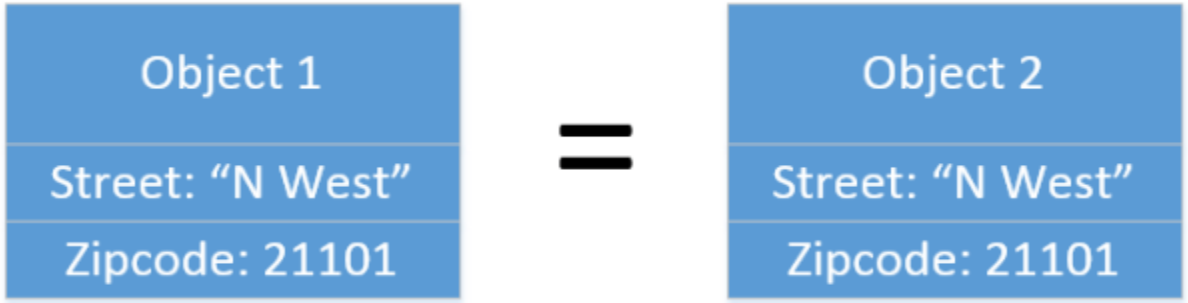
Clases Conceptuales: “Entity vs. Value Object”

- *Las Entidades* o *clases* del dominio de mi problema tienen un identificador, son modificables y comparables por *Identidad*.

Value Object:

- Son comparables por contenido (igualdad estructural), no tienen identificador.



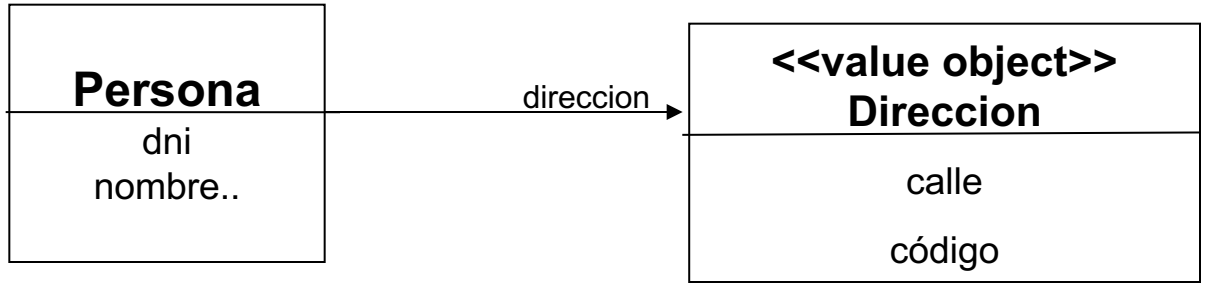
- No viven por si mismos, necesitan una entidad base, son intercambiables (un billete de 100 Pesos AR lo puedo cambiar por otro). Persisten adjunto a su base, no separadamente.
- Inmutables (No le defino *setters*).

Clases Conceptuales: “Entity vs. Value Object”

- Deben ser inmutables. No puede cambiar en su vida. Caso contrario NO es un un Value Object.
- ¿Cómo identificar un *Value Object* en el modelo?

Cuando necesitamos por ej. modelar Moneda, Fecha, Dirección, que puedan tener cierto comportamiento (getters..)

¿Cómo representar “Value Objects” en el modelo?



<<Value Object>> delante del nombre

Ejemplo en Java

```java
package vo;

import java.util.Objects;

public class DireccionPostal {

    private final String calle;
    private final String altura;
    private final String codigoPostal;
    private final String localidad;

    public DireccionPostal(String calle, String altura,
            String codigoPostal, String localidad) {
        //pueden incluirse validaciones de los parámetros.
        this.calle = calle;
        this.altura = altura;
        this.codigoPostal = codigoPostal;
        this.localidad = localidad;
    }

    // Getters Dependiendo del contexto.
    public String getCalle() {
        return calle;
    }
```

Ejemplo en Java

```java
package vo;

import java.util.Objects;

public class DireccionPostal {

    private final String calle
    private final String altur
    private final String codig
    private final String local

    public DireccionPostal(Str
            String codigoPosta
        //pueden incluirse val
        this.calle = calle;
        this.altura = altura;
        this.codigoPostal = co
        this.localidad = locali
    }

    // Getters Dependiendo del contexto.
    public String getCalle() {
        return calle;
    }
```

```java
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        DireccionPostal that = (DireccionPostal) o;
        return calle.equals(that.calle) &&
                altura.equals(that.altura) &&
                codigoPostal.equals(that.codigoPostal) &&
                localidad.equals(that.localidad);
    }

    @Override
    public int hashCode() {
        return Objects.hash(calle, altura, codigoPostal, localidad);
    }
```

Ejemplo en Java

```java
package vo;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class DireccionPostalTest {

    private DireccionPostal casa, home, biblioPublica;

    @BeforeEach
    public void setUp() {
        this.casa = new DireccionPostal("50 y 120", "s/n", "1900", "La Plata");
        this.home = new DireccionPostal("50 y 120", "s/n", "1900", "La Plata");
        this.biblioPublica = new DireccionPostal("Plaza Rocha", "137", "1900", "La Plata");
    }

    @Test
    void testEquals() {
        assertEquals(this.casa, this.home);
        assertNotEquals(this.casa, this.biblioPublica);
    }

}

```

Agregando Heurísticas para Asignación de responsabilidades (HAR)

Polimorfismo

Descripción: cuando el comportamiento varía según el tipo, asigne la responsabilidad a los tipos/las clases para las que varía el comportamiento.

Ejemplo: Se debe soportar distintas bonificaciones de pago con tarjeta de crédito.

(ya visto previamente)

… Como la bonificación del pago varía según el tipo de tarjeta, deberíamos asignarle la responsabilidad de la bonificación a los distintos tipos de tarjeta.

- Nos permite sustituir objetos que tienen idéntica interfaz.

“No hables con extraños”

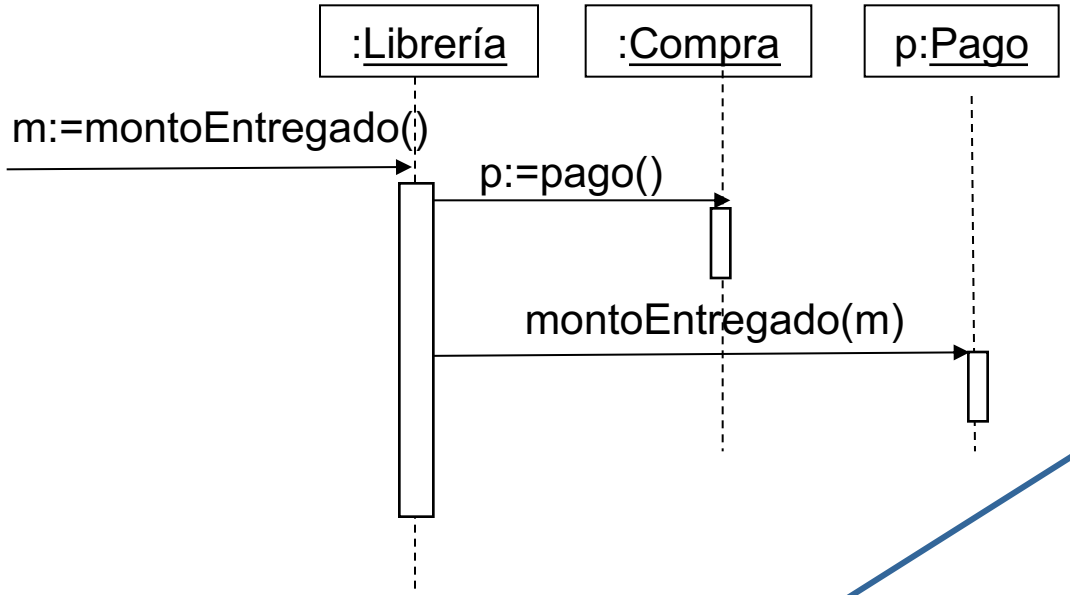
Descripción: Evite diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños).

Dentro de un método sólo pueden enviarse mensajes a objetos conocidos:

- Self/this
- un parámetro del método
- un objeto que esté asociado a self/this
- un miembro de una colección que sea atributo de self/this
- un objeto creado dentro del método

Los demás objetos son extraños (strangers)

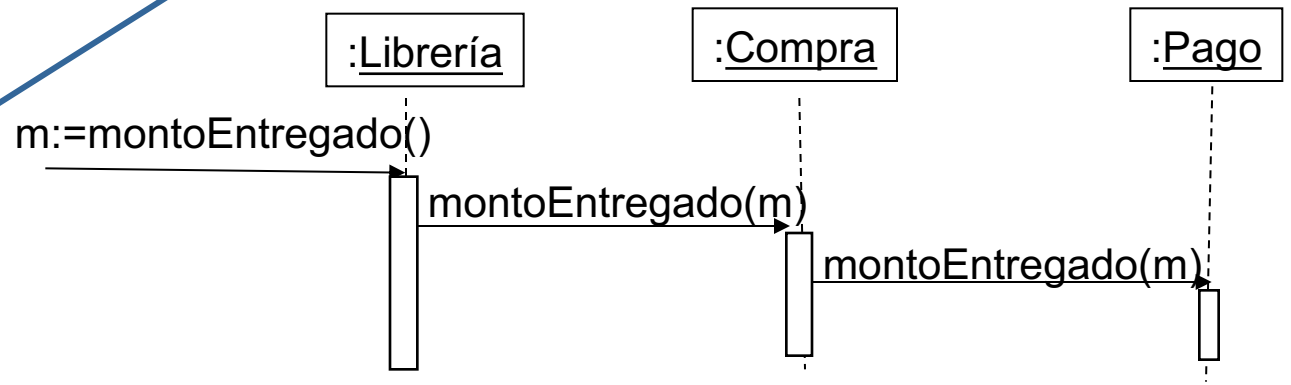
“No hables con extraños”



¿Dónde se da la envidia?
¿De qué tipo?

Mejor

Peor



Reuso de Código

Herencia vs. Composición

Herencia de Clases

- Herencia total: debo conocer todo el código que se hereda -> Reutilización de Caja Blanca
- Usualmente debemos redefinir
- Los cambios en la superclase se propagan automáticamente a las subclases
- Herencia de Estructura vs. Herencia de comportamiento
- Es útil para extender la funcionalidad del dominio de aplicación

Composición de Objetos

- Los objetos se componen en forma Dinámica -> Reutilización de Caja Negra
- Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

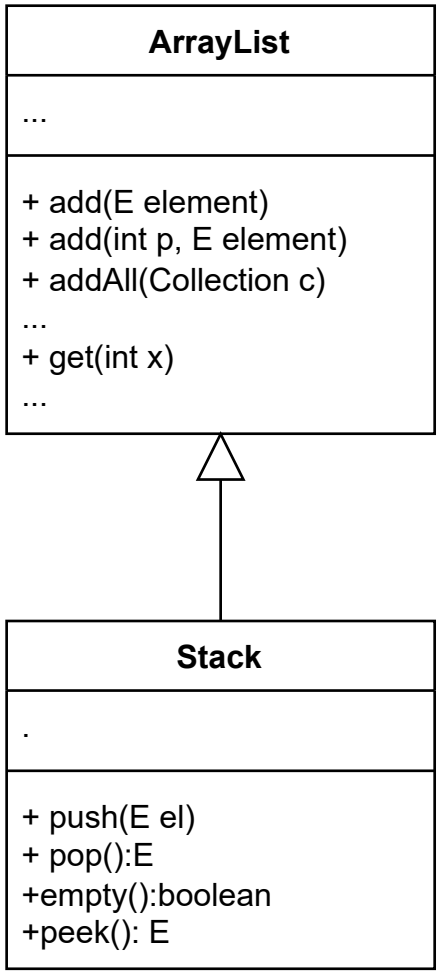
Herencia vs. Composición de Objetos

- Las clases y los objetos creados mediante herencia están *estrechamente acoplados* ya que cambiar algo en la superclase afecta directamente a la/las subclases.

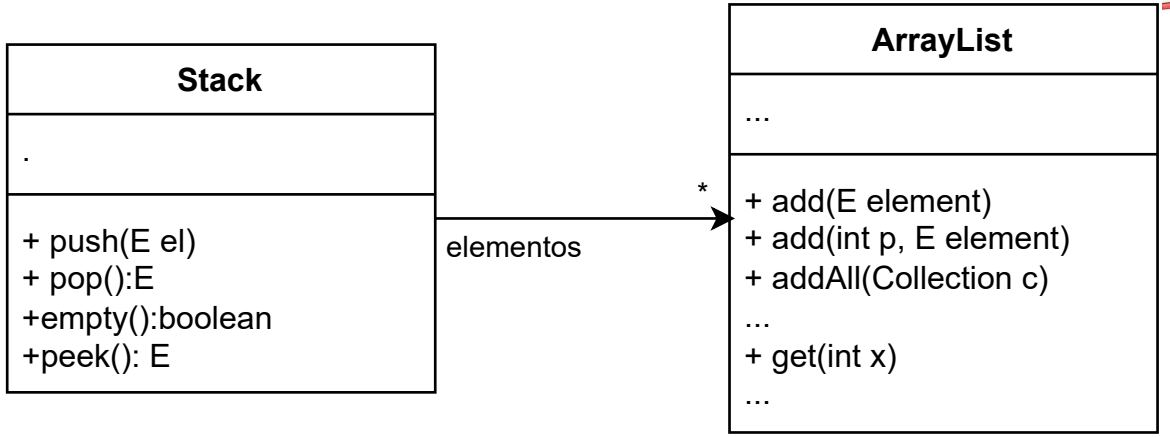
- Las clases y los objetos creados a través de la composición están *débilmente acoplados*, lo que significa que se pueden cambiar más fácilmente los componentes sin afectar el objeto contenedor.

Veamos un ejemplo

Herencia



Composición



Cómo hacer un mal uso de la herencia

```
import java.util.ArrayList;

 public class Stack<T> extends ArrayList<T> {

     public void push(T object) {
         this.add(object);
     }
     public T pop() {
         return this.remove(this.size() - 1);
     }
     public boolean empty() {
         return this.size() == 0;
     }
     public T peek() {
         return this.get(this.size() - 1);
     }
}
```

Cómo hacer un mal uso de la herencia

- Esta clase ***Stack*** funcionará como una pila, pero su interfaz es *voluminosa*; está formada por mensajes que hay que anular o redefinir!!

- La interfaz pública de esta clase no es solo push y pop, (esperable para una Pila), también incluye
- add en cualquier posición por índice,
- remove de una posición a otra,
- clear y muchos otros mensajes heredados de ArrayList que son inapropiados para una Pila.

Cómo hacer un mal uso de la herencia

El ejemplo tiene errores de diseño

- uno semántico:

"una pila es una ArrayList" no es cierto; Stack no es un subtipo adecuado de ArrayList (No cumple con Is-a). Se supone que una pila aplica el último en entrar, primero en salir, una restricción que se satisface con la interfaz push/pop, pero que no se cumple con la interfaz de ArrayList que es mucho más extensa.

- Otro mecánico:

heredar de ArrayList viola el encapsulaminto; usar ArrayList para contener la colección de objetos de la pila es una opción de implementación que puede y debe ocultarse.

Composición, no Herencia

```
import java.util.ArrayList;

 public class Stack<T> {
     private ArrayList<T> elementos


     public void push(T object) {
         elementos.add(object);
    }
     public T pop() {
        return elementos.remove(elementos.size() - 1);
    }
     public boolean empty() {
         return elementos.size() == 0;
    }
     public T peek() {
         return elementos.get(elementos.size() - 1);
    }
}
```

Cómo solucionar un mal uso de la herencia

Composición, no herencia…

- Mecánicamente, heredar de ArrayList no cumple con el encapsulamiento; en cambio,

- **componer con ArrayList para contener la colección de objetos de la pila es una opción de implementación que permite ocultarla públicamente.**

- En este caso, en vez de heredar, el uso o composición permite reuso y mantiene el encapsulamiento!!

Heurísticas para Diseño “ágil” Orientado a Objetos

(Principios S O L I D)

Principios para Diseño “ágil” Orientado a Objetos

Principios S O L I D

Relacionados a las **HAR**, para un buen estilo de DOO.
Promueven Alta Cohesión y Bajo Acoplamiento.

Robert C. Martin (2014) Agile Software Development, Principles, Patterns, and Practices. Pearson New International Edition (Capítulos 8 al 12)

S SRP: The Single-Responsibility Principle

Principio de Responsabilidad única. Una clase debería cambiar por una sola razón.

Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión).

O OCP: The Open-Closed Principle

Entidades de software (clases, módulos, funciones, etc.) deberían ser “abiertas” para extensión, y “cerradas” para modificación.

Abierto a extensión: ser capaz de añadir nuevas funcionalidades.

Cerrado a modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente.

Principios para Diseño “ágil” Orientado a Objetos

L LSP: The Liskov Substitution Principle

Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Es decir, que si el programa utiliza una clase (clase A), y ésta es extendida (clases B, C, D, etc…) el programa tiene que poder utilizar cualquiera de sus subclases y seguir siendo válido. Uso correcto de herencia (Is-a) y polimorfismo.

I ISP: The Interface-Segregation Principle

Las clases que tienen interfaces “voluminosas” son clases cuyas interfaces no son cohesivas.

Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deben estar forzadas a incluir métodos que no va a utilizar.

Principios para Diseño “ágil” Orientado a Objetos

D DIP: The Dependency-Inversion Principle

a. Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.

b. Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

Módulos de alto nivel: se refieren a los objetos que definen qué es y qué hace el sistema.
Módulos de bajo nivel: no están directamente relacionados con la lógica de negocio del programa (no definen el dominio). Por ejemplo, el mecanismo de persistencia o el acceso a red .

Abstracciones: se refieren a protocolos (o interfaces) o clases abstractas.

Detalles: son las implementaciones concretas, (cuál mecanismo de persistencia, etc).

Ser capaz de «*invertir*» una dependencia es lo mismo que ser capaz de «*intercambiar*» una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.

Single Responsibility

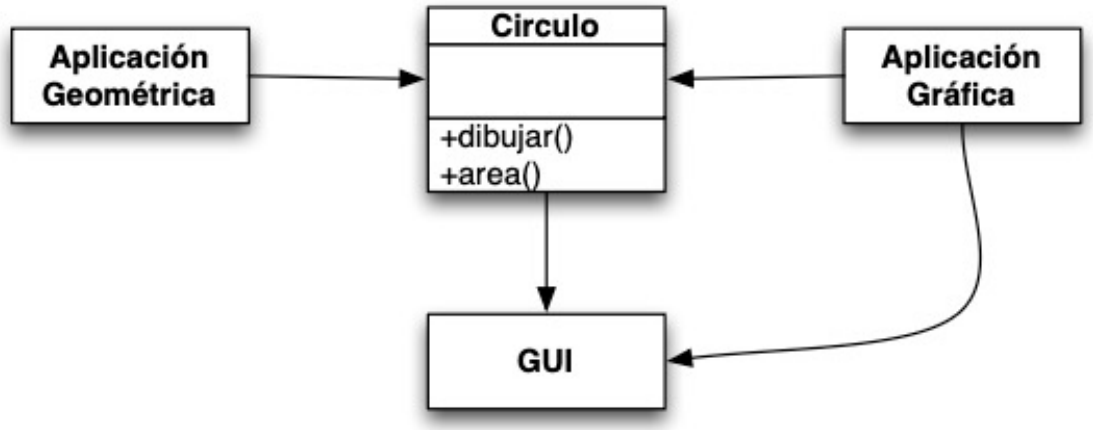


FIGURA: Clase Círculo

- La aplicación geométrica utiliza solamente el comportamiento relacionado a computar el area del círculo y jamas lo dibujará.
- La Aplicación Gráfica utiliza la opción de dibujar en la interfaz de usuario (GUI) al círculo.

Single Responsibility

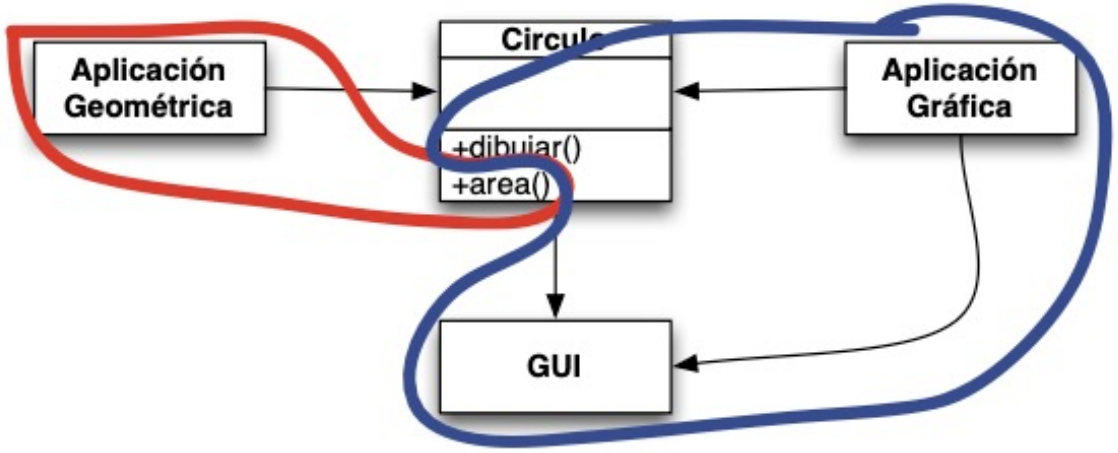


FIGURA: Clase Círculo

EL DISEÑO VIOLA EL PRINCIPIO DE SRP YA QUE TIENE DOS RESPONSABILIDADES

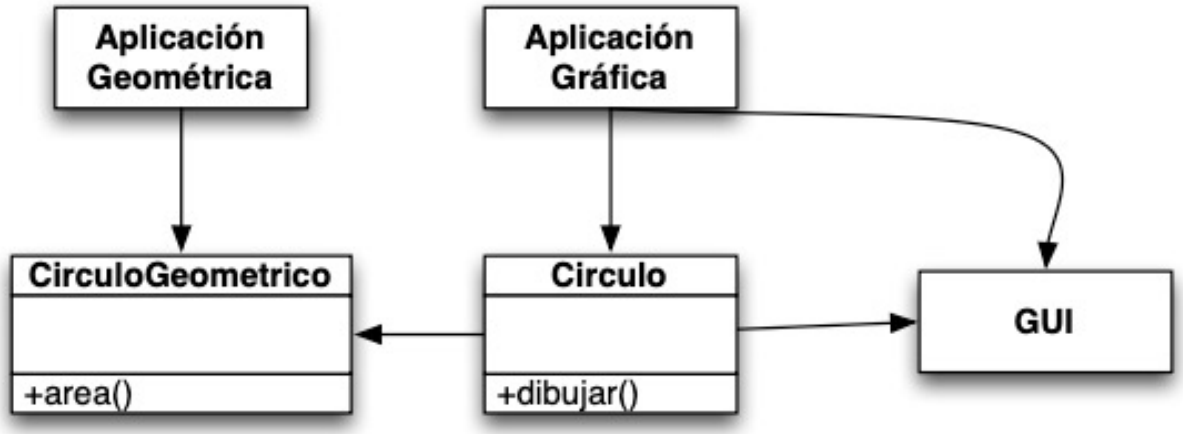
- La primera modela la representación geométrica del círculo
- La segunda es para renderizar el círculo en la interfáz gráfica.

Single Responsibility - Consecuencias

La violación del SRP genera varios problemas

- debemos incluir la GUI en la aplicacion de cálculo geométrico. En algunos lenguajes, como Java o .net debe sessre ensamblada para poder realizar el deploy con la aplicación geometrica.
- si un cambio en la aplicación Grafica causa que el circulo debe cambiar por alguna razon, entonces el cambio nos obliga a recompilar, re testear, y re-deployar la aplicacion de computación geometrica. Si olvidamos hacer esto, la aplicación puede romperse de formas poco predecibles.

Single Responsibility - Solución



Un mejor diseño es separar en las responsabilidades en dos clases totalmente diferentes. Este diseño mueve la porción computacional del círculo a la clase CirculoGeometrico. Ahora los cambios sobre la manera en que se dibuja el rectangulo no pueden afectar a la aplicación geométrica.

Single Responsibility - Conclusiones

- El principio de Única Responsabilidad (SRP) es uno de los principios más simples de comprender pero el más difícil de aplicar. Juntar responsabilidades es algo que realizamos naturalmente.
- Encontrar y separar esas responsabilidades es algo más relacionado a lo que el diseño de software en verdad realiza. El resto de los principios que discutiremos vuelven de una forma a este.

Open - Closed Principle - Algunas prácticas asociadas

- "Todas las variables de instancia deben ser privadas"
- "Deben evitarse las variables globales"
- Hacer uso de identificadores de tipos no es una buena práctica (ej. Variable de Instancia String tipo)

> "Todos los sistemas cambian durante su ciclo de vida. Debemos considerar esto para hacer nuestros sistemas lo más durables posible"
>
> — Ivar Jacobson

Open - Closed Principle

Abiertas para ser extendidas

Significa que el comportamiento se debe poder extender.

Cerradas para la modificación

El código de la clase es inviolable. Nadie puede alterar el código fuente de la misma.

Open - Closed Principle

Abiertas para ser extendidas

Significa que el comportamiento se debe poder extender.

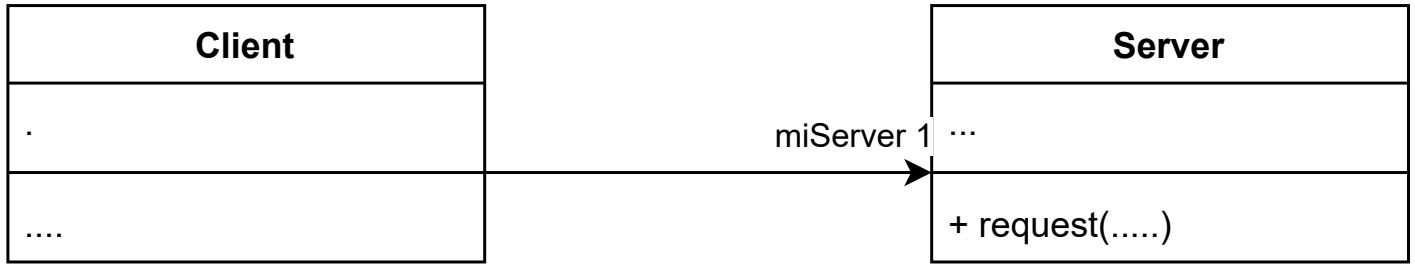
Cerradas para la modificación

El código de la clase es inviolable. Nadie puede alterar el código fuente de la misma.

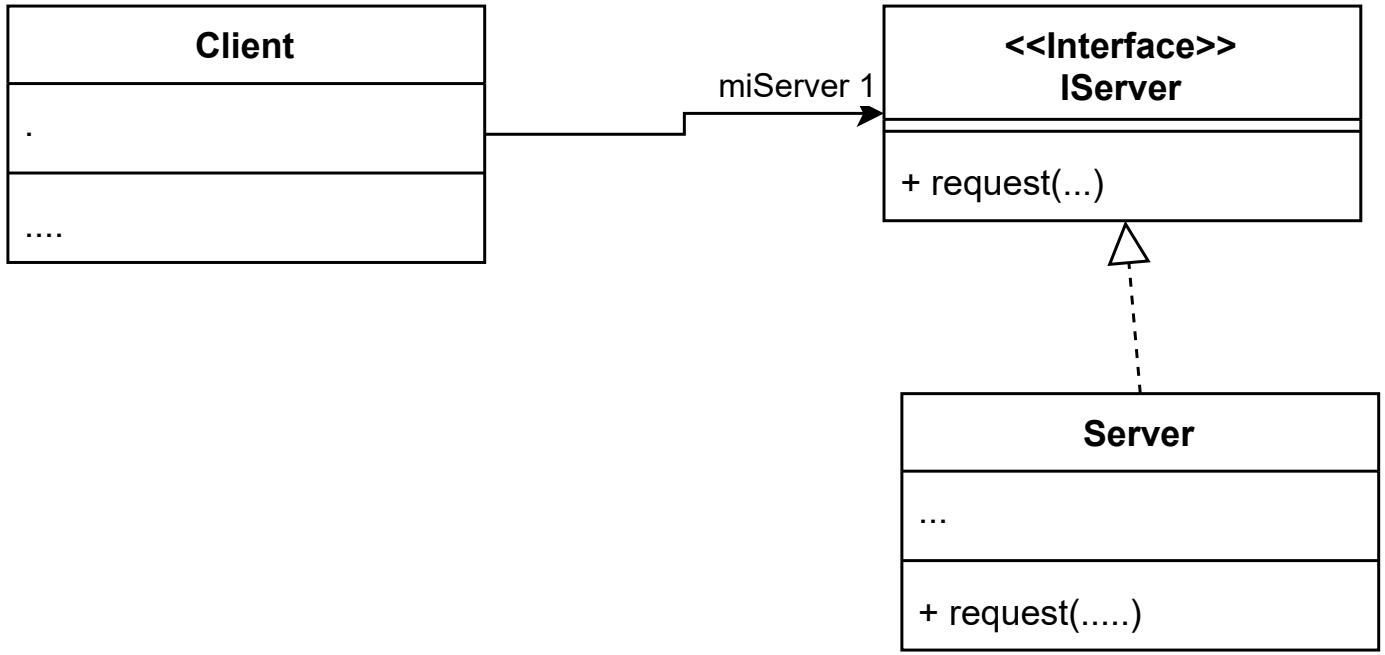
Open – Closed Principle

```
1.public class Mensaje {
2.    private String contenido;
3.    private String tipo; // "normal", "censurado",
                "encriptado"
4.

5.    public Mensaje(String contenido, String tipo) {
6.        this.contenido = contenido;
7.        this.tipo = tipo;
8.    }
9.

10.    public String getTipo() {
11.        return tipo;
12.    }
13.

14.    public String procesar() {
15.        String resultado = contenido;
16.

17.        if (tipo.equals("censurado") ||
               tipo.equals("encriptado")) {
18.            String[] malasPalabras = {"tonto", "feo"};
19.            for (String palabra : malasPalabras) {
20.                String censura =
"*".repeat(palabra.length());
21.                resultado = resultado.replaceAll("(?i)" +
palabra, censura);
22.            }
23.        }
24.

25.        if (tipo.equals("encriptado")) {
26.            resultado = new
StringBuilder(resultado).reverse().toString();
27.        }
28.

29.        return resultado;
30.    }
31.}
32.
```

Open – Closed Principle – Mirando el diseño



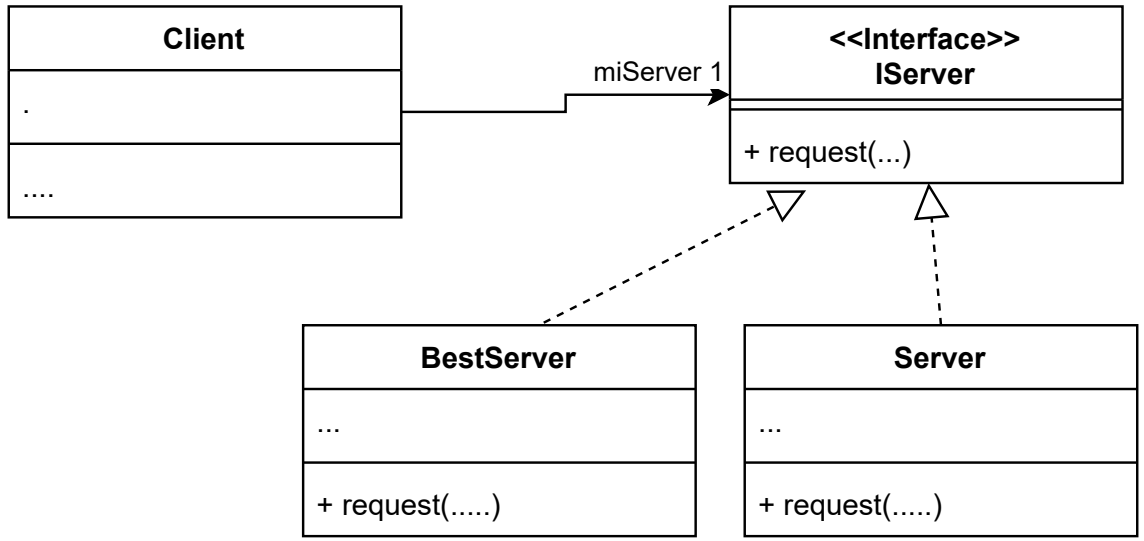
¿Cuál respeta mejor el OCP?



Open – Closed Principle – Mirando el diseño y el código

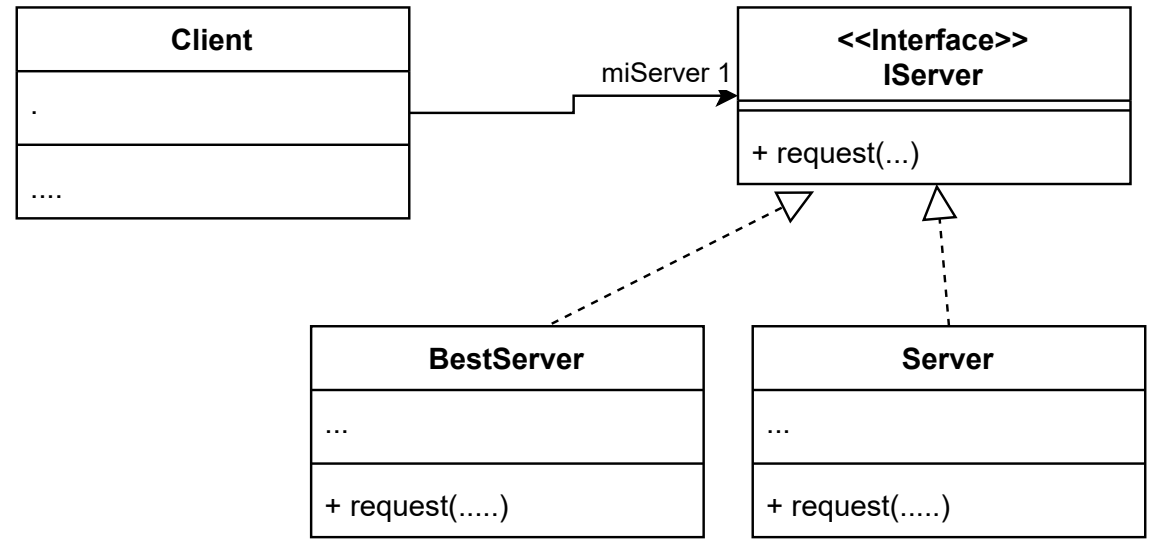
```java
package model;

public interface IServer {

    public void request(String value);

}
```

```java
package model;

public class Server implements IServer{

    @Override
    public void request(String value) {
        //....

    }

}
```



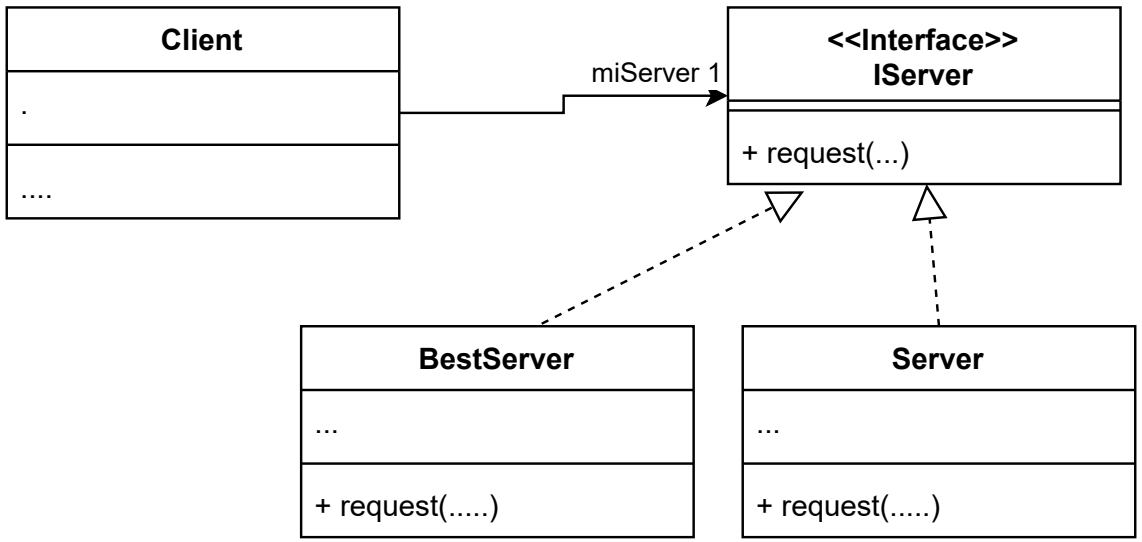
Open - Closed Principle - Mirando el diseño y el código

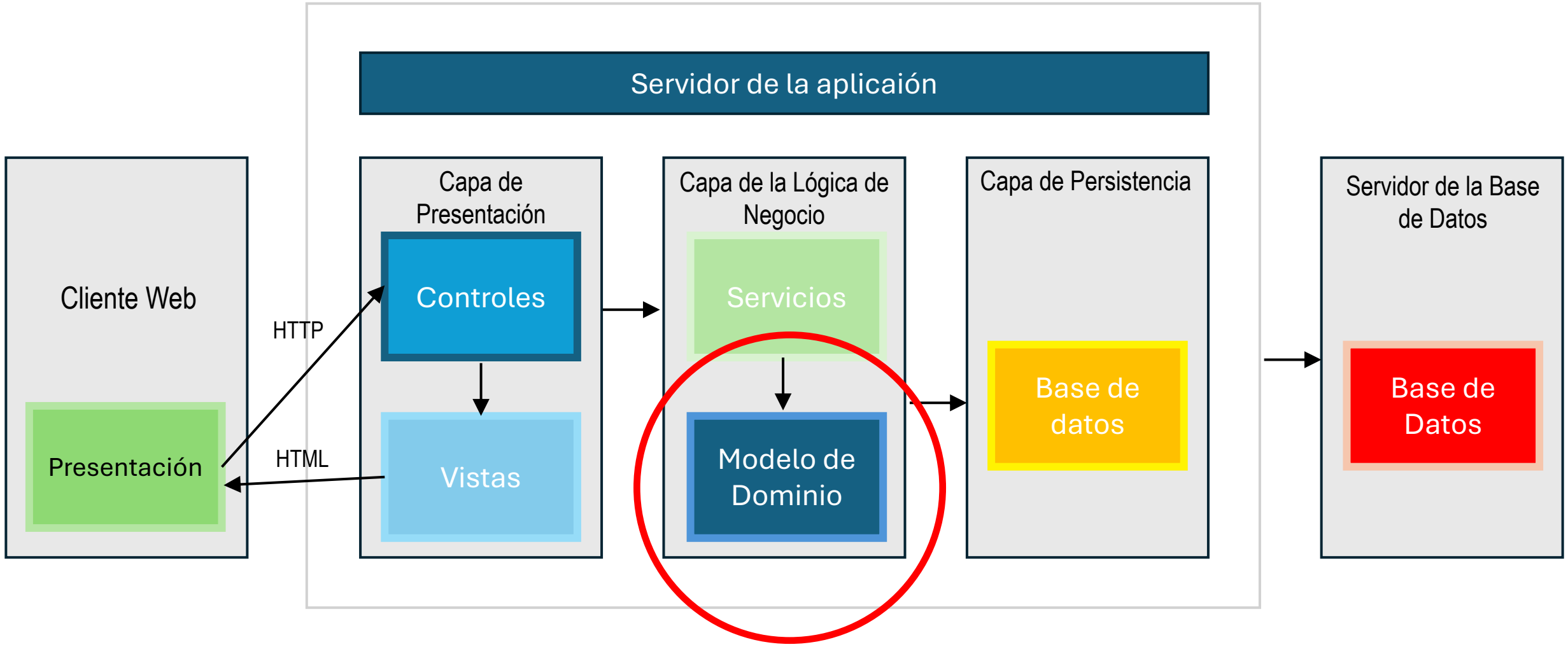
```java
package model;

public class Client {

    IServer miServer;

    public Client() {
        this.miServer=new Server();
    }

    public void generarRequest(String string) {
        this.miServer.request(string);
    }

}
```



Open - Closed Principle - Mirando el diseño y el código

```java
package model;

public class Client {

    IServer miServer;

    public Client(IServer server) {
        this.miServer=server;
    }

    public void generarRequest(String string) {
        this.miServer.request(string);
    }

}
```




Servidor de la aplicaión
Cliente Web
Presentación
HTTP
HTML
Capa de Presentación
Controles
Vistas
Capa de la Lógica de Negocio
Servicios
Modelo de Dominio
Capa de Persistencia
Base de datos
Servidor de la Base de Datos
Base de Datos