UT4 – OPTIMIZACIÓN Y DOCUMENTACIÓN – PARTE III- REFACTORING

Para todas las actividades, refactoriza el código y coméntalo. Además, se pide generar la documentación JAVADOC para cada ejercicio.

IMPORTANTE: Para cada ejercicio, indica qué técnica de refactorización has usado

NOTA: Si hiciera falta código auxiliar para que los ejemplos funcionen, debes añadirlo.

ACTIVIDAD 4.2

Código sin refactorizar:

```
public class Calculadora {
   public int sumar(int a, int b) {
      return a + b;
   }

   public int sumarTres(int a, int b, int c) {
      return a + b + c;
   }
}
```

ACTIVIDAD 4.3

Código sin refactorizar:

```
public class ReporteDeUsuario {
   public void imprimirReporteDeUsuario(Usuario usuario) {
      // Impresión de cabecera
      System.out.println("========== Reporte de Usuario ==========");
      System.out.println("========================================");

      // Detalles del usuario
      System.out.println("Nombre: " + usuario.getNombre());
      System.out.println("Edad: " + usuario.getEdad());
      System.out.println("Email: " + usuario.getEmail());

      // Verificación de la edad
      if(usuario.getEdad() < 18) {
         System.out.println("El usuario es menor de edad.");
      } else {
         System.out.println("El usuario es mayor de edad.");
      }

      // Pie de página del reporte
      System.out.println("========================================");
      System.out.println("Reporte generado el " + java.time.LocalDate.now());
   }
}
```

ACTIVIDAD 4.4

Código sin refactorizar:

```
public class ReporteDeEmpleado {
   public void generarReporte(String nombre, String departamento, int edad, double salario,
int añosEnLaEmpresa, boolean esGerente) {
      // Lógica para generar el reporte
      System.out.println("Generando reporte para: " + nombre);
      // Más lógica de generación de reporte...
   }
```

```
}
```

ACTIVIDAD 4.5

Código sin refactorizar:

```
public class SistemaDeReservas {
   private List<Reserva> reservas;
   private List<Cliente> clientes;

   public SistemaDeReservas() {
      this.reservas = new ArrayList<>();
      this.clientes = new ArrayList<>();
   }

   public void agregarReserva(Reserva reserva) {
      reservas.add(reserva);
   }

   public void agregarCliente(Cliente cliente) {
      clientes.add(cliente);
   }

   public Cliente buscarClientePorNombre(String nombre) {
      for (Cliente cliente : clientes) {
         if (cliente.getNombre().equals(nombre)) {
            return cliente;
         }
      }
      return null;
   }

   // Métodos para gestionar reservas
   // Métodos para gestionar clientes
}
```

ACTIVIDAD 4.6

Código sin refactorizar:

```
public class CalculadoraArea {
   public double calcularAreaCirculo(double radio) {
      double area = Math.PI * radio * radio;
      double diametro = 2 * radio;
      double circunferencia = Math.PI * diametro;
      return area;
   }
}
```

ACTIVIDAD 4.7

Código sin refactorizar:

```
public class CalculadoraDescuento {
   public double calcularDescuento(double precioOriginal) {
      return precioOriginal - (precioOriginal * 0.15);
   }
}
```

ACTIVIDAD 4.8

Código sin refactorizar:

```
public class Calculadora {
   public double realizarOperacion(String operacion, double num1, double num2) {
      switch (operacion) {
         case "suma":
            return num1 + num2;
         case "resta":
            return num1 - num2;
         case "multiplicacion":
```

```
            return num1 * num2;
        case "division":
            if (num2 != 0) {
                return num1 / num2;
            } else {
                throw new IllegalArgumentException("Divisor no puede ser cero.");
            }
        default:
            throw new UnsupportedOperationException("Operación no soportada.");
    }
  }
}
```

ACTIVIDAD 4.9

Código sin refactorizar:

```
public class ValidadorDeFormulario {
  public boolean validar(String nombre, int edad, String email) {
    if (nombre == null || nombre.isEmpty()) {
      return false;
    }
    if (edad < 18) {
      return false;
    }
    if (email == null || !email.contains("@")) {
      return false;
    }
    return true;
  }
}
```

ACTIVIDAD 4.10

Código sin refactorizar:

```
public class Calculo {
```

```
    public double calc(double[] nums) {
        double s = 0;
        for (int i = 0; i < nums.length; i++) {
            s += nums[i];
        }
        return s / nums.length;
    }
}
```

ACTIVIDAD 4.11

Código sin refactorizar:

```
public class Libro {
    private String t;
    private String a;
    private int p;

    public Libro(String t, String a, int p) {
        this.t = t;
        this.a = a;
        this.p = p;
    }

    // Métodos get y set para t, a y p
    public String getT() {
        return t;
    }

    public void setT(String t) {
        this.t = t;
    }

    public String getA() {
        return a;
    }

    public void setA(String a) {
        this.a = a;
    }
```

```
    public int getP() {
        return p;
    }

    public void setP(int p) {
        this.p = p;
    }
}
```

ACTIVIDAD 4.12

Código sin refactorizar:

```
public class CalculadoraImpuestos {
    public double calcularImpuesto(double ingreso, double tasa) {
        if (ingreso < 0 || tasa < 0 || tasa > 1) {
            throw new IllegalArgumentException("Ingreso o tasa no válidos.");
        }
        double impuesto = ingreso * tasa;
        return impuesto;
    }
}
```

SOLUCIONES

ACTIVIDAD 4.2

Código refactorizado:

```
public class Calculadora {
   public int sumar(int a, int b) {
      return a + b;
   }

   public int sumarTres(int a, int b, int c) {
      return sumar(sumar(a, b), c);
   }
}
```

Técnica de Refactorización Usada:

- ***Eliminación de Código Duplicado:*** En lugar de repetir la lógica de suma en ambos métodos, sumarTres ahora reutiliza el método sumar para sumar los dos primeros números y luego suma el resultado con el tercer número. Esto simplifica el mantenimiento del código y reduce el riesgo de errores si la lógica de suma necesita ser actualizada en el futuro.

ACTIVIDAD 4.3

Código refactorizado:

```
public class ReporteDeUsuario {

   private void imprimirCabecera() {
      System.out.println("========== Reporte de Usuario ==========");
      System.out.println("========================================");
```

```
    }

    private void imprimirDetallesUsuario(Usuario usuario) {
        System.out.println("Nombre: " + usuario.getNombre());
        System.out.println("Edad: " + usuario.getEdad());
        System.out.println("Email: " + usuario.getEmail());
    }

    private void verificarEdad(Usuario usuario) {
        if(usuario.getEdad() < 18) {
            System.out.println("El usuario es menor de edad.");
        } else {
            System.out.println("El usuario es mayor de edad.");
        }
    }

    private void imprimirPieDePagina() {
        System.out.println("=======================================");
        System.out.println("Reporte generado el " + java.time.LocalDate.now());
    }

    public void imprimirReporteDeUsuario(Usuario usuario) {
        imprimirCabecera();
        imprimirDetallesUsuario(usuario);
        verificarEdad(usuario);
        imprimirPieDePagina();
    }
}
```

Técnica de Refactorización Usada:

- **Descomposición de Método Largo:** El método *imprimirReporteDeUsuario* ha sido dividido en varios métodos más pequeños, cada uno responsable de una parte específica del reporte (cabecera, detalles del usuario, verificación de la edad, pie de página). Esto mejora la legibilidad del código y facilita su mantenimiento, ya que cada método se centra en una tarea única.

ACTIVIDAD 4.4

Código refactorizado:

Para mejorar esto, podemos introducir un objeto que encapsule todos estos parámetros.

```
public class DatosEmpleado {
   private String nombre;
   private String departamento;
   private int edad;
   private double salario;
   private int añosEnLaEmpresa;
   private boolean esGerente;

   public DatosEmpleado(String nombre, String departamento, int edad, double
salario, int añosEnLaEmpresa, boolean esGerente) {
      this.nombre = nombre;
      this.departamento = departamento;
      this.edad = edad;
      this.salario = salario;
      this.añosEnLaEmpresa = añosEnLaEmpresa;
      this.esGerente = esGerente;
   }

   // Getters para acceder a los atributos
}
```

Ahora, el método ***generarReporte*** puede aceptar un solo objeto ***DatosEmpleado. Java***

```
public class ReporteDeEmpleado {
   public void generarReporte(DatosEmpleado empleado) {
      // Lógica para generar el reporte
      System.out.println("Generando reporte para: " +
empleado.getNombre());
      // Más lógica de generación de reporte...
```

```
    }
}
```

Técnica de Refactorización Usada:

- **Introducción de un Objeto de Parámetros:** Al reemplazar la lista larga de parámetros con un objeto que los encapsula, se mejora significativamente la claridad del método. Esto no solo hace que el código sea más legible, sino que también facilita el manejo de datos del empleado, ya que ahora solo se necesita pasar un objeto en lugar de múltiples parámetros. Además, si en el futuro se necesitan más datos del empleado, se pueden agregar al objeto DatosEmpleado sin cambiar la firma del método generarReporte.

ACTIVIDAD 4.5

Código refactorizado:

Para abordar el problema de la clase grande, separamos las responsabilidades en dos clases distintas: **GestorDeClientes** para manejar los clientes y **GestorDeReservas** para manejar las reservas.

```
public class GestorDeClientes {
    private List<Cliente> clientes;

    public GestorDeClientes() {
        this.clientes = new ArrayList<>();
    }

    public void agregarCliente(Cliente cliente) {
        clientes.add(cliente);
    }

    public Cliente buscarClientePorNombre(String nombre) {
        for (Cliente cliente : clientes) {
            if (cliente.getNombre().equals(nombre)) {
                return cliente;
            }
```

```
        }
        return null;
    }
}


public class GestorDeReservas {
    private List<Reserva> reservas;

    public GestorDeReservas() {
        this.reservas = new ArrayList<>();
    }

    public void agregarReserva(Reserva reserva) {
        reservas.add(reserva);
    }

    // Otros métodos relacionados con la gestión de reservas
}
```

Finalmente, simplificamos la clase **SistemaDeReservas** para que utilice estas nuevas clases, delegando las responsabilidades adecuadamente:

```
public class SistemaDeReservas {
    private GestorDeClientes gestorDeClientes;
    private GestorDeReservas gestorDeReservas;

    public SistemaDeReservas() {
        this.gestorDeClientes = new GestorDeClientes();
        this.gestorDeReservas = new GestorDeReservas();
    }

    public void agregarCliente(Cliente cliente) {
        gestorDeClientes.agregarCliente(cliente);
    }

    public Cliente buscarClientePorNombre(String nombre) {
        return gestorDeClientes.buscarClientePorNombre(nombre);
    }
```

```
  public void agregarReserva(Reserva reserva) {
    gestorDeReservas.agregarReserva(reserva);
  }

  // Otros métodos que ahora delegan en gestorDeClientes y gestorDeReservas
}
```

Técnica de Refactorización Usada:

- **Separación de Responsabilidades (Clases muy grandes):** La refactorización crea clases específicas (GestorDeClientes y GestorDeReservas) para separar las responsabilidades de manejo de clientes y reservas. Esto no solo mejora la organización del código y su mantenibilidad, sino que también hace que SistemaDeReservas sea más modular, facilitando futuras extensiones o modificaciones en cada área de responsabilidad de manera independiente.

ACTIVIDAD 4.6

En este caso, las variables **diametro** y **circunferencia** se calculan, pero nunca se utilizan para nada útil en el método.

Código refactorizado:

```
public class CalculadoraArea {
  public double calcularAreaCirculo(double radio) {
    return Math.PI * radio * radio;
  }
}
```

Técnica de Refactorización Usada:

- **Eliminación de Código Muerto:** Se eliminaron las líneas que definían y calculaban diametro y circunferencia, ya que no contribuían al propósito del método. Esto simplifica el método, mejorando la claridad y eficiencia del código al eliminar operaciones innecesarias.

ACTIVIDAD 4.7

El código incluye "números mágicos", los cuales son valores numéricos directos que carecen de explicación o contexto, haciendo el código difícil de entender y mantener.

El valor 0.15 representa un descuento del 15%, pero esto no es inmediatamente claro para alguien que lee el código.

Código refactorizado:

```
public class CalculadoraDescuento {
   private static final double DESCUENTO = 0.15; // 15% de descuento

   public double calcularDescuento(double precioOriginal) {
      return precioOriginal - (precioOriginal * DESCUENTO);
   }
}
```

Técnica de Refactorización Usada:

- **Reemplazar Números Mágicos con Constantes Simbólicas**: Se extrajo el valor 0.15 a una constante con un nombre descriptivo. Esto mejora la legibilidad y mantenibilidad del código, facilitando futuras modificaciones al valor del descuento.

ACTIVIDAD 4.8

Para mejorar la legibilidad y mantenibilidad, se pueden separar las operaciones en métodos dedicados.

Código refactorizado:

```
public class Calculadora {
   public double realizarOperacion(String operacion, double num1, double num2) {
      switch (operacion) {
         case "suma":
            return suma(num1, num2);
         case "resta":
            return resta(num1, num2);
         case "multiplicacion":
            return multiplicacion(num1, num2);
         case "division":
            return division(num1, num2);
         default:
            throw new UnsupportedOperationException("Operación no soportada.");
      }
   }

   private double suma(double num1, double num2) {
      return num1 + num2;
   }

   private double resta(double num1, double num2) {
      return num1 - num2;
   }

   private double multiplicacion(double num1, double num2) {
      return num1 * num2;
   }

   private double division(double num1, double num2) {
      if (num2 == 0) {
         throw new IllegalArgumentException("Divisor no puede ser cero.");
      }
      return num1 / num2;
   }
}
```

Técnica de Refactorización Usada:

- **Separación de Responsabilidades:** Cada operación matemática se maneja en su propio método, mejorando la claridad y facilitando la modificación o extensión de cada operación de forma independiente.

ACTIVIDAD 4.9

Para mejorar la legibilidad y mantenibilidad, se pueden separar las validaciones en métodos privados.

Código refactorizado:

```
public class ValidadorDeFormulario {
   public boolean validar(String nombre, int edad, String email) {
      return esNombreValido(nombre) && esEdadValida(edad) &&
esEmailValido(email);

   }

   private boolean esNombreValido(String nombre) {
      return nombre != null && !nombre.isEmpty();
   }

   private boolean esEdadValida(int edad) {
      return edad >= 18;
   }

   private boolean esEmailValido(String email) {
      return email != null && email.contains("@");
   }
}
```

Técnica de Refactorización Usada:

- **Descomposición de Método Complejo:** Separa la lógica compleja en métodos más pequeños y descriptivos, lo cual mejora la legibilidad y simplifica el proceso de pruebas unitarias para cada condición de validación.

ACTIVIDAD 4.10

Para mejorar la claridad y legibilidad del código, renombramos la clase, el método, y las variables con nombres más descriptivos.

Código refactorizado:

```
public class CalculadoraPromedio {
   public double calcularPromedio(double[] numeros) {
      double sumaTotal = 0;
      for (int indice = 0; indice < numeros.length; indice++) {
         sumaTotal += numeros[indice];
      }
      return sumaTotal / numeros.length;
   }
}
```

Técnica de Refactorización Usada:

- **Renombramiento para Mayor Claridad:** Cambiar los nombres de clases, métodos, y variables por otros más descriptivos mejora significativamente la legibilidad del código. Este cambio facilita la comprensión del propósito y la lógica del método calcularPromedio, reduciendo la curva de aprendizaje y los errores potenciales para otros desarrolladores (o incluso para el autor original al volver a este código después de un tiempo).

ACTIVIDAD 4.11

Para hacer el código más claro y mantenible, podemos renombrar los atributos y métodos para que sean más descriptivos y reflejen su propósito real.

Código refactorizado:

```
public class Libro {
   private String titulo;
```

```
    private String autor;
    private int paginas;

    public Libro(String titulo, String autor, int paginas) {
        this.titulo = titulo;
        this.autor = autor;
        this.paginas = paginas;
    }

    // Métodos get y set renombrados para reflejar los nombres de atributos descriptivos
    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public int getPaginas() {
        return paginas;
    }

    public void setPaginas(int paginas) {
        this.paginas = paginas;
    }
}
```

Técnica de Refactorización Usada:

- **Renombramiento para Mayor Claridad:** Al cambiar los nombres de los atributos y métodos por otros más descriptivos, este ejercicio mejora la claridad y legibilidad del código. Los nuevos nombres, titulo, autor y páginas, comunican claramente el propósito y la naturaleza de cada atributo en la clase Libro. Esto no solo facilita la lectura y comprensión del código, sino que también mejora la experiencia de mantenimiento y extensión de este por parte de otros desarrolladores.

ACTIVIDAD 4.12

Para mejorar la claridad y la responsabilidad única del método, se pueden extraer la validación de los datos de entrada en un método separado.

Código refactorizado:

```
public class CalculadoraImpuestos {
    public double calcularImpuesto(double ingreso, double tasa) {
        validarEntradas(ingreso, tasa);
        return ingreso * tasa;
    }

    private void validarEntradas(double ingreso, double tasa) {
        if (ingreso < 0 || tasa < 0 || tasa > 1) {
            throw new IllegalArgumentException("Ingreso o tasa no válidos.");
        }
    }
}
```

Técnica de Refactorización Usada:

- **Extracción de Método:** Se extrajo la lógica de validación de datos de entrada a un método privado validarEntradas. Esto hace que el método calcularImpuesto sea más conciso y se centre en una única tarea: el cálculo del impuesto. La extracción de método mejora la legibilidad del código y facilita tanto el mantenimiento como la

posibilidad de reutilizar la validación de entradas en otros contextos dentro de la misma clase.