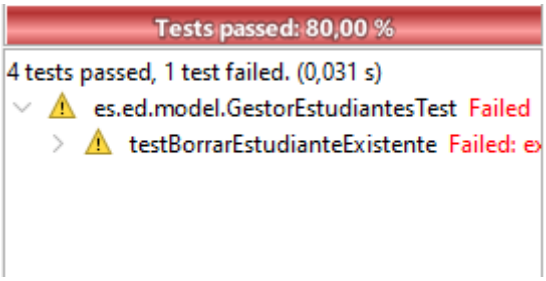
3.1. Las tres leyes del TDD

1. Escribe un test que falle antes de escribir código de producción.
2. No escribas más de un test unitario que falle a la vez (y que no compile).
3. Escribe solo el código de producción necesario para que el test que falla pase.

```
@Test
public void testBorrarEstudianteExistente() {
    int cantidadInicial = gestorEstudiantes.obtenerEstudiantes().length;
    gestorEstudiantes.borrarEstudiante(nia:"99999999");
    int cantidadDespuesDeBorrar = gestorEstudiantes.obtenerEstudiantes().length;
    assertEquals(cantidadInicial - 1, actual: cantidadDespuesDeBorrar);
}
```

Ley 1, Escribe un test que falle antes de escribir código de producción (el estudiante con nia "99999999" no existe)

Ley 2. No escribas más de un test unitario que falle a la vez (y que no compile) (Hacemos que solo falle ese)



ley 3 Escribe solo el código de producción necesario para que el test que falla pase.

```
public void borrarEstudiante(String nia) {
    for (int i = 0; i < cantidadEstudiantes; i++) {
        if (estudiantes[i].getNIA().equals(anObject:nia)) {
            // Mover los elementos siguientes hacia atrás
            System.arraycopy(src:estudiantes, i + 1, dest:estudiantes, destPos:i, cantidadEstudiantes - i - 1);
            estudiantes[cantidadEstudiantes - 1] = null; // Limpiar el último elemento
            cantidadEstudiantes--;
            break;
        }
    }
}
```

3.2. Mantén limpios los test

Un conjunto de tests ordenado mejora la legibilidad y mantiene la integridad del conjunto de pruebas a medida que el código evoluciona.

```
public class GestorEstudiantesTest {
    private GestorEstudiantes gestorEstudiantes;

    @BeforeAll
    public static void setUpBeforeAll() {...3 lines }

      @Test
    public void testBorrarEstudianteExistente() {...6 lines }


    @AfterAll
    public static void tearDownAfterAll() {...3 lines }

    @BeforeEach
    public void setUp() {...8 lines }

     @Test
    public void testVerificarMenorDeEdad() {...3 lines }

    @AfterEach
    public void tearDown() {...3 lines }

      @Test
    public void testVerificarMayorDeEdad() {...3 lines }

    @Test
    public void testAgregarEstudiante() {...6 lines }


    @Test
    public void testBorrarEstudianteNoExistente() {...6 lines }
```

MAL (Los test están totalmente desordenados y entrecruzados con @AfterAll, @BeforeEach, @AfterEach y @BeforeAll)

```
public class GestorEstudiantesTest {
    private GestorEstudiantes gestorEstudiantes;

    @BeforeAll
    public static void setUpBeforeAll() {...3 lines }

    @AfterAll
    public static void tearDownAfterAll() {...3 lines }

    @BeforeEach
    public void setUp() {...8 lines }

    @AfterEach
    public void tearDown() {...3 lines }

    @Test
    public void testAgregarEstudiante() {...6 lines }

    @Test
    public void testBorrarEstudianteExistente() {...6 lines }

    @Test
    public void testBorrarEstudianteNoExistente() {...6 lines }

    @Test
    public void testVerificarMayorDeEdad() {...3 lines }

    @Test
    public void testVerificarMenorDeEdad() {...3 lines }

}
```

BIEN

3.3. Clean Tests

El Clean Code se debería aplicar a los test también

```
@Test
public void prueba1() {
    int a = gestorEstudiantes.obtenerEstudiantes().length;
    gestorEstudiantes.borrarEstudiante(nia:"llllllll");
    int b = gestorEstudiantes.obtenerEstudiantes().length;

    assertEquals(a - 1, actual: b);
}
```

MAL

```
@Test
public void testBorrarEstudianteExistente() {
    int cantidadInicial = gestorEstudiantes.obtenerEstudiantes().length;
    gestorEstudiantes.borrarEstudiante(nia:"llllllll");
    int cantidadDespuesDeBorrar = gestorEstudiantes.obtenerEstudiantes().length;
    assertEquals(cantidadInicial - 1, actual: cantidadDespuesDeBorrar);
}
```

BIEN

3.4. Un Assert por test

Es importante que cada test compruebe solo una cosa, sino y el test falla, no sabremos por qué

```
@Test
public void testVerificarMayorDeEdad() {
    assertTrue(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"llllllll"));
    assertFalse(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"lllllll2"));
}
```

MAL

```
@Test
public void testVerificarMayorDeEdad() {
    assertTrue(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"llllllll"));
}
```

BIEN

3.5. Un único concepto por test

Cada test debería comprobar una única cosa

```
@Test
public void deberiaAgregarEstudianteYVerificarMayorDeEdad() {
    // Arrange (Preparación)
    int cantidadInicial = gestorEstudiantes.obtenerEstudiantes().length;

    // Act (Actuación)
    gestorEstudiantes.agregarEstudiante(new Estudiante(NIA:"44444444", nombre: "Ana", edad: 20));

    // Assert (Afirmación)
    int cantidadDespuesDeAgregar = gestorEstudiantes.obtenerEstudiantes().length;
    assertTrue(cantidadDespuesDeAgregar > cantidadInicial);

    assertTrue(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"44444444"));
}
```

MAL (Comprueba la edad y agrega el estudiante a la vez)

```
@Test
public void testAgregarEstudiante() {
    int cantidadInicial = gestorEstudiantes.obtenerEstudiantes().length;
    gestorEstudiantes.agregarEstudiante(new Estudiante(NIA:"33333333", nombre: "Pedro", edad: 21));
    int cantidadDespuesDeAgregar = gestorEstudiantes.obtenerEstudiantes().length;
    assertEquals(cantidadInicial + 1, actual: cantidadDespuesDeAgregar);
}

@Test
public void testVerificarMayorDeEdad() {
    assertTrue(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"llllllll"));
}
```

BIEN (Tenemos un test para agregar el estudiante y otro para verificar la mayoría de edad)

3.6. La regla *F.I.R.S.T.*

Un test debe ser:

- **Rápido** (Fast)
- **Independiente**: para que no importe el orden de ejecución
- **Repetible**: que se puedan repetir en cualquier entorno.
- **Auto-validable** (Self-Validating): Los tests fallan o pasan, no deberíamos tener que comprobar nada extra para asegurarnos de que fue correcto.
- **Oportuno** (Timely): deberían ser escritos justo antes del código de producción. Si lo escribes después, el código puede ser difícil de testear.

```
@Test
public void testAgregarEstudiante() {
    int cantidadInicial = gestorEstudiantes.obtenerEstudiantes().length;
    gestorEstudiantes.agregarEstudiante(new Estudiante(NIA:"55555555", nombre: "Lucía", edad: 25));
    int cantidadDespuesDeAgregar = gestorEstudiantes.obtenerEstudiantes().length;
    assertEquals(cantidadInicial + 1, actual: cantidadDespuesDeAgregar);
}

@Test
public void testVerificarMayorDeEdadBasadoEnAgregarEstudiante() {
    assertTrue(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"55555555"));
}
```

MAL (Por ejemplo, aquí hacemos que un test dependa del resultado del otro, haciendo que verifique la edad de alguien con un nia que hemos agregado en un test)

```
@Test
public void testAgregarEstudiante() {
    int cantidadInicial = gestorEstudiantes.obtenerEstudiantes().length;
    gestorEstudiantes.agregarEstudiante(new Estudiante(NIA:"33333333", nombre: "Pedro", edad: 21));
    int cantidadDespuesDeAgregar = gestorEstudiantes.obtenerEstudiantes().length;
    assertEquals(cantidadInicial + 1, actual: cantidadDespuesDeAgregar);
}

@Test
public void testVerificarMayorDeEdad() {
    assertTrue(condition: gestorEstudiantes.verificarMayorDeEdad(nia:"llllllll"));
}
```

BIEN

BLOQUE 7: CLASES

4.1. Organización de clases

El contenido de las clases debería seguir esta estructura, siempre suponiendo que tengamos elementos de ese tipo:

1. Constantes públicas
2. Constantes privadas
3. Variables públicas (rara vez son una buena idea)
4. Variables privadas
5. Funciones públicas
6. Las funciones privadas que son llamadas por las públicas irían justo debajo de la pública que las llama.

```
public class Estudiante {

    public static final String CURSO = "DAM1";
    private static final int EDAD_MAYORIA_EDUCATIVA = 18;
    public int edad;
    private String NIA;
    private String nombre;

    // Constructor
    public Estudiante(String NIA, String nombre, int edad) {...16 lines }

    public String getNombre() {...6 lines }

    public String getNIA() {...6 lines }

    public int getEdad() {...11 lines }

    // Método de comportamiento
    public boolean esMayorDeEdad() {...3 lines }

    // Métodos de sobreescritura de la clase Object
    @Override
    public String toString() {...3 lines }

    @Override
    public int hashCode() {...6 lines }

    @Override
    public boolean equals(Object obj) {...20 lines }

    private static class MiExcepcion extends RuntimeException {...6 lines }
}
```

4.2. Las clases deberían ser pequeñas

Una clase debe tener un enfoque claro en su objetivo principal y no crecer indiscriminadamente.

La clase "Estudiante" tiene un objetivo claro: representar la información asociada a un estudiante

```
private boolean tieneBeca;

public boolean tieneBeca() {
    return tieneBeca;
}

public void asignarBeca() {
    this.tieneBeca = true;
}
```

MAL(La clase Estudiante ahora tendría también la función de asignar becas, lo cual no le procede)

4.3. Principio de Responsabilidad Única

Una clase o módulo debería tener una sola razón para cambiar. Es decir, una clase debe tener una única responsabilidad, y cualquier cambio en esa responsabilidad debería provocar un cambio en la clase.

```
private boolean tieneBeca;

public boolean tieneBeca() {
    return tieneBeca;
}

public void asignarBeca() {
    this.tieneBeca = true;
}
```

MAL(Siguiendo con el ejemplo anterior, si se necesitaran funciones adicionales como la gestión de becas, es más apropiado tener una clase separada para manejarlo)

```
public class Estudiante {

    public static final String CURSO = "DAM1";
    private static final int EDAD_MAYORIA_EDUCATIVA = 18;
    public int edad;
    private String NIA;
    private String nombre;

    // Constructor
    public Estudiante(String NIA, String nombre, int edad) {...16 lines }

    public String getNombre() {...6 lines }

    public String getNIA() {...6 lines }

    public int getEdad() {...11 lines }

    // Método de comportamiento
    public boolean esMayorDeEdad() {...3 lines }

    // Métodos de sobreescritura de la clase Object
    @Override
    public String toString() {...3 lines }

    @Override
    public int hashCode() {...6 lines }

    @Override
    public boolean equals(Object obj) {...20 lines }

    private static class MiExcepcion extends RuntimeException {...6 lines }
}
```

BIEN

4.4. Cohesión

Las clases deben tener un número limitado de variables de instancia, y cada método debe manipular una o más de estas variables. Buscar la máxima cohesión, donde cada variable se usa en cada método, es ideal aunque a menudo difícil de lograr. Si las clases pierden cohesión, es recomendable dividirlas para mantener un diseño claro y modular.

```
// Método de comportamiento
public boolean esMayorDeEdad() {
    return edad >= 18;
}
```

BIEN (El método "EsMayorDeEdad()" manipula la variable "edad", manteniendo la cohesión. Si en el futuro se necesitaran funciones adicionales, deberíamos crear una clase separada para manejarlas)

4.5. Organiza tu código para prepararlo para el cambio

Principios clave para el diseño de clases:

1. **Principio de Responsabilidad Única (SRP):**
    - Una clase debería tener una única razón para cambiar, evitando múltiples responsabilidades en una sola clase.
    - La apertura de una clase para realizar cambios sugiere una posible mejora de diseño.
2. **Principio Open-Closed (OCP):**
    - Las clases deben estar abiertas para la extensión pero cerradas para la modificación, permitiendo la adición de nuevas funcionalidades sin modificar el código existente.
3. **No Dependencia de Detalles de Implementación:**
    - Evitar depender de detalles internos de una clase para reducir el riesgo de cambios en la implementación que afecten al código dependiente.
4. **Principio de Inversión de Dependencias (DIP):**
    - Las abstracciones deben depender de abstracciones, no de implementaciones concretas, para reducir el acoplamiento y facilitar la flexibilidad en el cambio de implementaciones.

```java
public interface InformacionEstudiante {
    String getNIA();
    String getNombre();
    int getEdad();
}
```

```java
public class Estudiante implements InformacionEstudiante{

    public static final String CURSO = "DAM1";
    private static final int EDAD_MAYORIA_EDUCATIVA = 18;
    public int edad;
    private String NIA;
    private String nombre;

    // Constructor
    public Estudiante(String NIA, String nombre, int edad) {...16 lines }

    @Override
    public String getNombre() {...6 lines }
    @Override
    public String getNIA() {...6 lines }
    @Override
    public int getEdad() {...11 lines }
```

SRP:

Estudiante se centra en representar la información del estudiante y se introduce la interfaz InformacionEstudiante para abstraer la información necesaria

OCP:

Implementamos la interfaz InformacionEstudiante para que otras clases relacionadas con la información del estudiante extiendan sin necesidad de modificar “estudiante”

No dependencia de Detalles de Implementación:

Algunos atributos estudiante usan la interfaz, reduciendo la dependencia de detalles de implementación

DIP:

Al estudiante implementar InformacionEstudiante, cumple con la abstracción en lugar de depender de detalles pequeños

4.6. Separa la construcción de un sistema de su uso

Principios clave para la construcción de objetos y reducción del acoplamiento:

1. **Responsabilidad de Construcción:**
    - La creación o instanciación de objetos es una responsabilidad importante en el desarrollo de software.
2. **Evitar Mezcla con Lógica de Uso:**
    - Evitar mezclar la creación de objetos con el código que los utiliza para reducir el acoplamiento entre ellos.
3. **Principio de Inversión de Dependencias (DIP):**
    - Utilizar el Principio de Inversión de Dependencias para depender de abstracciones en lugar de implementaciones concretas, permitiendo flexibilidad y cambio.
4. **Inyección de Dependencias:**
    - Aplicar la inyección de dependencias como herramienta para crear y suministrar dependencias a objetos, separando la construcción de la lógica de uso.
5. **Interfaces y Testing:**
    - Trabajar con interfaces facilita la modificación del código y mejora la capacidad de realizar pruebas unitarias al reducir el acoplamiento.

4.7. Utiliza copias de objetos para trabajar con concurrencia

Principios clave para trabajar con concurrencia y objetos inmutables:

1. **Inmutabilidad en Concurrencia:**
    - En entornos concurrentes, es ideal que el estado con el que trabajan los hilos sea inmutable para evitar resultados impredecibles debido a modificaciones concurrentes.
2. **Objetos Inmutables:**
    - Un objeto inmutable no puede ser modificado después de su creación; todas sus variables son finales.
3. **Evitar Compartir Memoria Mutable:**
    - Evitar compartir objetos mutables entre hilos para prevenir cambios concurrentes.
4. **Modificar Mediante Copia:**
    - Para modificar un objeto inmutable, se crea una copia con los valores cambiados en lugar de modificar el objeto original. Esto asegura que otros procesos concurrentes no se vean afectados.

```
public static Estudiante crearEstudiante(String NIA, String nombre, int edad) {
    return new Estudiante(NIA, nombre, edad);
}
```

Para ello vamos a crear un método para crear una copia

GITHUB CON LOS ARCHIVOS JAVA

https://github.com/Zerealust-Dani/CleanCode2Ev