Taller de diseño de software

```
// Rectangle.java
//Clase dominio
public class Rectangle {
private final double width;
private final double height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
public double getWidth() { return width; }
public double getHeight() { return height; }
}

// Circle.java
public class Circle {
    private final double radius;
    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() { return radius; }
}

//Clase de Lógica de Negocio:
// AreaCalculator.java
public class AreaCalculator {
        public double sum(Object[] shapes) {
            double totalArea = 0;
            for (Object shape : shapes) {
                if (shape instanceof Rectangle) {
                        Rectangle rect = (Rectangle) shape;
                        totalArea += rect.getWidth() * rect.getHeight();
            }
                if (shape instanceof Circle) {
```

```
                Circle circle = (Circle) shape;
                totalArea += Math.PI * circle.getRadius() * circle.getRadius();
            }
        }
        return totalArea;
    }
}
```

Punto de Discusión Inicial:

Considere el requisito de añadir una nueva figura, como Triangle. Analice el impacto de este
cambio en el sistema actual, específicamente en la clase AreaCalculator. Evalúe la
escalabilidad y el riesgo de regresión de este diseño.

Respuesta:

El metodo AreaCalculator no esta mal, de hecho funcionaria bien si se le agrega el objeto Triangle, pero si lo pones en una situación de escalabilidad y riesgo de regresión, optaria por una Interfaz en vez de una clase para AreaCalculator, además de cambiar el nombre a la interfaz a Shapes, y poner el mismo metodo, pero llamado AreaCalculator, así las clases que se van ingresando puede implementar el metodo si lo requieren y cada uno implementaria el algoritmo necesario para sacar el area.

Proceso de Refactorización Guiada

La refactorización se realizará en dos fases, cada una enfocada en un principio SOLID.

2.1. Aplicación del Principio de Responsabilidad Única (SRP)

El SRP establece que una clase debe tener una única razón para cambiar. Actualmente, la
clase AreaCalculator conoce la lógica interna para calcular el área de cada

figura, violando la
cohesión y el encapsulamiento.

● Acción: Delegue la responsabilidad del cálculo del área a cada clase de figura

1. Implemente un método público getArea() en la clase Rectangle.

```
// Rectangle.java
public class Rectangle {
    // ... campos y constructor existentes ...
        public double getArea() {
            return getWidth() * getHeight();
        }
}
```

2. Implemente un método análogo getArea() en la clase Circle.

```
// Circle.java
public class Circle {
// ... campo y constructor existentes ...
        public double getArea() {
            return Math.PI * getRadius() * getRadius();
        }
}
```

2.2. Aplicación del Principio Abierto/Cerrado (OCP)

● Acción: Introduzca una abstracción de la que dependan tanto el módulo de alto nivel
(AreaCalculator) como los de bajo nivel (Rectangle, Circle).

1. Defina una interfaz Shape que declare un contrato único: un método double getArea().

```
public interface Shape{
        public double getArea(){}
}
```

2. Modifique las clases Rectangle y Circle para que implementen la interfaz Shape.

1. Implemente un método público getArea() en la clase Rectangle.

```
// Rectangle.java
public class Rectangle implements Shape {
   // ... campos y constructor existentes ...
      public double getArea() {
         return getWidth() * getHeight();
      }
}
```

2. Implemente un método análogo getArea() en la clase Circle.

```
// Circle.java
public class Circle implements Shape {
// ... campo y constructor existentes ...
      public double getArea() {
         return Math.PI * getRadius() * getRadius();
      }
}
```

3. Refactorice el método sum en AreaCalculator para que acepte un arreglo de tipo
   Shape. El nuevo método debe iterar sobre la colección y, mediante polimorfismo,
   invocar el método getArea() de cada objeto, eliminando por completo las estructuras condicionales (if) y la evaluación de tipos (instanceof).

```
// AreaCalculator.java
public class AreaCalculator {
      public double sum(Shape[] shapes) {
         double totalArea = 0;
         for (Shape shape : shapes) {
                  totalArea += shape.getArea();
         }

      }
   return totalArea;
   }
}
```

Fase de Validación y Discusión

Para validar la efectividad de la refactorización, se introducirá un nuevo requisito.

● Ejercicio de Validación:

1. Implemente una nueva clase, Triangle, que se conforme a la interfaz Shape.

```
public class Triangle implements Shape{
        private final double height;
        private final double width;

        public Triangle(double height, double width){
        this.height = height;
        this.width = width;
        }

        public double getWidth() { return width; }
        public double getHeight() { return height; }

        public double getArea(){
           return (height * width)/2
        }
}
```

2. Incorpore una instancia de Triangle en el conjunto de datos de prueba.

   R/= Se tendria que hacer desde el archivo main o donde se hagan las pruebas para poder instanciarlo.

3. Verifique que el sistema procesa la nueva figura correctamente sin realizar ninguna modificación en la clase AreaCalculator.

Puntos de Discusión Final:

1. Compare la arquitectura inicial con la final en términos de acoplamiento y cohesión.

R/= En acomplamiento, la primera era un dependiente de la clase AreaCalculator y en cohesión estaban bien estructurada ya que cada responsabilidad es coherente a la clase.

- La estrucura final, tiene un acomplamiento bajo y una cohesión alta, y no dependen de AreaCalculator.

2. Explique cómo el diseño refactorizado utiliza el polimorfismo para cumplir con el Principio Abierto/Cerrado.

R/= Cada clase va a implementar el metodo de interface por lo que cada uno puede implementar su propia logica, sin dañar la firma, y otra clase puede implementarlo si lo ve necesario, además cuando se quiera utilizar ese metodo de la clase, es la misma acción para las demás clases por lo que se aplicaria bien el polimorfismo.

3. Analice cómo este nuevo diseño ejemplifica el Principio de Inversión de Dependencias (DIP), donde los módulos de alto nivel no dependen de los de bajo nivel, sino que ambos dependen de abstracciones.

Nota:

Gran parte del taller ya estaba resuelto o estaba guiado